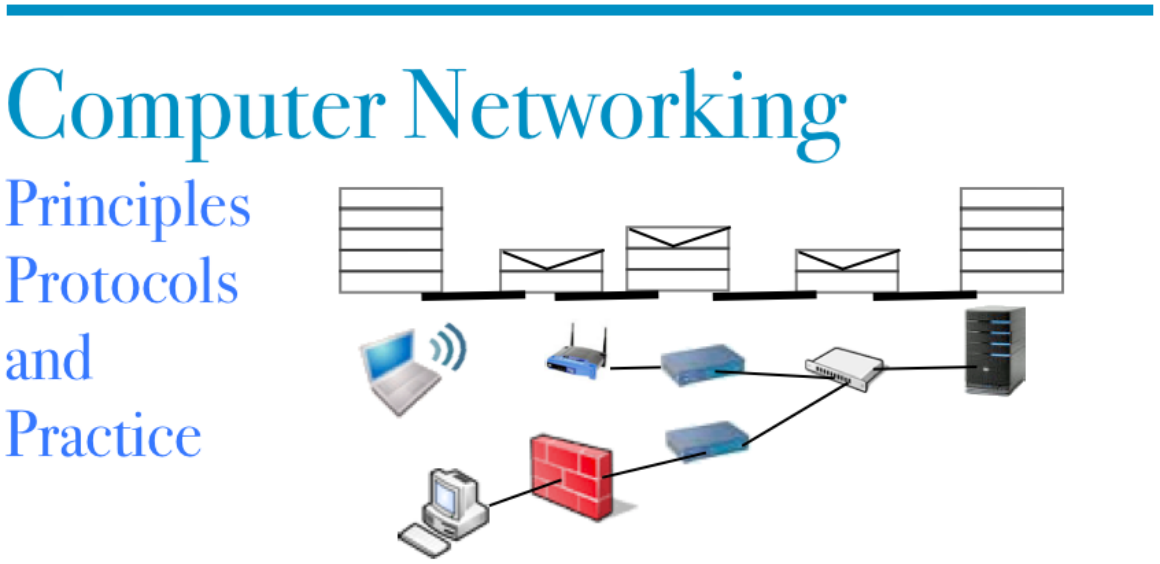


Computer Networking : Principles, Protocols and Practice

Release

Olivier Bonaventure

May 30, 2014

1 Table of Contents **3**
1.1 Preface . . . 3

2 Part 1: Principles **5**
2.1 Connecting two hosts . . . 5
2.2 Building a network . . . 25
2.3 Applications . . . 53
2.4 The transport layer . . . 56
2.5 Naming and addressing . . . 71
2.6 Sharing resources . . . 75
2.7 The reference models . . . 107

3 Part 2: Protocols **111**
3.1 The application layer . . . 111
3.2 The Domain Name System . . . 113
3.3 Electronic mail . . . 116
3.4 The HyperText Transfer Protocol . . . 125
3.5 Remote Procedure Calls . . . 134
3.6 Internet transport protocols . . . 137
3.7 The User Datagram Protocol . . . 138
3.8 The Transmission Control Protocol . . . 139
3.9 The Stream Control Transmission Protocol . . . 156
3.10 Congestion control . . . 161
3.11 The network layer . . . 167
3.12 The IPv6 subnet . . . 185
3.13 Routing in IP networks . . . 191
3.14 Intradomain routing . . . 192
3.15 Interdomain routing . . . 197
3.16 Datalink layer technologies . . . 210

4 Part 3: Practice **229**
4.1 Reliable transfer . . . 229
4.2 Building a network . . . 231
4.3 Serving applications . . . 237
4.4 Sharing resources . . . 244
4.5 Application layer . . . 252
4.6 Configuring DNS and HTTP servers . . . 255
4.7 Experimenting with Internet transport protocols . . . 257
4.8 Experimenting with Internet congestion control . . . 264
4.9 Configuring IPv6 . . . 266
4.10 IP Address Assignment Methods and Intradomain Routing . . . 270
4.11 Inter-domain routing and BGP . . . 276

4.12 Local Area Networks: The Spanning Tree Protocol and Virtual LANs 285

5 Appendices **289**
5.1 Glossary 289
5.2 Bibliography 293
5.3 Indices and tables 293

Bibliography **295**

Index **309**

Computer Networking

Principles Protocols and Practice

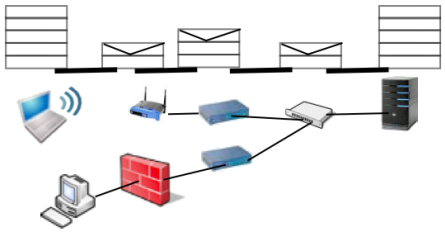


Table of Contents

1.1 Preface

This is the current draft of the second edition of the *Computer Networking : Principles, Protocols and Practice*. The document is updated every week.

The first edition of this ebook has been written by Olivier Bonaventure. Laurent Vanbever, Virginie Van den Schriek, Damien Saucez and Mickael Hoerdt have contributed to exercises. Pierre Reinbold designed the icons used to represent switches and Nipaul Long has redrawn many figures in the SVG format. Stephane Bortzmeyer sent many suggestions and corrections to the text. Additional information about the textbook is available at http://inl.info.ucl.ac.be/CNP3

1.1.1 About the author

Olivier Bonaventure is currently professor at Universite catholique de Louvain (Belgium) where he leads the IP Networking Lab and is vice-president of the ICTEAM institute. His research has been focused on Internet protocols for more than twenty years. Together with his Ph.D. students, he has developed traffic engineering techniques, performed various types of Internet measurements, improved the performance of routing protocols such as BGP and IS-IS and participated to the development of new Internet protocols including shim6, LISP and Multipath TCP. He frequently contributes to standardisation within the IETF. He was on the editorial board of IEEE/ACM Transactions on Networking and is Education Director of ACM SIGCOMM.

CHAPTER 2

Part 1: Principles

2.1 Connecting two hosts

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=1

The first step when building a network, even a worldwide network such as the Internet, is to connect two hosts together. This is illustrated in the figure below.

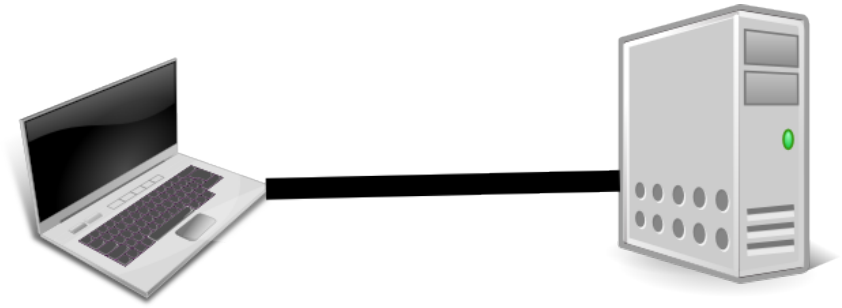


Figure 2.1: Connecting two hosts together

To enable the two hosts to exchange information, they need to be linked together by some kind of physical media. Computer networks have used various types of physical media to exchange information, notably :

- *electrical cable*. Information can be transmitted over different types of electrical cables. The most common ones are the twisted pairs (that are used in the telephone network, but also in enterprise networks) and the coaxial cables (that are still used in cable TV networks, but are no longer used in enterprise networks). Some networking technologies operate over the classical electrical cable.
- *optical fiber*. Optical fibers are frequently used in public and enterprise networks when the distance between the communication devices is larger than one kilometer. There are two main types of optical fibers : multimode and monomode. Multimode is much cheaper than monomode fiber because a LED can be used to send a signal over a multimode fiber while a monomode fiber must be driven by a laser. Due to the different modes of propagation of light, monomode fibers are limited to distances of a few kilometers while multimode fibers can be used over distances greater than several tens of kilometers. In both cases, repeaters can be used to regenerate the optical signal at one endpoint of a fiber to send it over another fiber.
- *wireless*. In this case, a radio signal is used to encode the information exchanged between the communicating devices. Many types of modulation techniques are used to send information over a wireless channel and there is lot of innovation in this field with new techniques appearing every year. While most wireless networks rely on radio signals, some use a laser that sends light pulses to a remote detector. These optical

techniques allow to create point-to-point links while radio-based techniques, depending on the directionality of the antennas, can be used to build networks containing devices spread over a small geographical area.

2.1.1 The physical layer

These physical media can be used to exchange information once this information has been converted into a suitable electrical signal. Entire telecommunication courses and textbooks are devoted to the problem of converting analog or digital information into an electrical signal so that it can be transmitted over a given physical *link*. In this book, we only consider two very simple schemes that allow to transmit information over an electrical cable. This enables us to highlight the key problems when transmitting information over a physical link. We are only interested in techniques that allow to transmit digital information through the wire and will focus on the transmission of bits, i.e. either *0* or *1*.

Note: Bit rate

In computer networks, the bit rate of the physical layer is always expressed in bits per second. One Mbps is one million bits per second and one Gbps is one billion bits per second. This is in contrast with memory specifications that are usually expressed in bytes (8 bits), KiloBytes (1024 bytes) or MegaBytes (1048576 bytes). Thus transferring one MByte through a 1 Mbps link lasts 8.39 seconds.

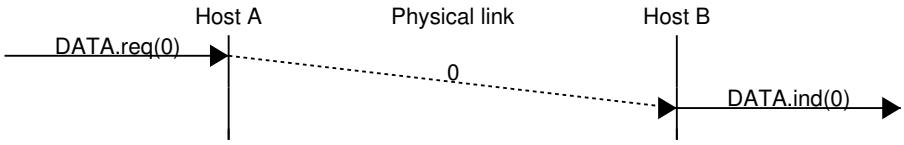
Bit rate	Bits per second
1 Kbps	10^3
1 Mbps	10^6
1 Gbps	10^9
1 Tbps	10^{12}

To understand some of the principles behind the physical transmission of information, let us consider the simple case of an electrical wire that is used to transmit bits. Assume that the two communicating hosts want to transmit one thousand bits per second. To transmit these bits, the two hosts can agree on the following rules :

- **On the sender side :**
  - set the voltage on the electrical wire at `+5V` during one millisecond to transmit a bit set to *1*
  - set the voltage on the electrical wire at `−5V` during one millisecond to transmit a bit set to *0*
- **On the receiver side :**
  - every millisecond, record the voltage applied on the electrical wire. If the voltage is set to `+5V`, record the reception of bit *1*. Otherwise, record the reception of bit *0*

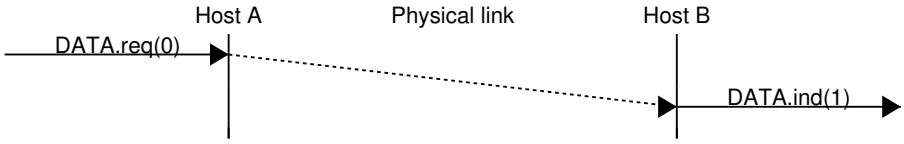
This transmission scheme has been used in some early networks. We use it as a basis to understand how hosts communicate. From a Computer Science viewpoint, dealing with voltages is unusual. Computer scientists frequently rely on models that enable them to reason about the issues that they face without having to consider all implementation details. The physical transmission scheme described above can be represented by using a *time-sequence diagram*.

A *time-sequence diagram* describes the interactions between communicating hosts. By convention, the communicating hosts are represented in the left and right parts of the diagram while the electrical link occupies the middle of the diagram. In such a time-sequence diagram, time flows from the top to the bottom of the diagram. The transmission of one bit of information is represented by three arrows. Starting from the left, the first horizontal arrow represents the request to transmit one bit of information. This request is represented by using a *primitive* which can be considered as a kind of procedure call. This primitive has one parameter (the bit being transmitted) and a name (*DATA.request* in this example). By convention, all primitives that are named *something.request* correspond to a request to transmit some information. The dashed arrow indicates the transmission of the corresponding electrical signal on the wire. Electrical and optical signals do not travel instantaneously. The diagonal dashed arrow indicates that it takes some time for the electrical signal to be transmitted from *Host A* to *Host B*. Upon reception of the electrical signal, the electronics on *Host B*'s network interface detects the voltage and converts it into a bit. This bit is delivered as a *DATA.indication* primitive. All primitives that are named *something.indication* correspond to the reception of some information. The dashed lines also represents the relationship between two (or more) primitives. Such a time-sequence diagram provides information about the ordering of the different primitives, but the distance between two primitives does not represent a precise amount of time.

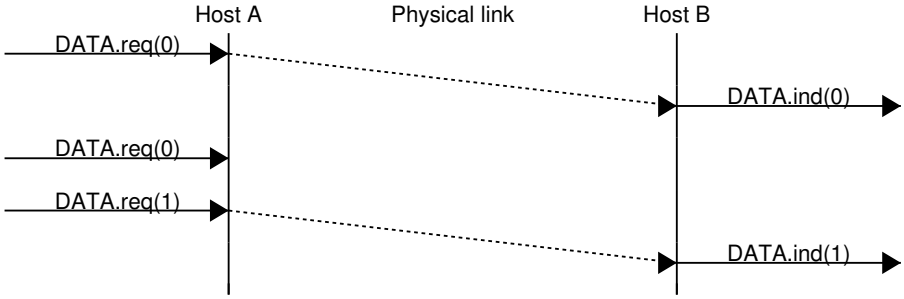


Time-sequence diagrams are usual when trying to understand the characteristics of a given communication scheme. When considering the above transmission scheme, is it useful to evaluate whether this scheme allows the two communicating hosts to reliably exchange information ? A digital transmission will be considered as reliable when a sequence of bits that is transmitted by a host is received correctly at the other end of the wire. In practice, achieving perfect reliability when transmitting information using the above scheme is difficult. Several problems can occur with such a transmission scheme.

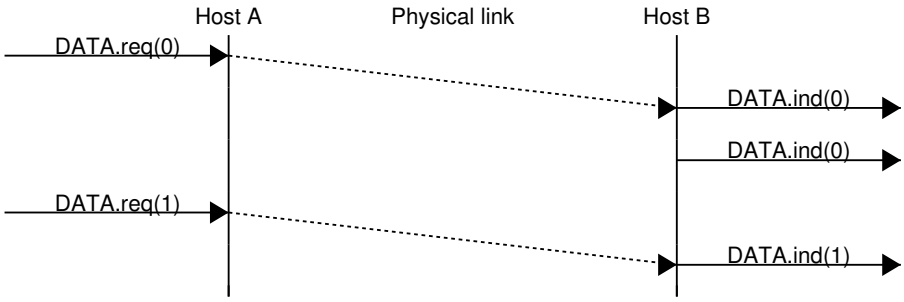
The first problem is that electrical transmission can be affected by electromagnetic interferences. These interferences can have various sources including natural phenomenons like thunderstorms, variations of the magnetic field, but also can be caused by interference with other electrical signals such as interference from neighboring cables, interferences from neighboring antennas, ... Due to all these interferences, there is unfortunately no guarantee that when a host transmit one bit on a wire, the same bit is received at the other end. This is illustrated in the figure below where a *DATA.request(0)* on the left host leads to a *Data.indication(1)* on the right host.



With the above transmission scheme, a bit is transmitted by setting the voltage on the electrical cable to a specific value during some period of time. We have seen that due to electromagnetic interferences, the voltage measured by the receiver can differ from the voltage set by the transmitter. This is the main cause of transmission errors. However, this is not the only type of problem that can occur. Besides defining the voltages for bits *0* and *1*, the above transmission scheme also specifies the duration of each bit. If one million bits are sent every second, then each bit lasts 1 microsecond. On each host, the transmission (resp. the reception) of each bit is triggered by a local clock having a 1 MHz frequency. These clocks are the second source of problems when transmitting bits over a wire. Although the two clocks have the same specification, they run on different hosts, possibly at a different temperature and with a different source of energy. In practice, it is possible that the two clocks do not operate at exactly the same frequency. Assume that the clock of the transmitting host operates at exactly 1000000 Hz while the receiving clock operates at 999999 Hz. This is a very small difference between the two clocks. However, when using the clock to transmit bits, this difference is important. With its 1000000 Hz clock, the transmitting host will generate one million bits during a period of one second. During the same period, the receiving host will sense the wire 999999 times and thus will receive one bit less than the bits originally transmitted. This small difference in clock frequencies implies that bits can "disappear" during their transmission on an electrical cable. This is illustrated in the figure below.



A similar reasoning applies when the clock of the sending host is slower than the clock of the receiving host. In this case, the receiver will sense more bits than the bits that have been transmitted by the sender. This is illustrated in the figure below where the second bit received on the right was not transmitted by the left host.



From a Computer Science viewpoint, the physical transmission of information through a wire is often considered as a black box that allows to transmit bits. This black box is often referred to as the *physical layer service* and is represented by using the *DATA.request* and *DATA.indication* primitives introduced earlier. This physical layer service facilitates the sending and receiving of bits. This service abstracts the technological details that are involved in the actual transmission of the bits as an electromagnetic signal. However, it is important to remember that the *physical layer service* is imperfect and has the following characteristics :

- the *Physical layer service* may change, e.g. due to electromagnetic interferences, the value of a bit being transmitted
- the *Physical layer service* may deliver *more* bits to the receiver than the bits sent by the sender
- the *Physical layer service* may deliver *fewer* bits to the receiver than the bits sent by the sender

Many other types of encodings have been defined to transmit information over an electrical cable. All physical layers are able to send and receive physical symbols that represent values *0* and *1*. However, for various reasons that are outside the scope of this chapter, several physical layers exchange other physical symbols as well. For example, the Manchester encoding used in several physical layers can send four different symbols. The Manchester encoding is a differential encoding scheme in which time is divided into fixed-length periods. Each period is divided in two halves and two different voltage levels can be applied. To send a symbol, the sender must set one of these two voltage levels during each half period. To send a *1* (resp. *0*), the sender must set a high (resp. low) voltage during the first half of the period and a low (resp. high) voltage during the second half. This encoding ensures that there will be a transition at the middle of each period and allows the receiver to synchronise its clock to the sender's clock. Apart from the encodings for *0* and *1*, the Manchester encoding also supports two additional symbols : *InvH* and *InvB* where the same voltage level is used for the two half periods. By definition, these two symbols cannot appear inside a frame which is only composed of *0* and *1*. Some technologies use these special symbols as markers for the beginning or end of frames.

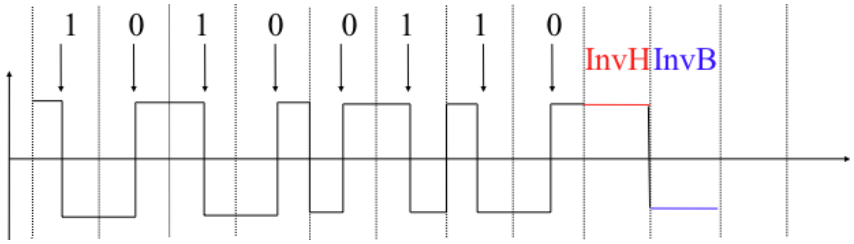


Figure 2.2: Manchester encoding

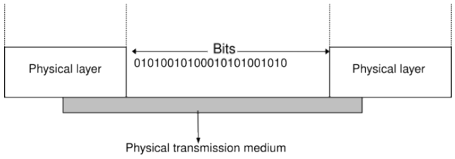


Figure 2.3: The Physical layer

All the functions related to the physical transmission or information through a wire (or a wireless link) are usually known as the *physical layer*. The physical layer allows thus two or more entities that are directly attached to the

same transmission medium to exchange bits. Being able to exchange bits is important as virtually any information can be encoded as a sequence of bits. Electrical engineers are used to processing streams of bits, but computer scientists usually prefer to deal with higher level concepts. A similar issue arises with file storage. Storage devices such as hard-disks also store streams of bits. There are hardware devices that process the bit stream produced by a hard-disk, but computer scientists have designed filesystems to allow applications to easily access such storage devices. These filesystems are typically divided into several layers as well. Hard-disks store sectors of 512 bytes or more. Unix filesystems group sectors in larger blocks that can contain data or *inodes* representing the structure of the filesystem. Finally, applications manipulate files and directories that are translated in blocks, sectors and eventually bits by the operating system.

Computer networks use a similar approach. Each layer provides a service that is built above the underlying layer and is closer to the needs of the applications. The datalink layer builds upon the service provided by the physical layer. We will see that it also contains several functions.

2.1.2 The datalink layer

Computer scientists are usually not interested in exchanging bits between two hosts. They prefer to write software that deals with larger blocks of data in order to transmit messages or complete files. Thanks to the physical layer service, it is possible to send a continuous stream of bits between two hosts. This stream of bits can include logical blocks of data, but we need to be able to extract each block of data from the bit stream despite the imperfections of the physical layer. In many networks, the basic unit of information exchanged between two directly connected hosts is often called a *frame*. A *frame* can be defined has a sequence of bits that has a particular syntax or structure. We will see examples of such frames later in this chapter.

To enable the transmission/reception of frames, the first problem to be solved is how to encode a frame as a sequence of bits, so that the receiver can easily recover the received frame despite the limitations of the physical layer.

If the physical layer were perfect, the problem would be very simple. We would simply need to define how to encode each frame as a sequence of consecutive bits. The receiver would then easily be able to extract the frames from the received bits. Unfortunately, the imperfections of the physical layer make this framing problem slightly more complex. Several solutions have been proposed and are used in practice in different network technologies.

Framing

The *framing* problem can be defined as : "*How does a sender encode frames so that the receiver can efficiently extract them from the stream of bits that it receives from the physical layer*".

A first solution to this problem is to require the physical layer to remain idle for some time after the transmission of each frame. These idle periods can be detected by the receiver and serve as a marker to delineate frame boundaries. Unfortunately, this solution is not acceptable for two reasons. First, some physical layers cannot remain idle and always need to transmit bits. Second, inserting an idle period between frames decreases the maximum bit rate that can be achieved.

Note: Bit rate and bandwidth

Bit rate and bandwidth are often used to characterize the transmission capacity of the physical service. The original definition of bandwidth, as listed in the Webster dictionary is *a range of radio frequencies which is occupied by a modulated carrier wave, which is assigned to a service, or over which a device can operate*. This definition corresponds to the characteristics of a given transmission medium or receiver. For example, the human ear is able to decode sounds in roughly the 0-20 KHz frequency range. By extension, bandwidth is also used to represent the capacity of a communication system in bits per second. For example, a Gigabit Ethernet link is theoretically capable of transporting one billion bits per second.

Given that multi-symbol encodings cannot be used by all physical layers, a generic solution which can be used with any physical layer that is able to transmit and receive only bits *0* and *1* is required. This generic solution is called *stuffing* and two variants exist : *bit stuffing* and *character stuffing*. To enable a receiver to easily delineate the frame boundaries, these two techniques reserve special bit strings as frame boundary markers and encode the frames so that these special bit strings do not appear inside the frames.

Bit stuffing reserves the *01111110* bit string as the frame boundary marker and ensures that there will never be six consecutive *1* symbols transmitted by the physical layer inside a frame. With bit stuffing, a frame is sent as follows. First, the sender transmits the marker, i.e. *01111110*. Then, it sends all the bits of the frame and inserts an additional bit set to *0* after each sequence of five consecutive *1* bits. This ensures that the sent frame never contains a sequence of six consecutive bits set to *1*. As a consequence, the marker pattern cannot appear inside the frame sent. The marker is also sent to mark the end of the frame. The receiver performs the opposite to decode a received frame. It first detects the beginning of the frame thanks to the *01111110* marker. Then, it processes the received bits and counts the number of consecutive bits set to *1*. If a *0* follows five consecutive bits set to *1*, this bit is removed since it was inserted by the sender. If a *1* follows five consecutive bits sets to *1*, it indicates a marker if it is followed by a bit set to *0*. The table below illustrates the application of bit stuffing to some frames.

Original frame	Transmitted frame
000100100100100100000011	01111110000100100100100100000101111110
011011111111111111110010	0111111001101111101111101111101100100111111 0
01111110	0111111001111101001111110

For example, consider the transmission of *011011111111111111110010*. The sender will first send the *01111110* marker followed by *011011111*. After these five consecutive bits set to *1*, it inserts a bit set to *0* followed by *11111*. A new *0* is inserted, followed by *11111*. A new *0* is inserted followed by the end of the frame *110010* and the *01111110* marker.

Bit stuffing increases the number of bits required to transmit each frame. The worst case for bit stuffing is of course a long sequence of bits set to *1* inside the frame. If transmission errors occur, stuffed bits or markers can be in error. In these cases, the frame affected by the error and possibly the next frame will not be correctly decoded by the receiver, but it will be able to resynchronize itself at the next valid marker.

Bit stuffing can be easily implemented in hardware. However, implementing it in software is difficult given the complexity of performing bit manipulations in software. Software implementations prefer to process characters than bits, software-based datalink layers usually use *character stuffing*. This technique operates on frames that contain an integer number of characters. In computer networks, characters are usually encoded by relying on the *ASCII* table. This table defines the encoding of various alphanumeric characters as a sequence of bits. **RFC 20** provides the ASCII table that is used by many protocols on the Internet. For example, the table defines the following binary representations :

- *A* : *1000011* b
- *0* : *0110000* b
- *z* : *1111010* b
- @ : *1000000* b
- *space* : *0100000* b

In addition, the *ASCII* table also defines several non-printable or control characters. These characters were designed to allow an application to control a printer or a terminal. These control characters include *CR* and *LF*, that are used to terminate a line, and the *BEL* character which causes the terminal to emit a sound.

- *NUL*: *0000000* b
- *BEL*: *0000111* b
- *CR* : *0001101* b
- *LF* : *0001010* b
- *DLE*: *0010000* b
- *STX*: *0000010* b
- *ETX*: *0000011* b

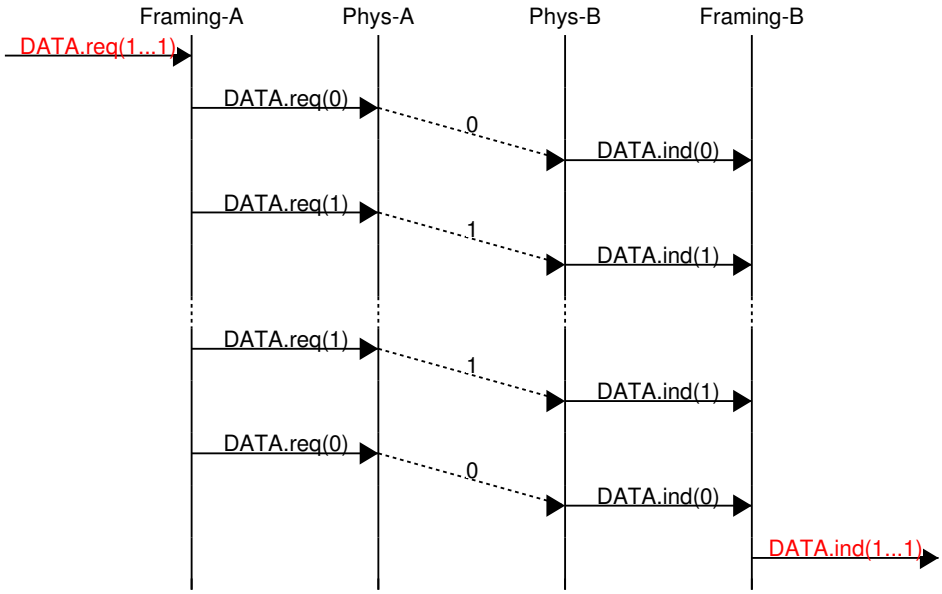
Some characters are used as markers to delineate the frame boundaries. Many *character stuffing* techniques use the *DLE*, *STX* and *ETX* characters of the ASCII character set. *DLE STX* (resp. *DLE ETX*) is used to mark the beginning (end) of a frame. When transmitting a frame, the sender adds a *DLE* character after each transmitted *DLE* character. This ensures that none of the markers can appear inside the transmitted frame. The receiver detects the frame boundaries and removes the second *DLE* when it receives two consecutive *DLE* characters. For

example, to transmit frame *1 2 3 DLE STX 4*, a sender will first send *DLE STX* as a marker, followed by *1 2 3 DLE*. Then, the sender transmits an additional *DLE* character followed by *STX 4* and the *DLE ETX* marker.

Original frame	Transmitted frame
1 2 3 4	*DLE STX* **1 2 3 4** *DLE ETX*
1 2 3 DLE STX 4	*DLE STX* **1 2 3** *DLE* **DLE STX** *4 DLE ETX*
DLE STX DLE ETX	*DLE STX DLE* **DLE STX** *DLE* **DLE ETX** *DLE ETX*

Character stuffing , like bit stuffing, increases the length of the transmitted frames. For *character stuffing*, the worst frame is a frame containing many *DLE* characters. When transmission errors occur, the receiver may incorrectly decode one or two frames (e.g. if the errors occur in the markers). However, it will be able to resynchronise itself with the next correctly received markers.

Bit stuffing and character stuffing allow to recover frames from a stream of bits or bytes. This framing mechanism provides a richer service than the physical layer. Through the framing service, one can send and receive complete frames. This framing service can also be represented by using the *DATA.request* and *DATA.indication* primitives. This is illustrated in the figure below, assuming hypothetical frames containing four useful bit and one bit of framing for graphical reasons.



We can now build upon the framing mechanism to allow the hosts to exchange frames containing an integer number of bits or bytes. Once the framing problem has been solved, we can focus our designing a technique that allows to reliably exchange frames.

Recovering from transmission errors

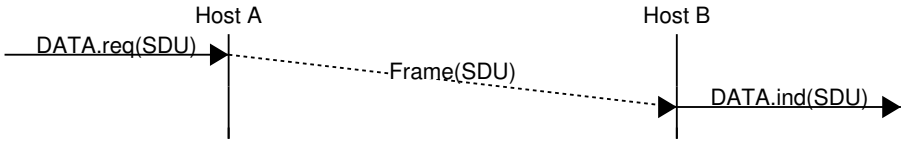
In this section, we develop a reliable datalink protocol running above the physical layer service. To design this protocol, we first assume that the physical layer provides a perfect service. We will then develop solutions to recover from the transmission errors.

The datalink layer is designed to send and receive frames on behalf of a user. We model these interactions by using the *DATA.req* and *DATA.ind* primitives. However, to simplify the presentation and to avoid confusion between a *DATA.req* primitive issued by the user of the datalink layer entity, and a *DATA.req* issued by the datalink layer entity itself, we will use the following terminology :

- the interactions between the user and the datalink layer entity are represented by using the classical *DATA.req* and the *DATA.ind* primitives
- the interactions between the datalink layer entity and the framing sublayer are represented by using *send* instead of *DATA.req* and *recvd* instead of *DATA.ind*

When running on top of a perfect framing sublayer, a datalink entity can simply issue a *send(SDU)* upon arrival of a *DATA.req(SDU)*. Similarly, the receiver issues a *DATA.ind(SDU)* upon receipt of a *recvd(SDU)*. Such a simple

protocol is sufficient when a single SDU is sent. This is illustrated in the figure below.



Unfortunately, this is not always sufficient to ensure a reliable delivery of the SDUs. Consider the case where a client sends tens of SDUs to a server. If the server is faster that the client, it will be able to receive and process all the segments sent by the client and deliver their content to its user. However, if the server is slower than the client, problems may arise. The datalink entity contains buffers to store SDUs that have been received as a *Data.request* but have not yet been sent. If the application is faster than the physical link, the buffer may become full. At this point, the operating system suspends the application to let the datalink entity empty its transmission queue. The datalink entity also uses a buffer to store the received frames that have not yet been processed by the application. If the application is slow to process the data, this buffer may overflow and the datalink entity will not able to accept any additional frame. The buffers of the datalink entity have a limited size and if they overflow, the arriving frames will be discarded, even if they are correct.

To solve this problem, a reliable protocol must include a feedback mechanism that allows the receiver to inform the sender that it has processed a frame and that another one can be sent. This feedback is required even though there are no transmission errors. To include such a feedback, our reliable protocol must process two types of frames :

- data frames carrying a SDU
- control frames carrying an acknowledgment indicating that the previous frames was processed correctly

These two types of frames can be distinguished by dividing the frame in two parts :

- the *header* that contains one bit set to *0* in data frames and set to *1* in control frames
- the payload that contains the SDU supplied by the application

The datalink entity can then be modelled as a finite state machine, containing two states for the receiver and two states for the sender. The figure below provides a graphical representation of this state machine with the sender above and the receiver below.

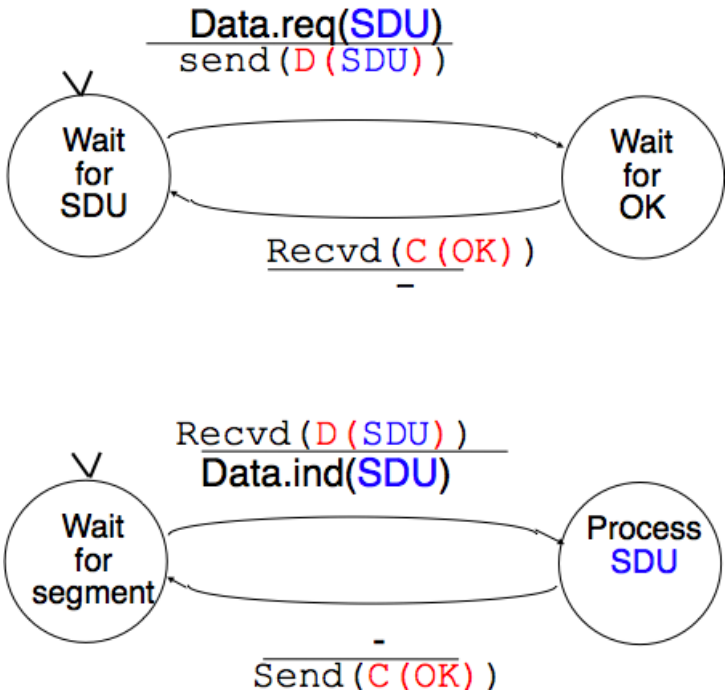
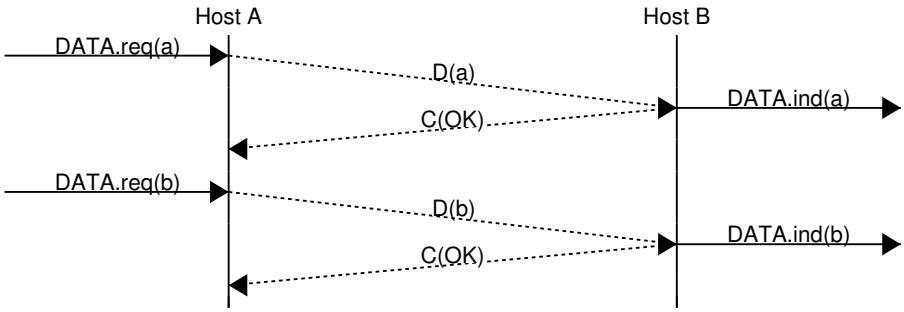


Figure 2.4: Finite state machine of the simplest reliable protocol

The above FSM shows that the sender has to wait for an acknowledgement from the receiver before being able to transmit the next SDU. The figure below illustrates the exchange of a few frames between two hosts.



Note: Services and protocols

An important aspect to understand before studying computer networks is the difference between a *service* and a *protocol*. In order to understand the difference between the two, it is useful to start with real world examples. The traditional Post provides a service where a postman delivers letters to recipients. The Post defines precisely which types of letters (size, weight, etc) can be delivered by using the Standard Mail service. Furthermore, the format of the envelope is specified (position of the sender and recipient addresses, position of the stamp). Someone who wants to send a letter must either place the letter at a Post Office or inside one of the dedicated mailboxes. The letter will then be collected and delivered to its final recipient. Note that for the regular service the Post usually does not guarantee the delivery of each particular letter, some letters may be lost, and some letters are delivered to the wrong mailbox. If a letter is important, then the sender can use the registered service to ensure that the letter will be delivered to its recipient. Some Post services also provide an acknowledged service or an express mail service that is faster than the regular service.

Reliable data transfer on top of an imperfect link

The datalink layer must deal with the transmission errors. In practice, we mainly have to deal with two types of errors in the datalink layer :

- Frames can be corrupted by transmission errors
- Frames can be lost or unexpected frames can appear

A first glance, loosing frames might seem strange on single link. However, if we take framing into account, transmission errors can affect the frame delineation mechanism and make the frame unreadable. For the same reason, a receiver could receive two (likely invalid) frames after a sender has transmitted a single frame.

To deal with these types of imperfections, reliable protocols rely on different types of mechanisms. The first problem is transmission errors. Data transmission on a physical link can be affected by the following errors :

- random isolated errors where the value of a single bit has been modified due to a transmission error
- random burst errors where the values of *n* consecutive bits have been changed due to transmission errors
- random bit creations and random bit removals where bits have been added or removed due to transmission errors

The only solution to protect against transmission errors is to add redundancy to the frames that are sent. *Information Theory* defines two mechanisms that can be used to transmit information over a transmission channel affected by random errors. These two mechanisms add redundancy to the transmitted information, to allow the receiver to detect or sometimes even correct transmission errors. A detailed discussion of these mechanisms is outside the scope of this chapter, but it is useful to consider a simple mechanism to understand its operation and its limitations.

Besides framing, datalink layers also include mechanisms to detect and sometimes even recover from transmission errors. To allow a receiver to detect transmission errors, a sender must add some redundant information as an *error detection* code to the frame sent. This *error detection* code is computed by the sender on the frame that it transmits. When the receiver receives a frame with an error detection code, it recomputes it and verifies whether the received *error detection code* matches the computer *error detection code*. If they match, the frame is considered to be valid.

Many error detection schemes exist and entire books have been written on the subject. A detailed discussion of these techniques is outside the scope of this book, and we will only discuss some examples to illustrate the key principles.

To understand *error detection codes*, let us consider two devices that exchange bit strings containing *N* bits. To allow the receiver to detect a transmission error, the sender converts each string of *N* bits into a string of *N+r* bits. Usually, the *r* redundant bits are added at the beginning or the end of the transmitted bit string, but some techniques interleave redundant bits with the original bits. An *error detection code* can be defined as a function that computes the *r* redundant bits corresponding to each string of *N* bits. The simplest error detection code is the parity bit. There are two types of parity schemes : even and odd parity. With the *even* (resp. *odd*) parity scheme, the redundant bit is chosen so that an even (resp. odd) number of bits are set to *1* in the transmitted bit string of *N+r* bits. The receiver can easily recompute the parity of each received bit string and discard the strings with an invalid parity. The parity scheme is often used when 7-bit characters are exchanged. In this case, the eighth bit is often a parity bit. The table below shows the parity bits that are computed for bit strings containing three bits.

3 bits string	Odd parity	Even parity
000	1	0
001	0	1
010	0	1
100	0	1
111	0	1
110	1	0
101	1	0
011	1	0

The parity bit allows a receiver to detect transmission errors that have affected a single bit among the transmitted *N+r* bits. If there are two or more bits in error, the receiver may not necessarily be able to detect the transmission error. More powerful error detection schemes have been defined. The Cyclical Redundancy Checks (CRC) are widely used in datalink layer protocols. An N-bits CRC can detect all transmission errors affecting a burst of less than N bits in the transmitted frame and all transmission errors that affect an odd number of bits. Additional details about CRCs may be found in [Williams1993].

It is also possible to design a code that allows the receiver to correct transmission errors. The simplest *error correction code* is the triple modular redundancy (TMR). To transmit a bit set to *1* (resp. *0*), the sender transmits *111* (resp. *000*). When there are no transmission errors, the receiver can decode *111* as *1*. If transmission errors have affected a single bit, the receiver performs majority voting as shown in the table below. This scheme allows the receiver to correct all transmission errors that affect a single bit.

Received bits	Decoded bit
000	0
001	0
010	0
100	0
111	1
110	1
101	1
011	1

Other more powerful error correction codes have been proposed and are used in some applications. The Hamming Code is a clever combination of parity bits that provides error detection and correction capabilities.

Reliable protocols use error detection schemes, but none of the widely used reliable protocols rely on error correction schemes. To detect errors, a frame is usually divided into two parts :

- a *header* that contains the fields used by the reliable protocol to ensure reliable delivery. The header contains a checksum or Cyclical Redundancy Check (CRC) [Williams1993] that is used to detect transmission errors
- a *payload* that contains the user data

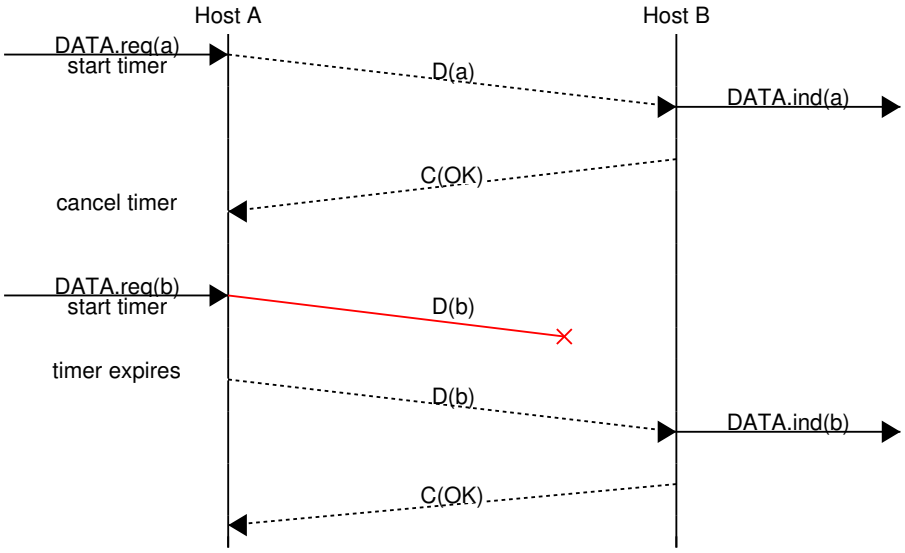
Some headers also include a *length* field, which indicates the total length of the frame or the length of the payload.

The simplest error detection scheme is the checksum. A checksum is basically an arithmetic sum of all the bytes that a frame is composed of. There are different types of checksums. For example, an eight bit checksum can be computed as the arithmetic sum of all the bytes of (both the header and trailer of) the frame. The checksum is

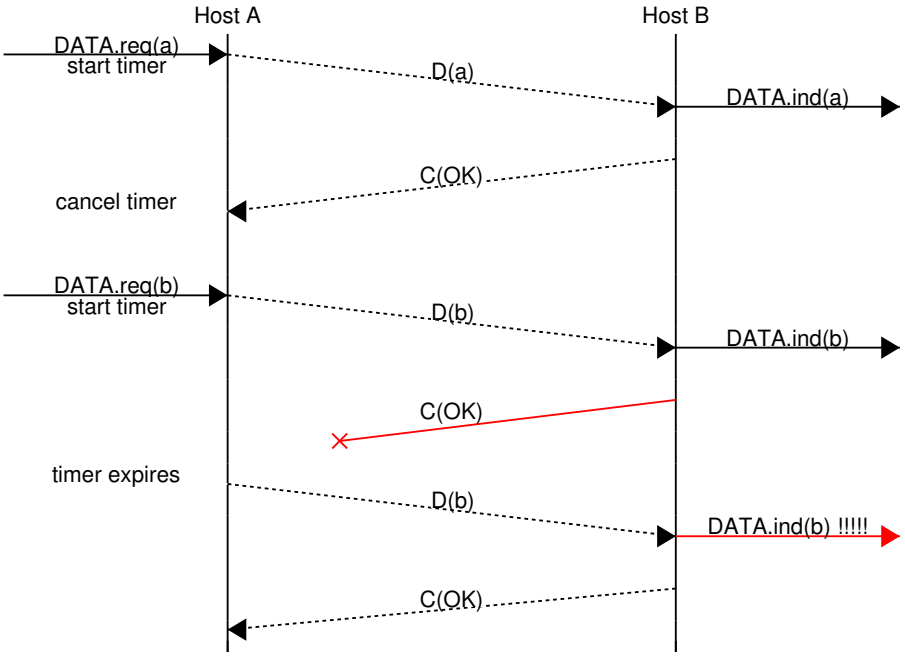
computed by the sender before sending the frame and the receiver verifies the checksum upon frame reception. The receiver discards frames received with an invalid checksum. Checksums can be easily implemented in software, but their error detection capabilities are limited. Cyclical Redundancy Checks (CRC) have better error detection capabilities [SGP98], but require more CPU when implemented in software.

Note: Checksums, CRCs, ...

Most of the protocols in the TCP/IP protocol suite rely on the simple Internet checksum in order to verify that a received packet has not been affected by transmission errors. Despite its popularity and ease of implementation, the Internet checksum is not the only available checksum mechanism. Cyclical Redundancy Checks (CRC) are very powerful error detection schemes that are used notably on disks, by many datalink layer protocols and file formats such as zip or png. They can easily be implemented efficiently in hardware and have better error-detection capabilities than the Internet checksum [SGP98] . However, CRCs are sometimes considered to be too CPU-intensive for software implementations and other checksum mechanisms are preferred. The TCP/IP community chose the Internet checksum, the OSI community chose the Fletcher checksum [Sklower89] . Nowadays there are efficient techniques to quickly compute CRCs in software [Feldmeier95]



Unfortunately, retransmission timers alone are not sufficient to recover from segment losses. Let us consider, as an example, the situation depicted below where an acknowledgement is lost. In this case, the sender retransmits the data segment that has not been acknowledged. Unfortunately, as illustrated in the figure below, the receiver considers the retransmission as a new segment whose payload must be delivered to its



user.

To solve this problem, datalink protocols associate a *sequence number* to each data frame. This *sequence number* is one of the fields found in the header of data frames. We use the notation *D(x,...)* to indicate a data frame whose sequence number field is set to value *x*. The acknowledgements also contain a sequence number indicating the data frames that it is acknowledging. We use *OKx* to indicate an acknowledgement frame that confirms the reception of *D(x,...)*. The sequence number is encoded as a bit string of fixed length. The simplest reliable protocol is the Alternating Bit Protocol (ABP).

The Alternating Bit Protocol uses a single bit to encode the sequence number. It can be implemented easily. The sender and the receiver only require a four-state Finite State Machine.

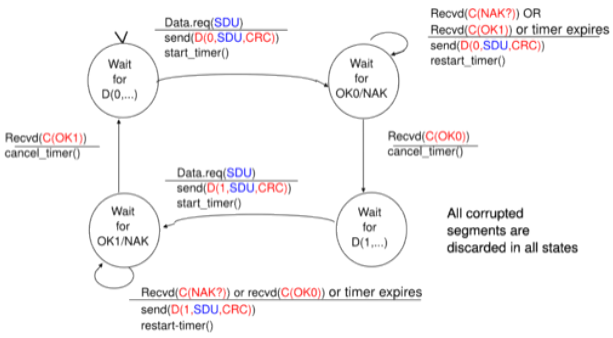


Figure 2.5: Alternating bit protocol : Sender FSM

The initial state of the sender is *Wait for D(0,...)*. In this state, the sender waits for a *Data.request*. The first data frame that it sends uses sequence number *0*. After having sent this frame, the sender waits for an *OK0* acknowledgement. A frame is retransmitted upon expiration of the retransmission timer or if an acknowledgement with an incorrect sequence number has been received.

The receiver first waits for *D(0,...)*. If the frame contains a correct *CRC*, it passes the SDU to its user and sends *OK0*. If the frame contains an invalid CRC, it is immediately discarded. Then, the receiver waits for *D(1,...)*. In this state, it may receive a duplicate *D(0,...)* or a data frame with an invalid CRC. In both cases, it returns an *OK0* frame to allow the sender to recover from the possible loss of the previous *OK0* frame.

Note: Dealing with corrupted frames

The receiver FSM of the Alternating bit protocol discards all frames that contain an invalid CRC. This is the safest approach since the received frame can be completely different from the frame sent by the remote host. A receiver should not attempt at extracting information from a corrupted frame because it cannot know which portion of the frame has been affected by the error.

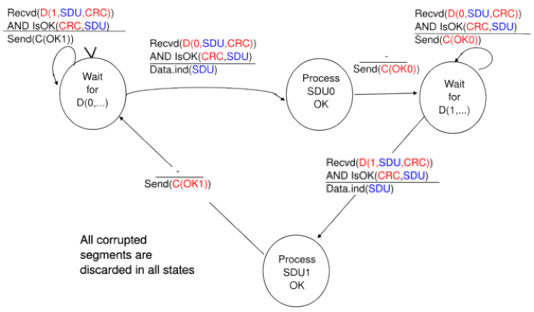
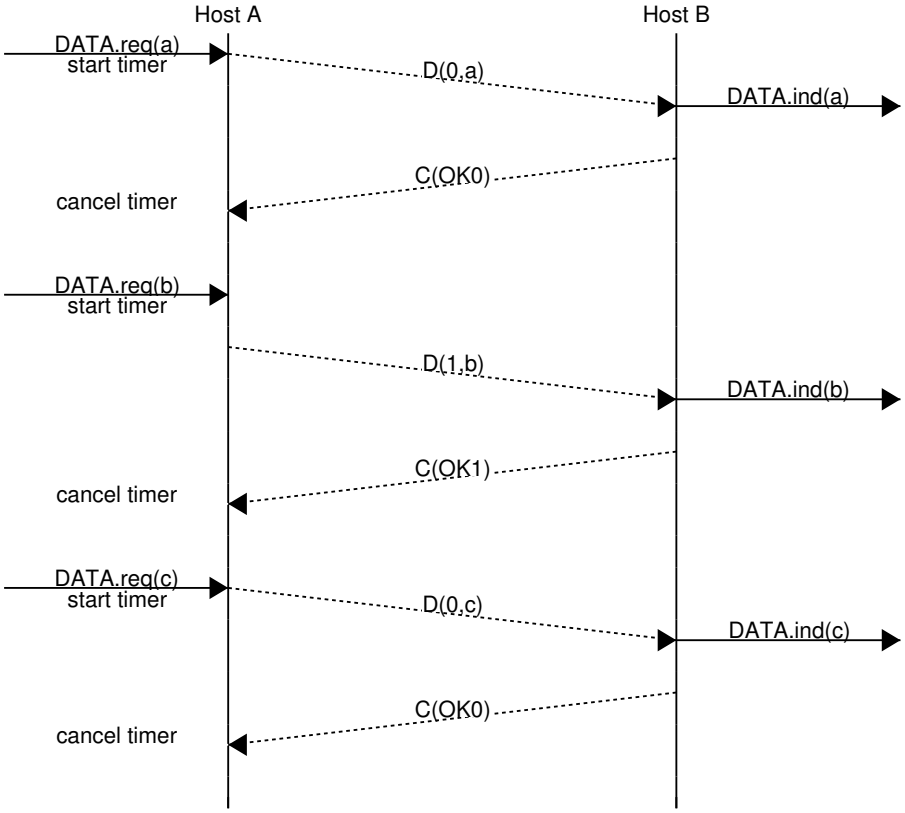
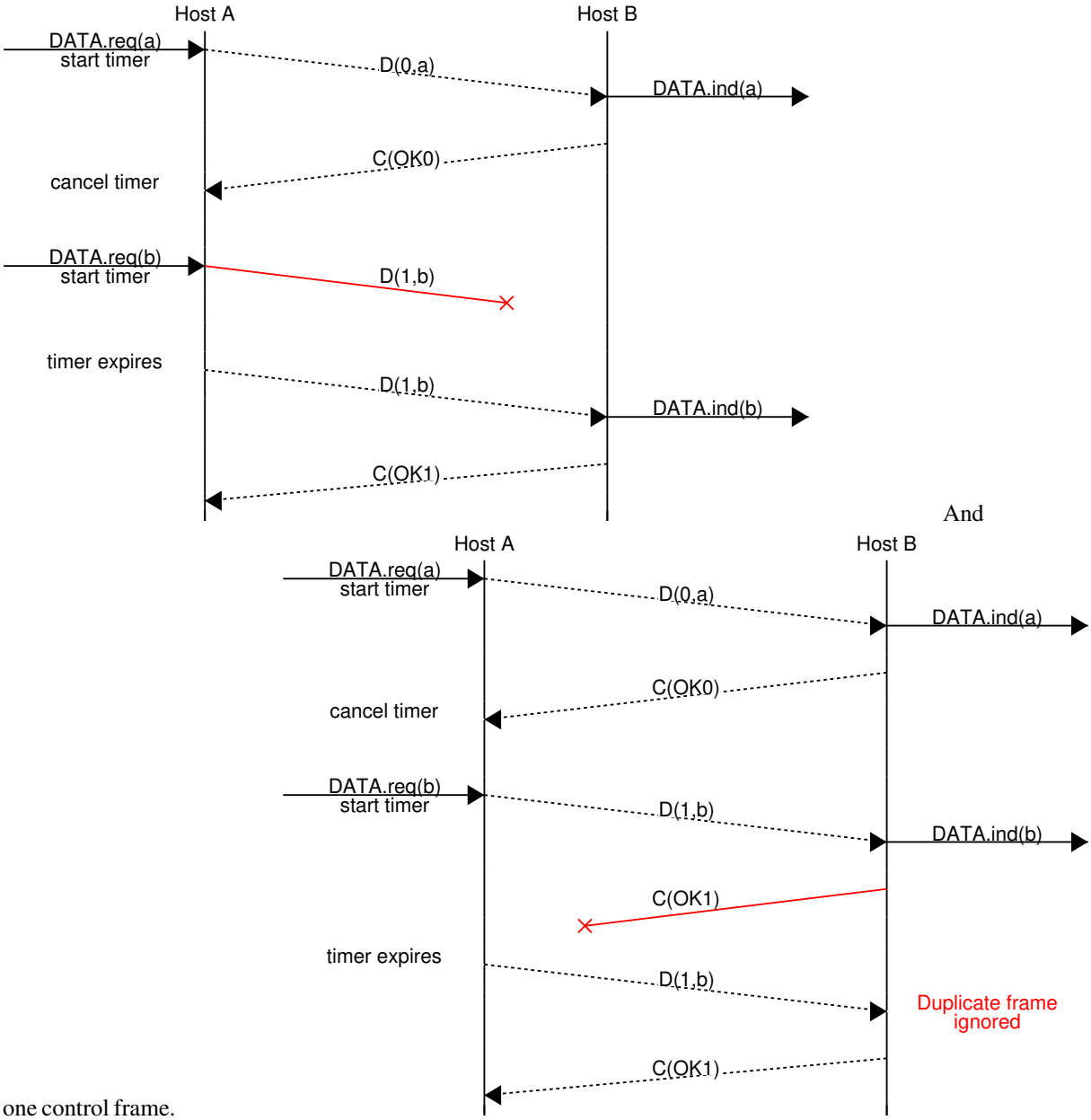


Figure 2.6: Alternating bit protocol : Receiver FSM

The figure below illustrates the operation of the alternating bit protocol.



The Alternating Bit Protocol can recover from the losses of data or control frames. This is illustrated in the two figures below. The first figure shows the loss of one data segment.



the loss of one control frame.

The Alternating Bit Protocol can recover from transmission errors and frame losses. However, it has one important drawback. Consider two hosts that are directly connected by a 50 Kbits/sec satellite link that has a 250 milliseconds propagation delay. If these hosts send 1000 bits frames, then the maximum throughput that can be achieved by the alternating bit protocol is one frame every $20 + 250 + 250 = 520$ milliseconds if we ignore the transmission time of the acknowledgement. This is less than 2 Kbits/sec !

Go-back-n and selective repeat

To overcome the performance limitations of the alternating bit protocol, reliable protocols rely on *pipelining*. This technique allows a sender to transmit several consecutive frames without being forced to wait for an acknowledgement after each frame. Each data frame contains a sequence number encoded in an n bits field.

Pipelining allows the sender to transmit frames at a higher rate. However this higher transmission rate may overload the receiver. In this case, the frames sent by the sender will not be correctly received by their final destination. The reliable protocols that rely on pipelining allow the sender to transmit W unacknowledged frames before being forced to wait for an acknowledgement from the receiving entity.

This is implemented by using a *sliding window*. The sliding window is the set of consecutive sequence numbers that the sender can use when transmitting frames without being forced to wait for an acknowledgement. The figure

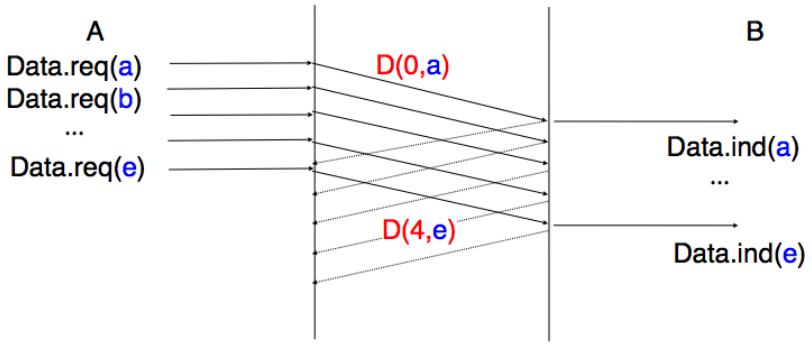


Figure 2.7: Pipelining improves the performance of reliable protocols

below shows a sliding window containing five segments (*6,7,8,9* and *10*). Two of these sequence numbers (*6* and *7*) have been used to send frames and only three sequence numbers (*8*, *9* and *10*) remain in the sliding window. The sliding window is said to be closed once all sequence numbers contained in the sliding window have been used.

The figure below illustrates the operation of the sliding window. It uses a sliding window of three frames. The sender can thus transmit three frames before being forced to wait for an acknowledgement. The sliding window moves to the higher sequence numbers upon the reception of each acknowledgement. When the first acknowledgement (*OK0*) is received, it allows the sender to move its sliding window to the right and sequence number *3* becomes available. This sequence number is used later to transmit the frame containing *d*.

In practice, as the frame header includes an *n* bits field to encode the sequence number, only the sequence numbers between 0 and $2^n - 1$ can be used. This implies that, during a long transfer, the same sequence number will be used for different frames and the sliding window will wrap. This is illustrated in the figure below assuming that *2* bits are used to encode the sequence number in the frame header. Note that upon reception of *OK1*, the sender slides its window and can use sequence number *0* again.

Unfortunately, frame losses do not disappear because a reliable protocol uses a sliding window. To recover from losses, a sliding window protocol must define :

- a heuristic to detect frame losses
- a *retransmission strategy* to retransmit the lost frames

The simplest sliding window protocol uses the *go-back-n* recovery. Intuitively, *go-back-n* operates as follows. A *go-back-n* receiver is as simple as possible. It only accepts the frames that arrive in-sequence. A *go-back-n* receiver discards any out-of-sequence frame that it receives. When *go-back-n* receives a data frame, it always returns an acknowledgement containing the sequence number of the last in-sequence frame that it has received. This acknowledgement is said to be *cumulative*. When a *go-back-n* receiver sends an acknowledgement for sequence number *x*, it implicitly acknowledges the reception of all frames whose sequence number is earlier than *x*. A key advantage of these cumulative acknowledgements is that it is easy to recover from the loss of an acknowledgement. Consider for example a *go-back-n* receiver that received frames *1*, *2* and *3*. It sent *OK1*, *OK2* and *OK3*. Unfortunately, *OK1* and *OK2* were lost. Thanks to the cumulative acknowledgements, when the receiver receives *OK3*, it knows that all three frames have been correctly received.

The figure below shows the FSM of a simple *go-back-n* receiver. This receiver uses two variables : *lastack* and *next*. *next* is the next expected sequence number and *lastack* the sequence number of the last data frame that has

Figure 2.8: The sliding window

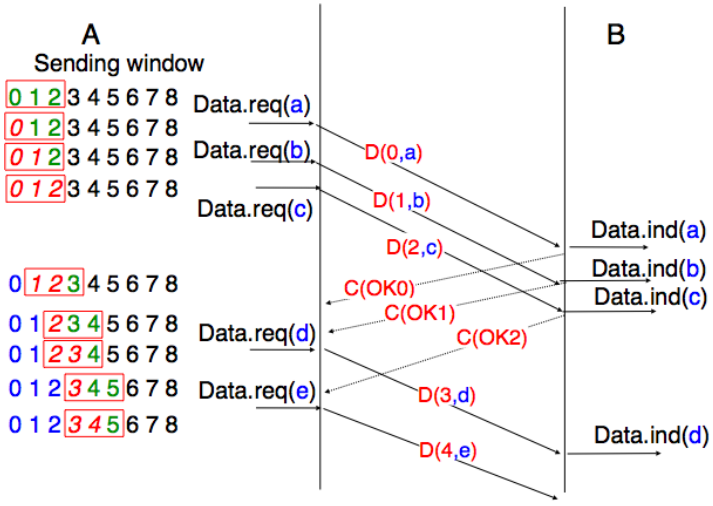


Figure 2.9: Sliding window example

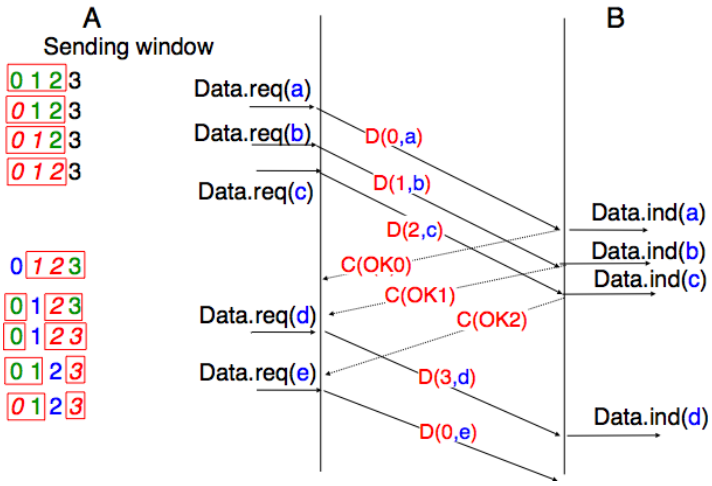


Figure 2.10: Utilisation of the sliding window with modulo arithmetic

been acknowledged. The receiver only accepts the frame that are received in sequence. *maxseq* is the number of different sequence numbers (2^n).

A *go-back-n* sender is also very simple. It uses a sending buffer that can store an entire sliding window of frames [1] . The frames are sent with increasing sequence numbers (modulo *maxseq*). The sender must wait for an acknowledgement once its sending buffer is full. When a *go-back-n* sender receives an acknowledgement, it removes from the sending buffer all the acknowledged frames and uses a retransmission timer to detect frame losses. A simple *go-back-n* sender maintains one retransmission timer per connection. This timer is started when the first frame is sent. When the *go-back-n sender* receives an acknowledgement, it restarts the retransmission timer only if there are still unacknowledged frames in its sending buffer. When the retransmission timer expires, the *go-back-n* sender assumes that all the unacknowledged frames currently stored in its sending buffer have been lost. It thus retransmits all the unacknowledged frames in the buffer and restarts its retransmission timer.

The operation of *go-back-n* is illustrated in the figure below. In this figure, note that upon reception of the out-of-sequence frame *D(2,c)*, the receiver returns a cumulative acknowledgement *C(OK,0)* that acknowledges all the frames that have been received in sequence. The lost frame is retransmitted upon the expiration of the retransmission timer.

The main advantage of *go-back-n* is that it can be easily implemented, and it can also provide good performance when only a few frames are lost. However, when there are many losses, the performance of *go-back-n* quickly drops for two reasons :

- the *go-back-n* receiver does not accept out-of-sequence frames
- the *go-back-n* sender retransmits all unacknowledged frames once it has detected a loss

Selective repeat is a better strategy to recover from losses. Intuitively, *selective repeat* allows the receiver to accept out-of-sequence frames. Furthermore, when a *selective repeat* sender detects losses, it only retransmits the frames that have been lost and not the frames that have already been correctly received.

A *selective repeat* receiver maintains a sliding window of *W* frames and stores in a buffer the out-of-sequence frames that it receives. The figure below shows a five-frame receive window on a receiver that has already received frames *7* and *9*.

[1] The size of the sliding window can be either fixed for a given protocol or negotiated during the connection establishment phase. Some protocols allow to change the maximum window size during the data transfert. We will see explain with real protocols later.

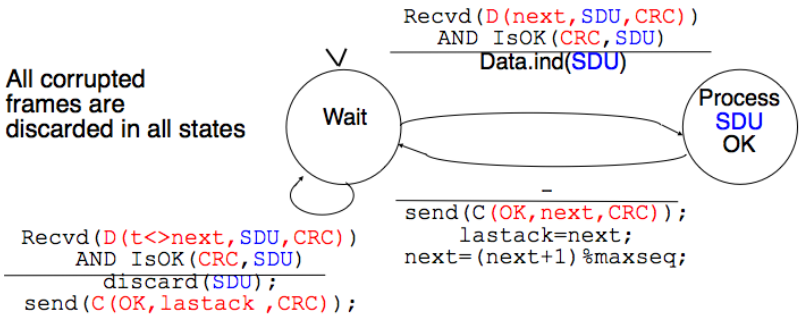


Figure 2.11: Go-back-n : receiver FSM

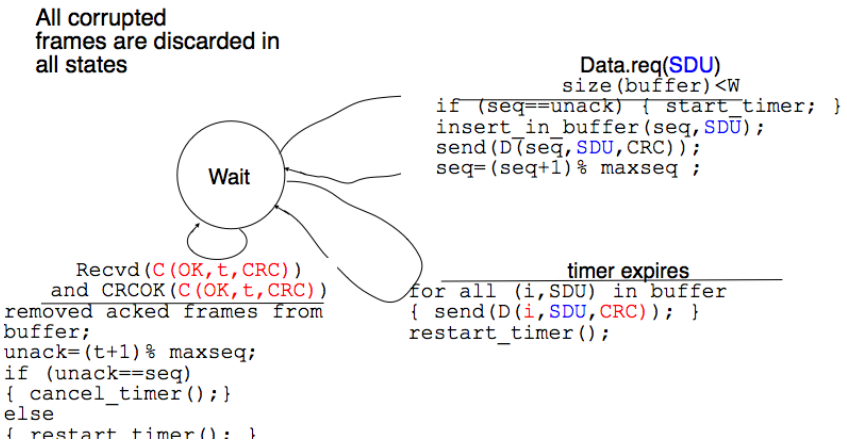


Figure 2.12: Go-back-n : sender FSM

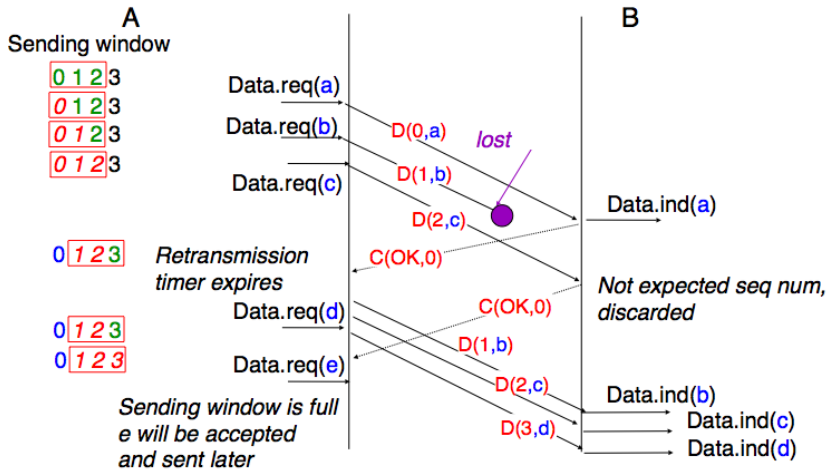


Figure 2.13: Go-back-n : example

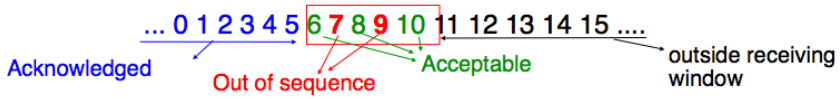


Figure 2.14: The receiving window with selective repeat

A *selective repeat* receiver discards all frames having an invalid CRC, and maintains the variable *lastack* as the sequence number of the last in-sequence frame that it has received. The receiver always includes the value of *lastack* in the acknowledgements that it sends. Some protocols also allow the *selective repeat* receiver to acknowledge the out-of-sequence frames that it has received. This can be done for example by placing the list of the correctly received, but out-of-sequence frames in the acknowledgements together with the *lastack* value.

When a *selective repeat* receiver receives a data frame, it first verifies whether the frame is inside its receiving window. If yes, the frame is placed in the receive buffer. If not, the received frame is discarded and an acknowledgement containing *lastack* is sent to the sender. The receiver then removes all consecutive frames starting at *lastack* (if any) from the receive buffer. The payloads of these frames are delivered to the user, *lastack* and the receiving window are updated, and an acknowledgement acknowledging the last frame received in sequence is sent.

The *selective repeat* sender maintains a sending buffer that can store up to *W* unacknowledged frames. These frames are sent as long as the sending buffer is not full. Several implementations of a *selective repeat* sender are possible. A simple implementation associates one retransmission timer to each frame. The timer is started when the frame is sent and cancelled upon reception of an acknowledgement that covers this frame. When a retransmission timer expires, the corresponding frame is retransmitted and this retransmission timer is restarted. When an acknowledgement is received, all the frames that are covered by this acknowledgement are removed from the sending buffer and the sliding window is updated.

The figure below illustrates the operation of *selective repeat* when frames are lost. In this figure, *C(OK,x)* is used to indicate that all frames, up to and including sequence number *x* have been received correctly.

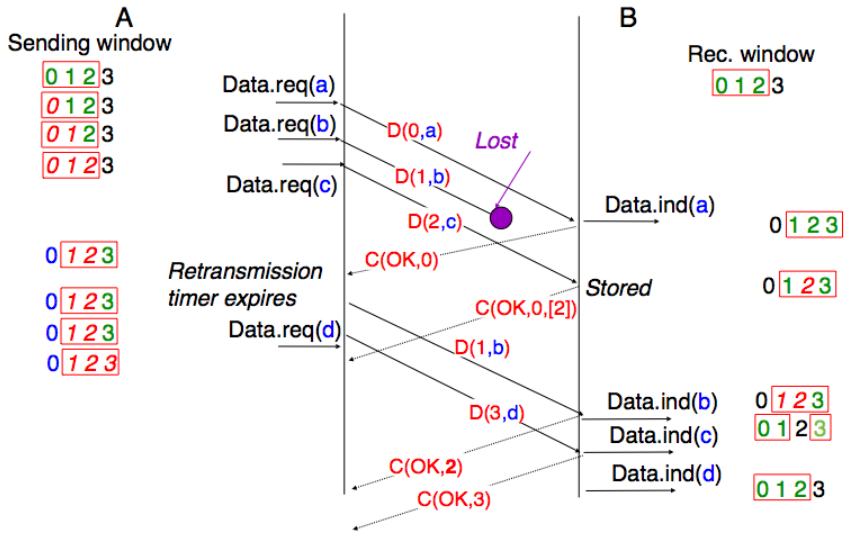


Figure 2.15: Selective repeat : example

Pure cumulative acknowledgements work well with the *go-back-n* strategy. However, with only cumulative acknowledgements a *selective repeat* sender cannot easily determine which frames have been correctly received after a data frame has been lost. For example, in the figure above, the second *C(OK,0)* does not inform explicitly the sender of the reception of *D(2,c)* and the sender could retransmit this frame although it has already been received. A possible solution to improve the performance of *selective repeat* is to provide additional information about the received frames in the acknowledgements that are returned by the receiver. For example, the receiver could add in the returned acknowledgement the list of the sequence numbers of all frames that have already been received. Such acknowledgements are sometimes called *selective acknowledgements*. This is illustrated in the figure above.

In the figure above, when the sender receives *C(OK,0,[2])*, it knows that all frames up to and including *D(0,...)* have been correctly received. It also knows that frame *D(2,...)* has been received and can cancel the retransmission

timer associated to this frame. However, this frame should not be removed from the sending buffer before the reception of a cumulative acknowledgement (*C(OK,2)* in the figure above) that covers this frame.

Note: Maximum window size with *go-back-n* and *selective repeat*

A reliable protocol that uses *n* bits to encode its sequence number can send up to 2^n successive frames. However, to ensure a reliable delivery of the frames, *go-back-n* and *selective repeat* cannot use a sending window of 2^n frames. Consider first *go-back-n* and assume that a sender sends 2^n frames. These frames are received in-sequence by the destination, but all the returned acknowledgements are lost. The sender will retransmit all frames. These frames will all be accepted by the receiver and delivered a second time to the user. It is easy to see that this problem can be avoided if the maximum size of the sending window is $2^n - 1$ frames. A similar problem occurs with *selective repeat*. However, as the receiver accepts out-of-sequence frames, a sending window of $2^n - 1$ frames is not sufficient to ensure a reliable delivery. It can be easily shown that to avoid this problem, a *selective repeat* sender cannot use a window that is larger than $\frac{2^n}{2}$ frames.

Reliable protocols often need to send data in both directions. To reduce the overhead caused by the acknowledgements, most reliable protocols use *piggybacking*. Thanks to this technique, a datalink entity can place the acknowledgements and the receive window that it advertises for the opposite direction of the data flow inside the header of the data frames that it sends. The main advantage of piggybacking is that it reduces the overhead as it is not necessary to send a complete frame to carry an acknowledgement. This is illustrated in the figure below where the acknowledgement number is underlined in the data frames. Piggybacking is only used when data flows in both directions. A receiver will generate a pure acknowledgement when it does not send data in the opposite direction as shown in the bottom of the figure.

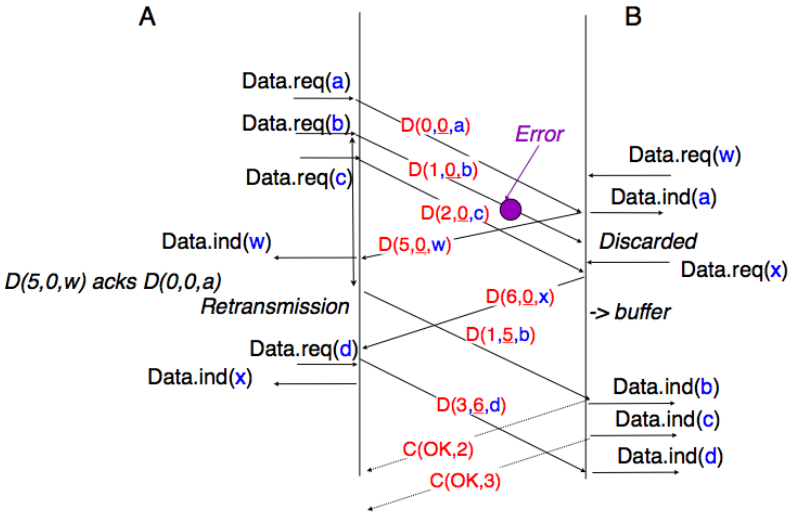


Figure 2.16: Piggybacking example

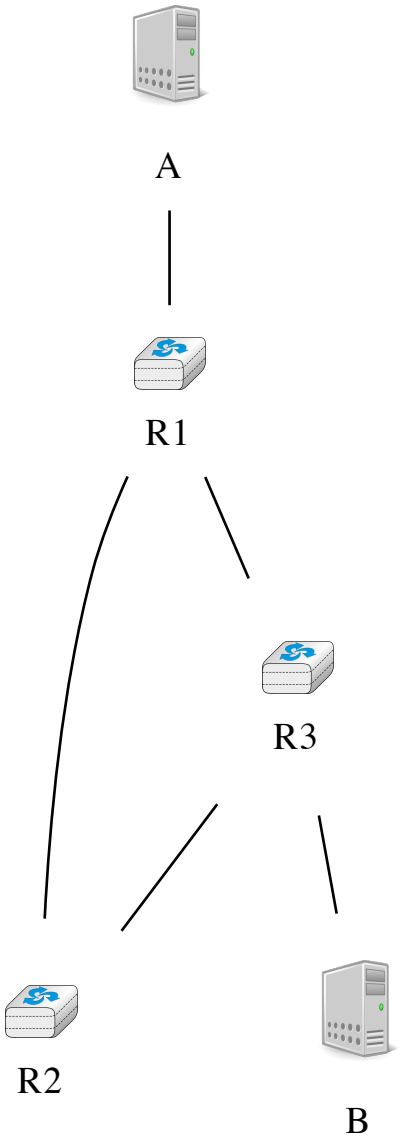
2.2 Building a network

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=2

In the previous section, we have explained how reliable protocols allow hosts to exchange data reliably even if the underlying physical layer is imperfect and thus unreliable. Connecting two hosts together through a wire is the first step to build a network. However, this is not sufficient. Hosts usually need to interact with remote hosts that

are not directly connected through a direct physical layer link. This can be achieved by adding one layer above the datalink layer : the *network* layer.

The main objective of the network layer is to allow endsystems, connected to different networks, to exchange information through intermediate systems called *router*. The unit of information in the network layer is called a *packet*.



Before explaining the network layer in detail, it is useful to remember the characteristics of the service provided by the *datalink* layer. There are many variants of the datalink layer. Some provide a reliable service while others do not provide any guarantee of delivery. The reliable datalink layer services are popular in environments such as wireless networks were transmission errors are frequent. On the other hand, unreliable services are usually used when the physical layer provides an almost reliable service (i.e. only a negligible fraction of the frames are affected by transmission errors). Such *almost reliable* services are frequently in wired and optical networks. In this chapter, we will assume that the datalink layer service provides an *almost reliable* service since this is both

the most general one and also the most widely deployed one.

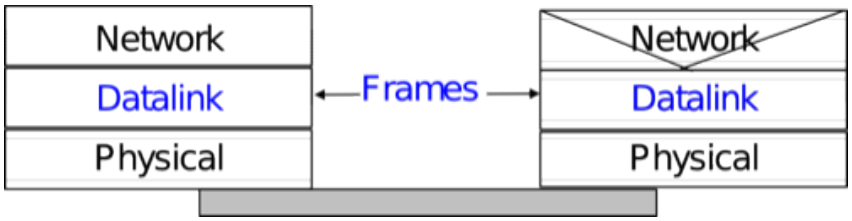


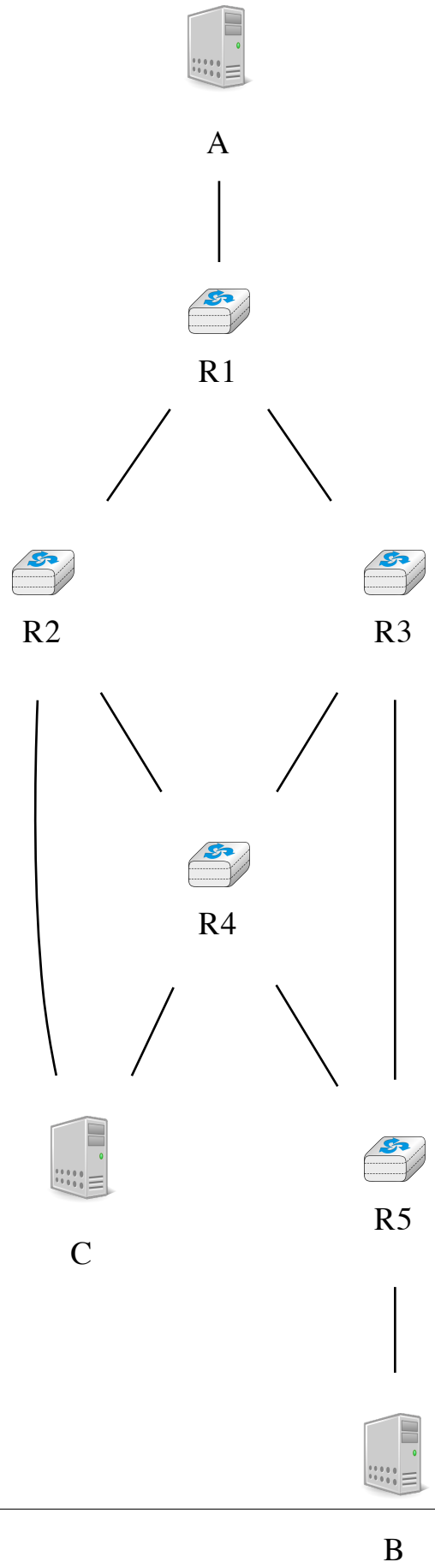
Figure 2.17: The point-to-point datalink layer

There are three main types of datalink layers. The simplest datalink layer is when there are only two communicating systems that are directly connected through the physical layer. Such a datalink layer is used when there is a point-to-point link between the two communicating systems. The two systems can be endsystems or routers. PPP (Point-to-Point Protocol), defined in **RFC 1661**, is an example of such a point-to-point datalink layer. Datalink layers exchange *frames* and a datalink *frame* sent by a datalink layer entity on the left is transmitted through the physical layer, so that it can reach the datalink layer entity on the right. Point-to-point datalink layers can either provide an unreliable service (frames can be corrupted or lost) or a reliable service (in this case, the datalink layer includes retransmission mechanisms). .. The unreliable service is frequently used above physical layers (e.g. optical fiber, twisted pairs) having a low bit error ratio while reliability mechanisms are often used in wireless networks to recover locally from transmission errors.

The second type of datalink layer is the one used in Local Area Networks (LAN). Conceptually, a LAN is a set of communicating devices such that any two devices can directly exchange frames through the datalink layer. Both endsystems and routers can be connected to a LAN. Some LANs only connect a few devices, but there are LANs that can connect hundreds or even thousands of devices. In this chapter, we focus on the utilization of point-to-point datalink layers. We will describe later the organisation and the operation of Local Area Networks and their impact on the network layer.

Even if we only consider the point-to-point datalink layers, there is an important characteristics of these layers that we cannot ignore. No datalink layer is able to send frames of unlimited size. Each datalink layer is characterized by a maximum frame size. There are more than a dozen different datalink layers and unfortunately most of them use a different maximum frame size. This heterogeneity in the maximum frame sizes will cause problems when we will need to exchange data between hosts attached to different types of datalink layers.

As a first step, let us assume that we only need to exchange small amount of data. In this case, there is no issue with the maximum length of the frames. However, there are other more interesting problems that we need to tackle. To understand these problems, let us consider the network represented in the figure below.


A
R1
R2
R3
R4
R5
C
B

This network contains two types of devices. The end hosts, represented as a small workstation and the routers, represented as boxes with three arrows. An endhost is a device which is able to send and receive data for its own usage in contrast with routers that most of the time forward data towards their final destination. Routers have multiple links to neighboring routers or endhosts. Endhosts are usually attached via a single link to the network. Nowadays, with the growth of wireless networks, more and more endhosts are equipped with several physical interfaces. These endhosts are often called *multihomed*. Still, using several interfaces at the same time often leads to practical issues that are beyond the scope of this document. For this reason, we will only consider *single-homed* hosts in this ebook.

To understand the key principles behind the operation of a network, let us analyse all the operations that need to be performed to allow host *A* in the above network to send one byte to host *B*. Thanks to the datalink layer used above the *A-R1* link, host *A* can easily send a byte to router *R1* inside a frame. However, upon reception of this frame, router *R1* needs to understand that the byte is destined to host *B* and not to itself. This is the objective of the network layer.

The network layer enables the transmission of information between hosts that are not directly connected through intermediate routers. This transmission is carried out by putting the information to be transmitted inside a data structure which is called a *packet*. Like a frame that contains useful data and control information, a packet also contains useful data and control information. An important issue in the network layer is the ability to identify a node (host or router) inside the network. This identification is performed by associating an address to each node. An *address* is usually represented as a sequence of bits. Most networks use fixed-length addresses. At this stage, let us simply assume that each of the nodes in the above network has an address which corresponds to the binary representation on its name on the figure.

To send one byte of information to host *B*, host *A* needs to place this information inside a *packet*. In addition to the data being transmitted, the packet must also contain either the addresses of the source and the destination nodes or information that indicates the path that needs to be followed to reach the destination.

There are two possible organisations for the network layer :

- *datagram*
- *virtual circuits*

2.2.1 The datagram organisation

The first and most popular organisation of the network layer is the datagram organisation. This organisation is inspired by the organisation of the postal service. Each host is identified by a *network layer address*. To send information to a remote host, a host creates a packet that contains :

- the network layer address of the destination host
- its own network layer address
- the information to be sent

To understand the datagram organisation, let us consider the figure below. A network layer address, represented by a letter, has been assigned to each host and router. To send some information to host *J*, host *A* creates a packet containing its own address, the destination address and the information to be exchanged.

With the datagram organisation, routers use *hop-by-hop forwarding*. This means that when a router receives a packet that is not destined to itself, it looks up the destination address of the packet in its *forwarding table*. A *forwarding table* is a data structure that maps each destination address (or set of destination addresses) to the outgoing interface over which a packet destined to this address must be forwarded to reach its final destination. The router consults its forwarding table for each packet that it handles.

The figure illustrates some possible forwarding tables in this network. By inspecting the forwarding tables of the different routers, one can find the path followed by packets sent from a source to a particular destination. In the example above, host *A* sends its packet to router *R1*. *R1* consults its routing table and forwards the packet towards *R2*. Based on its own routing table, *R2* decides to forward the packet to *R5* that can deliver it to its destination. Thus, the path from *A* to *J* is *A -> R1 -> R2 -> R5 -> J*.

The computation of the forwarding tables of all the routers inside a network is a key element for the correct operation of the network. This computation can be carried out in different ways and it is possible to use both

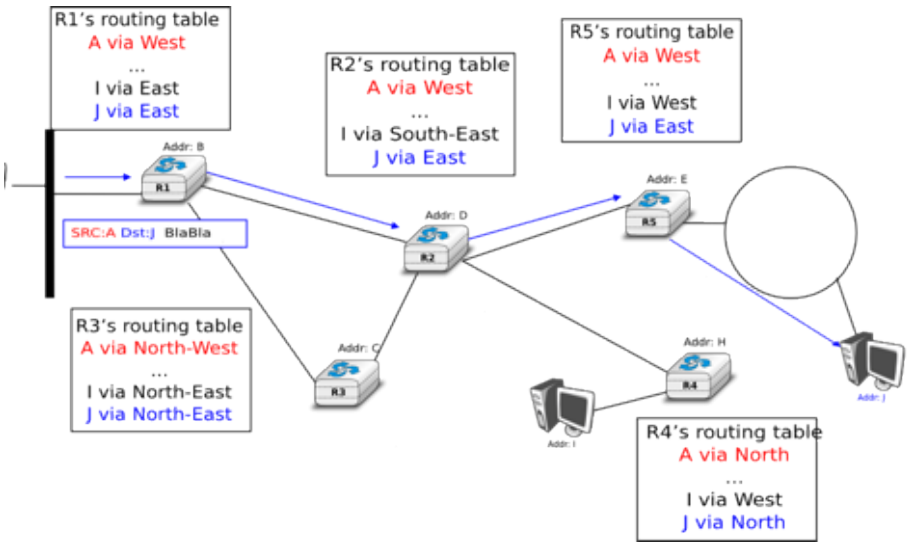


Figure 2.18: A simple internetwork

distributed and centralized algorithms. These algorithms provide different performance, may lead to different types of paths, but their composition must lead to valid path.

In a network, a path can be defined as the list of all intermediate routers for a given source destination pair. For a given source/destination pair, the path can be derived by first consulting the forwarding table of the router attached to the source to determine the next router on the path towards the chosen destination. Then, the forwarding table of this router is queried for the same destination... The queries continue until the destination is reached. In a network that has valid forwarding tables, all the paths between all source/destination pairs contain a finite number of intermediate routers. However, if forwarding tables have not been correctly computed, two types of invalid path can occur.

A path may lead to a black hole. In a network, a black hole is a router that receives packets for at least one given source/destination pair but does not have any entry inside its forwarding table for this destination. Since it does not know how to reach the destination, the router cannot forward the received packets and must discard them. Any centralized or distributed algorithm that computes forwarding tables must ensure that there are not black holes inside the network.

A second type of problem may exist in networks using the datagram organisation. Consider a path that contains a cycle. For example, router *R1* sends all packets towards destination *D* via router *R2*, router *R2* forwards these packets to router *R3* and finally router *R3*'s forwarding table uses router *R1* as its nexthop to reach destination *D*. In this case, if a packet destined to *D* is received by router *R1*, it will loop on the *R1* -> *R2* -> *R3* -> *R1* cycle and will never reach its final destination. As in the black hole case, the destination is not reachable from all sources in the network. However, in practice the loop problem is worse than the black hole problem because when a packet is caught in a forwarding loop, it unnecessarily consumes bandwidth. In the black hole case, the problematic packet is quickly discarded. We will see later that network layer protocols include techniques to minimize the impact of such forwarding loops.

Any solution which is used to compute the forwarding tables of a network must ensure that all destinations are reachable from any source. This implies that it must guarantee the absence of black holes and forwarding loops.

The *forwarding tables* and the precise format of the packets that are exchanged inside the network are part of the *data plane* of the network. This *data plane* contains all the protocols and algorithms that are used by hosts and routers to create and process the packets that contain user data. On high-end routers, the data plane is often implemented in hardware for performance reasons.

Besides the *data plane*, a network is also characterized by its *control plane*. The control plane includes all the protocols and algorithms (often distributed) that are used to compute the forwarding tables that are installed on all routers inside the network. While there is only one possible *data plane* for a given networking technology, different networks using the same technology may use different control planes. The simplest *control plane* for a network is always to compute manually the forwarding tables of all routers inside the network. This simple control plane is sufficient when the network is (very) small, usually up to a few routers.

In most networks, manual forwarding tables are not a solution for two reasons. First, most networks are too large to enable a manual computation of the forwarding tables. Second, with manually computed forwarding tables, it is very difficult to deal with link and router failures. Networks need to operate 24h a day, 365 days per year. During the lifetime of a network, many events can affect the routers and links that it contains. Link failures are regular events in deployed networks. Links can fail for various reasons, including electromagnetic interference, fiber cuts, hardware or software problems on the terminating routers, ... Some links also need to be added to the network or removed because their utilisation is too low or their cost is too high. Similarly, routers also fail. There are two types of failures that affect routers. A router may stop forwarding packets due to hardware or software problem (e.g. due to a crash of its operating system). A router may also need to be halted from time to time (e.g. to upgrade its operating system to fix some bugs). These planned and unplanned events affect the set of links and routers that can be used to forward packets in the network. Still, most network users expect that their network will continue to correctly forward packets despite all these events. With manually computed forwarding tables, it is usually impossible to precompute the forwarding tables while taking into account all possible failure scenarios.

An alternative to manually computed forwarding tables is to use a network management platform that tracks the network status and can push new forwarding tables on the routers when it detects any modification to the network topology. This solution gives some flexibility to the network managers in computing the paths inside their network. However, this solution only works if the network management platform is always capable of reaching all routers even when the network topology changes. This may require a dedicated network that allows the management platform to push information on the forwarding tables.

Another interesting point that is worth being discussed is when the forwarding tables are computed. A widely used solution is to compute the entries of the forwarding tables for all destinations on all routers. This ensures that each router has a valid route towards each destination. These entries can be updated when an event occurs and the network topology changes. A drawback of this approach is that the forwarding tables can become large in large networks since each router must maintain one entry for each destination at all times inside its forwarding table.

Some networks use the arrival of packets as the trigger to compute the corresponding entries in the forwarding tables. Several technologies have been built upon this principle. When a packet arrives, the router consults its forwarding table to find a path towards the destination. If the destination is present in the forwarding table, the packet is forwarded. Otherwise, the router needs to find a way to forward the packet and update its forwarding table.

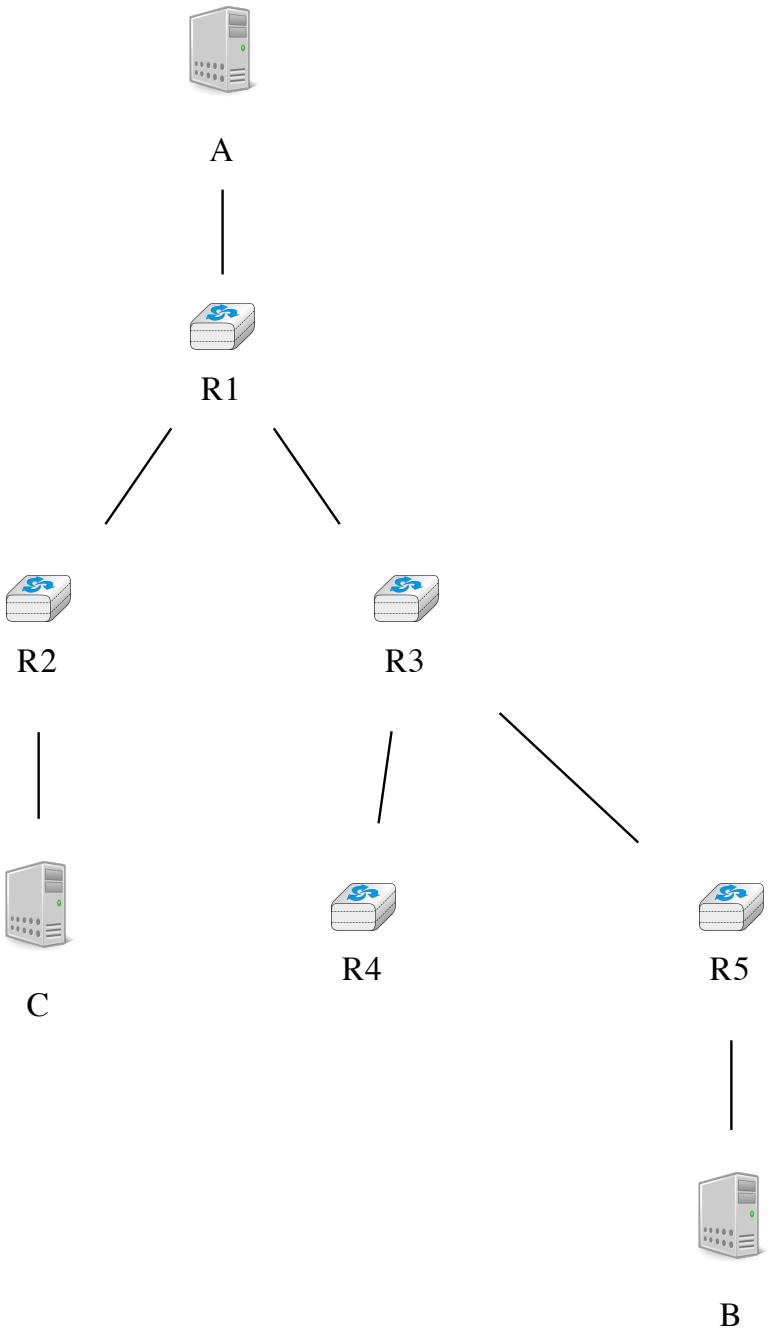
Computing forwarding tables

Several techniques to update the forwarding tables upon the arrival of a packet have been used in deployed networks. In this section, we briefly present the principles that underly three of these techniques.

The first technique assumes that the underlying network topology is a tree. A tree is the simplest network to be considered when forwarding packets. The main advantage of using a tree is that there is only one path between any pair of nodes inside the network. Since a tree does not contain any cycle, it is impossible to have forwarding loops in a tree-shaped network.

In a tree-shaped network, it is relatively simple for each node to automatically compute its forwarding table by inspecting the packets that it receives. For this, each node uses the source and destination addresses present inside each packet. The source address allows to learn the location of the different sources inside the network. Each source has a unique address. When a node receives a packet over a given interface, it learns that the source (address) of this packet is reachable via this interface. The node maintains a data structure that maps each known source address to an incoming interface. This data structure is often called the port-address table since it indicates the interface (or port) to reach a given address. Learning the location of the sources is not sufficient, nodes also need to forward packets towards their destination. When a node receives a packet whose destination address is already present inside its port-address table, it simply forwards the packet on the interface listed in the port-address table. In this case, the packet will follow the port-address table entries in the downstream nodes and will reach the destination. If the destination address is not included in the port-address table, the node simply forwards the packet on all its interfaces, except the interface from which the packet was received. Forwarding a packet over all interfaces is usually called *broadcasting* in the terminology of computer networks. Sending the packet over all interfaces except one is a costly operation since the packet will be sent over links that do not reach the destination. Given the tree-shape of the network, the packet will explore all downstream branches of the tree and will thus finally reach its destination. In practice, the *broadcasting* operation does not occur too often and its cost is limited.

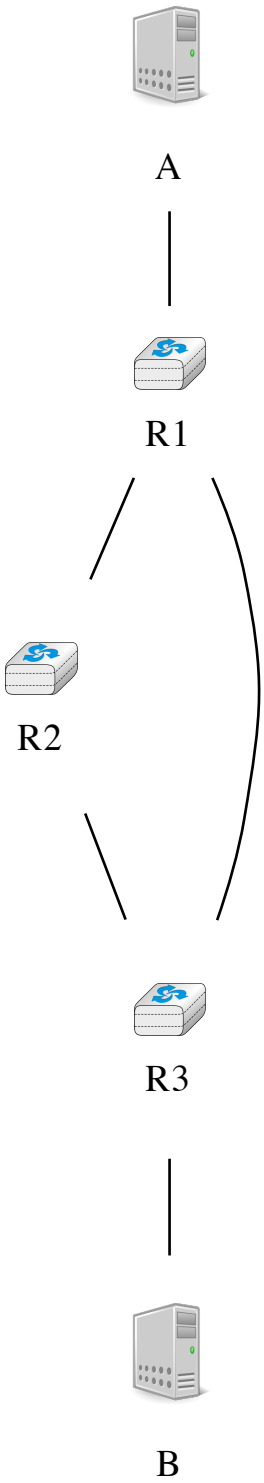
To understand the operation of the port-address table, let us consider the example network shown in the figure below. This network contains three hosts : *A*, *B* and *C* and five nodes, *R1* to *R5*. When the network boots, all the forwarding tables of the nodes are empty.



Host *A* sends a packet towards *B*. When receiving this packet, *R1* learns that *A* is reachable via its *North* interface. Since it does not have an entry for destination *B* in its port-address table, it forwards the packet to both *R2* and *R3*. When *R2* receives the packet, it updates its own forwarding table and forward the packet to *C*. Since *C* is not the intended recipient, it simply discards the received packet. Node *R3* also received the packet. It learns that *A* is reachable via its *North* interface and broadcasts the packet to *R4* and *R5*. *R5* also updates its forwarding table and

finally forwards it to destination *B*.‘Let us now consider what happens when *B* sends a reply to *A*. *R5* first learns that *B* is attached to its *South* port. It then consults its port-address table and finds that *A* is reachable via its *North* interface. The packet is then forwarded hop-by-hop to *A* without any broadcasting. If *C* sends a packet to *B*, this packet will reach *R1* that contains a valid forwarding entry in its forwarding table.

By inspecting the source and destination addresses of packets, network nodes can automatically derive their forwarding tables. As we will discuss later, this technique is used in Ethernet networks. Despite being widely used, it has two important drawbacks. First, packets sent to unknown destinations are broadcasted in the network even if the destination is not attached to the network. Consider the transmission of ten packets destined to *Z* in the network above. When a node receives a packet towards this destination, it can only broadcast the packet. Since *Z* is not attached to the network, no node will ever receive a packet whose source is *Z* to update its forwarding table. The second and more important problem is that few networks have a tree-shaped topology. It is interesting to analyze what happens when a port-address table is used in a network that contains a cycle. Consider the simple network shown below with a single host.



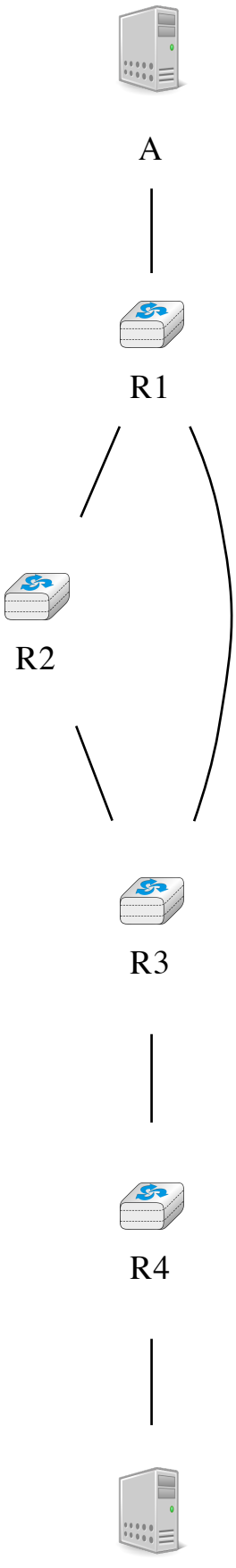
Assume that the network has started and all port-station and forwarding tables are empty. Host *A* sends a packet towards *B*. Upon reception of this packet, *R1* updates its port-address table. Since *B* is not present in the port-

address table, the packet is broadcasted. Both *R2* and *R3* receive a copy of the packet sent by *A*. They both update their port-address table. Unfortunately, they also both broadcast the received packet. *B* receives a first copy of the packet, but *R3* and *R2* receive it again. *R3* will then broadcast this copy of the packet to *B* and *R1* while *R2* will broadcast its copy to *R1*. Although *B* has already received two copies of the packet, it is still inside the network and will continue to loop. Due to the presence of the cycle, a single packet towards an unknown destination generates copies of this packet that loop and will saturate the network bandwidth. Network operators who are using port-address tables to automatically compute the forwarding tables also use distributed algorithms to ensure that the network topology is always a tree.

Another technique can be used to automatically compute forwarding tables. It has been used in interconnecting Token Ring networks and in some wireless networks. Intuitively, *Source routing* enables a destination to automatically discover the paths from a given source towards itself. This technique requires nodes to change some information inside some packets. For simplicity, let us assume that the *data plane* supports two types of packets :

- the *data packets*
- the *control packets*

Data packets are used to exchange data while *control packets* are used to discover the paths between endhosts. With *Source routing*, network nodes can be kept as simple as possible and all the complexity is placed on the endhosts. This is in contrast with the previous technique where the nodes had to maintain a port-address and a forwarding table while the hosts simply sent and received packets. Each node is configured with one unique address and there is one identifier per outgoing link. For simplicity and to avoid cluttering the figures with those identifiers, we will assume that each node uses as link identifiers north, west, south, ... In practice, a node would associate one integer to each outgoing link.


A
R1
R2
R3
R4
B

In the network above, node *R2* is attached to two outgoing links. *R2* is connected to both *R1* and *R3*. *R2* can easily determine that it is connected to these two nodes by exchanging packets with them or observing the packets that it receives over each interface. Assume for example that when a host or node starts, it sends a special control packet over each of its interfaces to advertise its own address to its neighbors. When a host or node receives such a packet, it automatically replies with its own address. This exchange can also be used to verify whether a neighbor, either node or host, is still alive. With *source routing*, the data plane packets include a list of identifiers. This list is called a *source route* and indicates the path to be followed by the packet as a sequence of link identifiers. When a node receives such a *data plane* packet, it first checks whether the packet's destination is direct neighbor. In this case, the packet is forwarded to the destination. Otherwise, the node extracts the next address from the list and forwards it to the neighbor. This allows the source to specify the explicit path to be followed for each packet. For example, in the figure above there are two possible paths between *A* and *B*. To use the path via *R2*, *A* would send a packet that contains *R1,R2,R3* as source route. To avoid going via *R2*, *A* would place *R1,R3* as the source route in its transmitted packet. If *A* knows the complete network topology and all link identifiers, it can easily compute the source route towards each destination. If needed, it could even use different paths, e.g. for redundancy, to reach a given destination. However, in a real network hosts do not usually have a map of the entire network topology.

In networks that rely on source routing, hosts use control packets to automatically discover the best path(s). In addition to the source and destination addresses, *control packets* contain a list that records the intermediate nodes. This list is often called the *record route* because it allows to record the route followed by a given packet. When a node receives a *control packet*, it first checks whether its address is included in the record route. If yes, the control packet is silently discarded. Otherwise, it adds its own address to the *record route* and forwards the packet to all its interfaces, except the interface over which the packet has been received. Thanks to this, the *control packet* will be able to explore all paths between a source and a given destination.

For example, consider again the network topology above. *A* sends a control packet towards *B*. The initial *record route* is empty. When *R1* receives the packet, it adds its own address to the *record route* and forwards a copy to *R2* and another to *R3*. *R2* receives the packet, adds itself to the *record route* and forwards it to *R3*. *R3* receives two copies of the packet. The first contains the *[R1,R2] record route* and the second *[R1]*. In the end, *B* will receive two control packets containing *[R1,R2,R3,R4]* and *[R1,R3,R4]* as *record routes*. *B* can keep these two paths or select the best one and discard the second. A popular heuristic is to select the *record route* of the first received packet as being the best one since this likely corresponds to the shortest delay path.

With the received *record route*, *B* can send a *data packet* to *A*. For this, it simply reverses the chosen *record route*. However, we still need to communicate the chosen path to *A*. This can be done by putting the *record route* inside a control packet which is sent back to *A* over the reverse path. An alternative is to simply send a *data packet* back to *A*. This packet will travel back to *A*. To allow *A* to inspect the entire path followed by the *data packet*, its *source route* must contain all intermediate routers when it is received by *A*. This can be achieved by encoding the *source route* using a data structure that contains an index and the ordered list of node addresses. The index always points to the next address in the *source route*. It is initialized at *0* when a packet is created and incremented by each intermediate node.

Flat or hierarchical addresses

The last, but important, point to discuss about the *data plane* of the networks that rely on the datagram mode is their addressing scheme. In the examples above, we have used letters to represent the addresses of the hosts and network nodes. In practice, all addresses are encoded as a bit string. Most network technologies use a fixed size bit string to represent source and destination address. These addresses can be organized in two different ways.

The first organisation, which is the one that we have implicitly assumed until now, is the *flat addressing* scheme. Under this scheme, each host and network node has a unique address. The unicity of the addresses is important for the operation of the network. If two hosts have the same address, it can become difficult for the network to forward packets towards this destination. *Flat addresses* are typically used in situations where network nodes and hosts need to be able to communicate immediately with unique addresses. These *flat addresses* are often embedded inside the hardware of network interface cards. The network card manufacturer creates one unique address for each interface and this address is stored in the read-only memory of the interface. An advantage of this addressing scheme is that it easily supports ad-hoc and mobile networks. When a host moves, it can attach to another network and remain confident that its address is unique and enables it to communicate inside the new network.

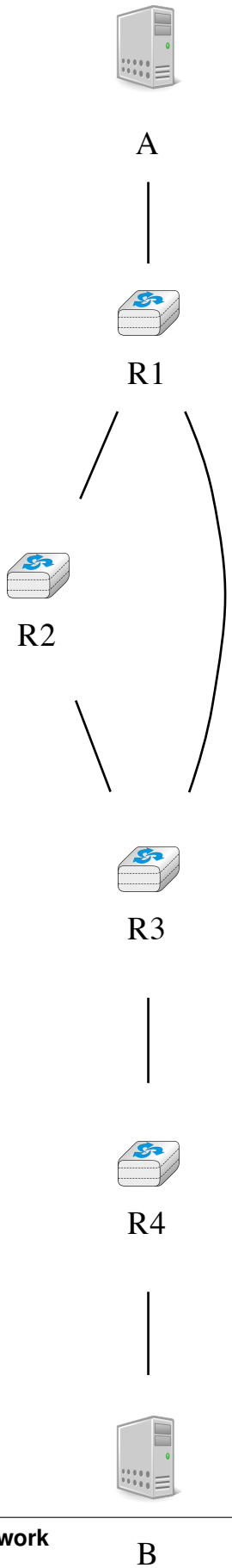
With *flat addressing* the lookup operation in the forwarding table can be implemented as an exact match. The *forwarding table* contains the (sorted) list of all known destination addresses. When a packet arrives, a network

node only needs to check whether this address is part of the forwarding table or not. In software, this is an *O(log(n))* operation if the list is sorted. In hardware, Content Addressable Memories can perform this lookup operation efficiently, but their size is usually limited.

A drawback of the *flat addressing scheme* is that the forwarding tables grow linearly with the number of hosts and nodes in the network. With this addressing scheme, each forwarding table must contain an entry that points to every address reachable inside the network. Since large networks can contain tens of millions or more of hosts, this is a major problem on network nodes that need to be able to quickly forward packets. As an illustration, it is interesting to consider the case of an interface running at 10 Gbps. Such interfaces are found on high-end servers and in various network nodes today. Assuming a packet size of 1000 bits, a pretty large and conservative number, such interface must forward ten million packets every second. This implies that a network node that receives packets over such a link must forward one 1000 bits packet every 100 nanoseconds. This is the same order of magnitude as the memory access times of old DRAMs.

A widely used alternative to the *flat addressing scheme* is the *hierarchical addressing scheme*. This addressing scheme builds upon the fact that networks usually contain much more hosts than network nodes. In this case, a first solution to reduce the size of the forwarding tables is to create a hierarchy of addresses. This is the solution chosen by the post office were addresses contain a country, sometimes a state or province, a city, a street and finally a street number. When an enveloppe is forwarded by a postoffice in a remote country, it only looks at the destination country, while a post office in the same province will look at the city information. Only the post office responsible for a given city will look at the street name and only the postman will use the street number. *Hierarchical addresses* provide a similar solution for network addresses. For example, the address of an Internet host attached to a campus network could contain in the high-order bits an identification of the Internet Service Provider (ISP) that serves the campus network. Then, a subsequent block of bits identifies the campus network which is one of the customers from the ISP. Finally, the low order bits of the address identify the host in the campus network.

This hierarchical allocation of addresses can be applied in any type of network. In practice, the allocation of the addresses must follow the network topology. Usually, this is achieved by dividing the addressing space in consecutive blocks and then allocating these blocks to different parts of the network. In a small network, the simplest solution is to allocate one block of addresses to each network node and assign the host addresses from the attached node.


A
R1
R2
R3
R4
B

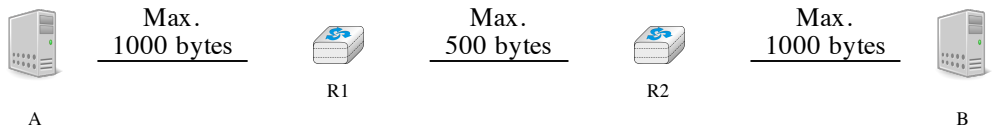
In the above figure, assume that the network uses 16 bits addresses and that the prefix *01001010* has been assigned to the entire network. Since the network contains four routers, the network operator could assign one block of sixty-four addresses to each router. *R1* would use address *0100101000000000* while *A* could use address *0100101000000001*. *R2* could be assigned all adresses from *0100101001000000* to *0100101001111111*. *R4* could then use *0100101011000000* and assign ‘‘0100101011000001‘ to *B*. Other allocation schemes are possible. For example, *R3* could be allocated a larger block of addresses than *R2* and *R4* could use a sub-block from *R3* ‘s address block.

The main advantage of hierarchical addresses is that it is possible to significantly reduce the size of the forwarding tables. In many networks, the number of nodes can be several orders of magnitude smaller than the number of hosts. A campus network may contain a few dozen of network nodes for thousands of hosts. The largest Internet Services Providers typically contain no more than a few tens of thousands of network nodes but still serve tens or hundreds of millions of hosts.

Despite their popularity, *hierarchical addresses* have some drawbacks. Their first drawback is that a lookup in the forwarding table is more complex than when using *flat addresses*. For example, on the Internet, network nodes have to perform a longest-match to forward each packet. This is partially compensated by the reduction in the size of the forwarding tables, but the additional complexity of the lookup operation has been a difficulty to implement hardware support for packet forwarding. A second drawback of the utilisation of hierarchical addresses is that when a host connects for the first time to a network, it must contact one network node to determine its own address. This requires some packet exchanges between the host and some network nodes. Furthermore, if a host moves and is attached to another network node, its network address will change. This can be an issue with some mobile hosts.

Dealing with heterogeneous datalink layers

Sometimes, the network layer needs to deal with heterogenous datalink layers. For example, two hosts connected to different datalink layers exchange packets via routers that are using other types of datalink layers. Thanks to the network layer, this exchange of packets is possible provided that each packet can be placed inside a datalink layer frame before being transmitted. If all datalink layers support the same frame size, this is simple. When a node receives a frame, it decapsulate the packet that it contains, checks the header and forwards it, encapsulated inside another frame, to the outgoing interface. Unfortunately, the encapsulation operation is not always possible. Each datalink layer is characterized by the maximum frame size that it supports. Datalink layers typically support frames containing up to a few hundreds or a few thousands of bytes. The maximum frame size that a given datalink layer supports depends on its underlying technology and unfortunately, most datalink layers support a different maximum frame size. This implies that when a host sends a large packet inside a frame to its nexthop router, there is a risk that this packet will have to traverse a link that is not capable of forwarding the packet inside a single frame. In principle, there are three possibilities to solve this problem. We will discuss them by considering a simpler scenario with two hosts connected to a router as shown in the figure below.



Considering in the network above that host *A* wants to send a 900 bytes packet (870 bytes of payload and 30 bytes of header) to host *B* via router *R1*. Host *A* encapsulates this packet inside a single frame. The frame is received by router *R1* which extracts the packet. Router *R1* has three possible options to process this packet.

1. The packet is too large and router *R1* cannot forward it to router *R2*. It rejects the packet and sends a control packet back to the source (host *A*) to indicate that it cannot forward packets longer than 500 bytes (minus the packet header). The source will have to react to this control packet by retransmitting the information in smaller packets.

2. The network layer is able to fragment a packet. In our example, the router could fragment the packet in two parts. The first part contains the beginning of the payload and the second the end. There are two possible ways to achieve this fragmentation.

1. Router *R1* fragments the packet in two fragments before transmitting them to router *R2*. Router *R2* reassembles the two packet fragments in a larger packet before transmitting them on the link towards host *B*.

2. Each of the packet fragments is a valid packet that contains a header with the source (host *A*) and destination (host *B*) addresses. When router *R2* receives a packet fragment, it treats this packet as a regular packet and forwards it to its final destination (host *B*). Host *B* reassembles the received fragments.

These three solutions have advantages and drawbacks. With the first solution, routers remain simple and do not need to perform any fragmentation operation. This is important when routers are implemented mainly in hardware. However, hosts are more complex since they need to store the packets that they produce if they need to pass through a link that does not support large packets. This increases the buffering required on the end hosts. Furthermore, a single large packet may potentially need to be retransmitted several times. Consider for example a network similar to the one shown above but with four routers. Assume that the link *R1->R2* supports 1000 bytes packets, link *R2->R3* 800 bytes packets and link *R3->R4* 600 bytes packets. A host attached to *R1* that sends large packet will have to first try 1000 bytes, then 800 bytes and finally 600 bytes. Fortunately, this scenario does not occur very often in practice and this is the reason why this solution is used in real networks.

Fragmenting packets on a per-link basis, as presented for the second solution, can minimize the transmission overhead since a packet is only fragmented on the links where fragmentation is required. Large packets can continue to be used downstream of a link that only accepts small packets. However, this reduction of the overhead comes with two drawbacks. First, fragmenting packets, potentially on all links, increases the processing time and the buffer requirements on the routers. Second, this solution leads to a longer end-to-end delay since the downstream router has to reassemble all the packet fragments before forwarding the packet.

The last solution is a compromise between the two others. Routers need to perform fragmentation but they do not need to reassemble packet fragments. Only the hosts need to have buffers to reassemble the received fragments. This solution has a lower end-to-end delay and requires fewer processing time and memory on the routers.

The first solution to the fragmentation problem presented above suggests the utilization of control packets to inform the source about the reception of a too long packet. This is only one of the functions that are performed by the control protocol in the network layer. Other functions include :

- sending a control packet back to the source if a packet is received by a router that does not have a valid entry in its forwarding table
- sending a control packet back to the source if a router detects that a packet is looping inside the network
- verifying that packets can reach a given destination

We will discuss these functions in more details when we will describe the protocols that are used in the network layer of the TCP/IP protocol suite.

2.2.2 Virtual circuit organisation

The second organisation of the network layer, called *virtual circuits*, has been inspired by the organisation of telephone networks. Telephone networks have been designed to carry phone calls that usually last a few minutes. Each phone is identified by a telephone number and is attached to a telephone switch. To initiate a phone call, a telephone first needs to send the destination's phone number to its local switch. The switch cooperates with the other switches in the network to create a bi-directional channel between the two telephones through the network. This channel will be used by the two telephones during the lifetime of the call and will be released at the end of the call. Until the 1960s, most of these channels were created manually, by telephone operators, upon request of the caller. Today's telephone networks use automated switches and allow several channels to be carried over the same physical link, but the principles remain roughly the same.

In a network using virtual circuits, all hosts are also identified with a network layer address. However, packet forwarding is not performed by looking at the destination address of each packet. With the *virtual circuit* organ-

isation, each data packet contains one label [2]. A label is an integer which is part of the packet header. Network nodes implement *label switching* to forward *labelled data packet*. Upon reception of a packet, a network nodes consults its *label forwarding table* to find the outgoing interface for this packet. In contrast with the datagram mode, this lookup is very simple. The *label forwarding table* is an array stored in memory and the label of the incoming packet is the index to access this array. This implies that the lookup operation has an *O(1)* complexity in contrast with other packet forwarding techniques. To ensure that on each node the packet label is an index in the *label forwarding table*, each network node that forwards a packet replaces the label of the forwarded packet with the label found in the *label forwarding table*. Each entry of the *label forwarding table* contains two pieces of information :

- the outgoing interface for the packet
- the label for the outgoing packet

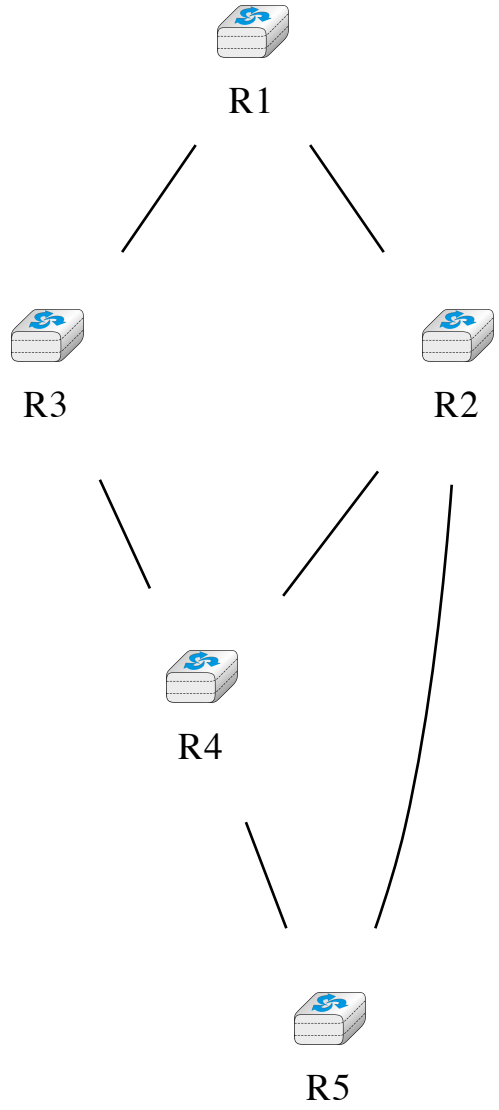
For example, consider the *label forwarding table* of a network node below.

index	outgoing interface	label
0	South	7
1	none	none
2	West	2
3	East	2

If this node receives a packet with *label=2*, it forwards the packet on its *West* interface and sets the *label* of the outgoing packet to *2*. If the received packet's *label* is set to *3*, then the packet is forwarded over the *East* interface and the *label* of the outgoing packet is set to *2*. If a packet is received with a label field set to *1*, the packet is discarded since the corresponding *label forwarding table* entry is invalid.

Label switching enables a full control over the path followed by packets inside the network. Consider the network below and assume that we want to use two virtual circuits : *R1->R3->R4->R2->R5* and *R2->R1->R3->R4->R5*.

[2] We will see later a more detailed description of Multiprotocol Label Switching, a networking technology that is capable of using one or more labels.



To create these virtual circuits, we need to configure the label forwarding tables‘ of all network nodes. For simplicity, assume that a label forwarding table only contains two entries. Assume that *R5* wants to receive the packets from the virtual circuit created by *R1* (resp. *R2*) with *label=1* (*label=0*). *R4* could use the following *label forwarding table*:

index	outgoing interface	label
0	->R2	1
1	->R5	0

Since a packet received with *label=1* must be forwarded to *R5* with *label=1*, *R2*‘s *label forwarding table* could contain :

index	outgoing interface	label
0	none	none
1	->R5	1

Two virtual circuits pass through *R3*. They both need to be forwarded to *R4*, but *R4* expects *label=1* for packets belonging to the virtual circuit originated by *R2* and *label=0* for packets belonging to the other virtual circuit. *R3* could choose to leave the labels unchanged.

index	outgoing interface	label
0	->R4	0
1	->R4	1

With the above *label forwarding table*, *R1* needs to originate the packets that belong to the *R1->R3->R4->R2->R5* with *label=1*. The packets received from *R2* and belonging to the *R2->R1->R3->R4->R5* would then use *label=0* on the *R1-R3* link. *R1*'s label forwarding table could be built as follows :

index	outgoing interface	label
0	->R3	0
1	none	1

The figure below shows the path followed by the packets on the *R1->R3->R4->R2->R5* path in red with on each arrow the label used in the packets.

We will discuss later Multi-Protocol Label Switching (MPLS) as the example of a deployed networking technology that relies on label switching. MPLS is more complex than the above description because it has been designed to be easily integrated with datagram technologies. However, the principles remain. *Asynchronous Transfer Mode'(ATM) and Frame Relay are other examples of technologies that rely on 'label switching*.

Nowadays, most deployed networks rely on distributed algorithms, called routing protocols, to compute the forwarding tables that are installed on the network nodes. These distributed algorithms are part of the *control plane*. They are usually implemented in software and are executed on the main CPU of the network nodes. There are two main families of routing protocols : distance vector routing and link state routing. Both are capable of discovering autonomously the network and react dynamically to topology changes.

2.2.3 The control plane

One of the objectives of the *control plane* in the network layer is to maintain the routing tables that are used on all routers. As indicated earlier, a routing table is a data structure that contains, for each destination address (or block of addresses) known by the router, the outgoing interface over which the router must forward a packet destined to this address. The routing table may also contain additional information such as the address of the next router on the path towards the destination or an estimation of the cost of this path.

In this section, we discuss the main techniques that can be used to maintain the forwarding tables in a network.

Distance vector routing

Distance vector routing is a simple distributed routing protocol. Distance vector routing allows routers to automatically discover the destinations reachable inside the network as well as the shortest path to reach each of these destinations. The shortest path is computed based on *metrics* or *costs* that are associated to each link. We use *l.cost* to represent the metric that has been configured for link *l* on a router.

Each router maintains a routing table. The routing table *R* can be modelled as a data structure that stores, for each known destination address *d*, the following attributes :

- *R[d].link* is the outgoing link that the router uses to forward packets towards destination *d*
- *R[d].cost* is the sum of the metrics of the links that compose the shortest path to reach destination *d*
- *R[d].time* is the timestamp of the last distance vector containing destination *d*

A router that uses distance vector routing regularly sends its distance vector over all its interfaces. The distance vector is a summary of the router's routing table that indicates the distance towards each known destination. This distance vector can be computed from the routing table by using the pseudo-code below.

```
Every N seconds:
 v=Vector()
 for d in R[]:
```

```
    # add destination d to vector
    v.add(Pair(d,R[d].cost))
 for i in interfaces
    # send vector v on this interface
    send(v,interface)
```

When a router boots, it does not know any destination in the network and its routing table only contains itself. It thus sends to all its neighbours a distance vector that contains only its address at a distance of *0*. When a router receives a distance vector on link *l*, it processes it as follows.

```
# V : received Vector
# l : link over which vector is received
def received(V,l):
   # received vector from link l
   for d in V[]
     if not (d in R[]) :
        # new route
        R[d].cost=V[d].cost+l.cost
        R[d].link=l
        R[d].time=now
     else :
        # existing route, is the new better ?
        if ( ((V[d].cost+l.cost) < R[d].cost) or ( R[d].link == l) )  :
             # Better route or change to current route
             R[d].cost=V[d].cost+l.cost
             R[d].link=l
             R[d].time=now
```

The router iterates over all addresses included in the distance vector. If the distance vector contains an address that the router does not know, it inserts the destination inside its routing table via link *l* and at a distance which is the sum between the distance indicated in the distance vector and the cost associated to link *l*. If the destination was already known by the router, it only updates the corresponding entry in its routing table if either :

- the cost of the new route is smaller than the cost of the already known route *((V[d].cost+l.cost) < R[d].cost)*
- the new route was learned over the same link as the current best route towards this destination *(R[d].link == l)*

The first condition ensures that the router discovers the shortest path towards each destination. The second condition is used to take into account the changes of routes that may occur after a link failure or a change of the metric associated to a link.

To understand the operation of a distance vector protocol, let us consider the network of five routers shown below.

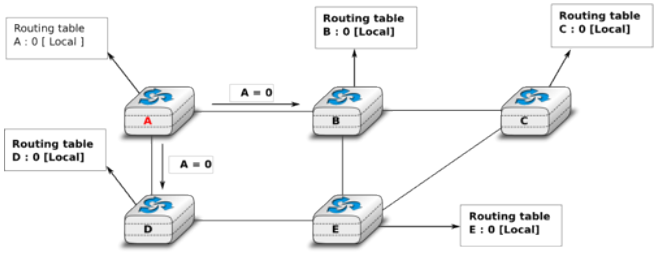


Figure 2.19: Operation of distance vector routing in a simple network

Assume that *A* is the first to send its distance vector *[A=0]*.

- *B* and *D* process the received distance vector and update their routing table with a route towards *A*.
- *D* sends its distance vector *[D=0,A=1]* to *A* and *E*. *E* can now reach *A* and *D*.
- *C* sends its distance vector *[C=0]* to *B* and *E*

- *E* sends its distance vector *[E=0,D=1,A=2,C=1]* to *D*, *B* and *C*. *B* can now reach *A*, *C*, *D* and *E*
- *B* sends its distance vector *[B=0,A=1,C=1,D=2,E=1]* to *A*, *C* and *E*. *A*, *B*, *C* and *E* can now reach all destinations.
- *A* sends its distance vector *[A=0,B=1,C=2,D=1,E=2]* to *B* and *D*.

At this point, all routers can reach all other routers in the network thanks to the routing tables shown in the figure below.

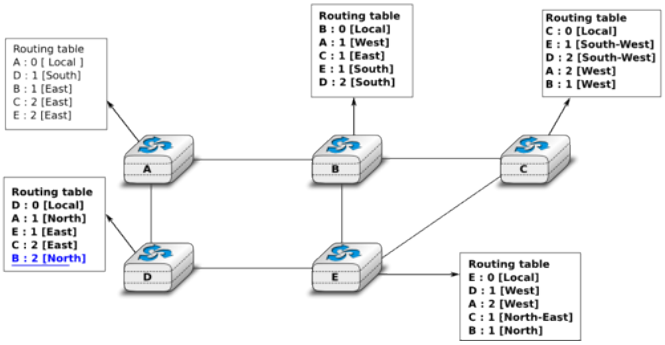


Figure 2.20: Routing tables computed by distance vector in a simple network

To deal with link and router failures, routers use the timestamp stored in their routing table. As all routers send their distance vector every *N* seconds, the timestamp of each route should be regularly refreshed. Thus no route should have a timestamp older than *N* seconds, unless the route is not reachable anymore. In practice, to cope with the possible loss of a distance vector due to transmission errors, routers check the timestamp of the routes stored in their routing table every *N* seconds and remove the routes that are older than $3 \times N$ seconds. When a router notices that a route towards a destination has expired, it must first associate an ∞ cost to this route and send its distance vector to its neighbours to inform them. The route can then be removed from the routing table after some time (e.g. $3 \times N$ seconds), to ensure that the neighbouring routers have received the bad news, even if some distance vectors do not reach them due to transmission errors.

Consider the example above and assume that the link between routers *A* and *B* fails. Before the failure, *A* used *B* to reach destinations *B*, *C* and *E* while *B* only used the *A*-*B* link to reach *A*. The affected entries timeout on routers *A* and *B* and they both send their distance vector.

- *A* sends its distance vector $[A = 0, D = \infty, C = \infty, D = 1, E = \infty]$. *D* knows that it cannot reach *B* anymore via *A*
- *D* sends its distance vector $[D = 0, B = \infty, A = 1, C = 2, E = 1]$ to *A* and *E*. *A* recovers routes towards *C* and *E* via *D*.
- *B* sends its distance vector $[B = 0, A = \infty, C = 1, D = 2, E = 1]$ to *E* and *C*. *D* learns that there is no route anymore to reach *A* via *B*.
- *E* sends its distance vector $[E = 0, A = 2, C = 1, D = 1, B = 1]$ to *D*, *B* and *C*. *D* learns a route towards *B*. *C* and *B* learn a route towards *A*.

At this point, all routers have a routing table allowing them to reach all another routers, except router *A*, which cannot yet reach router *B*. *A* recovers the route towards *B* once router *D* sends its updated distance vector $[A = 1, B = 2, C = 2, D = 1, E = 1]$. This last step is illustrated in figure *Routing tables computed by distance vector after a failure*, which shows the routing tables on all routers.

Consider now that the link between *D* and *E* fails. The network is now partitioned into two disjoint parts : (*A* , *D*) and (*B*, *E*, *C*). The routes towards *B*, *C* and *E* expire first on router *D*. At this time, router *D* updates its routing table.

If *D* sends $[D = 0, A = 1, B = \infty, C = \infty, E = \infty]$, *A* learns that *B*, *C* and *E* are unreachable and updates its routing table.

Unfortunately, if the distance vector sent to *A* is lost or if *A* sends its own distance vector ($[A = 0, D = 1, B = 3, C = 3, E = 2]$) at the same time as *D* sends its distance vector, *D* updates its routing table to use the

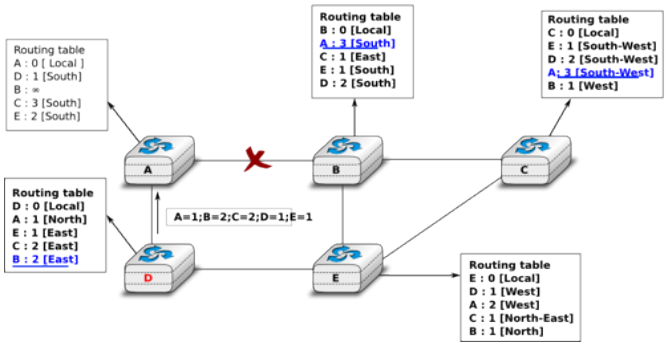


Figure 2.21: Routing tables computed by distance vector after a failure

shorter routes advertised by A towards B, C and E. After some time D sends a new distance vector : $[D = 0, A = 1, E = 3, C = 4, B = 4]$. A updates its routing table and after some time sends its own distance vector $[A = 0, D = 1, B = 5, C = 5, E = 4]$, etc. This problem is known as the *count to infinity problem* in networking literature. Routers A and D exchange distance vectors with increasing costs until these costs reach ∞. This problem may occur in other scenarios than the one depicted in the above figure. In fact, distance vector routing may suffer from count to infinity problems as soon as there is a cycle in the network. Cycles are necessary to have enough redundancy to deal with link and router failures. To mitigate the impact of counting to infinity, some distance vector protocols consider that $16 = \infty$. Unfortunately, this limits the metrics that network operators can use and the diameter of the networks using distance vectors.

This count to infinity problem occurs because router A advertises to router D a route that it has learned via router D. A possible solution to avoid this problem could be to change how a router creates its distance vector. Instead of computing one distance vector and sending it to all its neighbors, a router could create a distance vector that is specific to each neighbour and only contains the routes that have not been learned via this neighbour. This could be implemented by the following pseudocode.

```
Every N seconds:
 # one vector for each interface
 for l in interfaces:
   v=Vector()
   for d in R[]:
     if (R[d].link != i) :
        v=v+Pair(d,R[d.cost])
   send(v)
   # end for d in R[]
 #end for l in interfaces
```

This technique is called *split-horizon*. With this technique, the count to infinity problem would not have happened in the above scenario, as router A would have advertised $[A = 0]$, since it learned all its other routes via router D. Another variant called *split-horizon with poison reverse* is also possible. Routers using this variant advertise a cost of ∞ for the destinations that they reach via the router to which they send the distance vector. This can be implemented by using the pseudo-code below.

```
Every N seconds:
 for l in interfaces:
   # one vector for each interface
   v=Vector()
   for d in R[]:
     if (R[d].link != i) :
        v=v+Pair(d,R[d.cost])
     else:
        v=v+Pair(d,infinity);
   send(v)
   # end for d in R[]
 #end for l in interfaces
```

Unfortunately, split-horizon, is not sufficient to avoid all count to infinity problems with distance vector routing.

Consider the failure of link *A*-*B* in the network of four routers below.

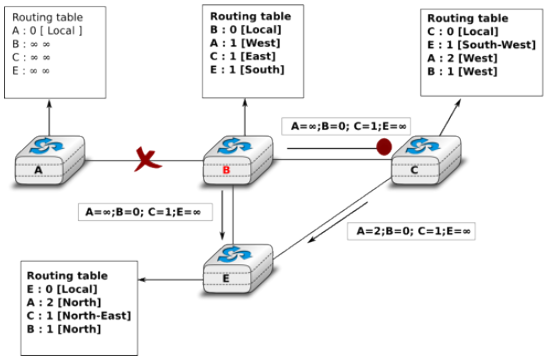


Figure 2.22: Count to infinity problem

After having detected the failure, router *B* sends its distance vectors :

- $[A = \infty, B = 0, C = \infty, E = 1]$ to router *C*
- $[A = \infty, B = 0, C = 1, E = \infty]$ to router *E*

If, unfortunately, the distance vector sent to router *C* is lost due to a transmission error or because router *C* is overloaded, a new count to infinity problem can occur. If router *C* sends its distance vector $[A = 2, B = 1, C = 0, E = \infty]$ to router *E*, this router installs a route of distance *3* to reach *A* via *C*. Router *E* sends its distance vectors $[A = 3, B = \infty, C = 1, E = 1]$ to router *B* and $[A = \infty, B = 1, C = \infty, E = 0]$ to router *C*. This distance vector allows *B* to recover a route of distance *4* to reach *A*.

Note: Forwarding tables versus routing tables

Routers usually maintain at least two data structures that contain information the reachable destinations. The first data structure is the *routing table*. The *routing table* is a data structure that associates a destination to an outgoing interface or a nexthop router and a set of additional attributes. Different routing protocols can associate different attributes for each destination. Distance vector routing protocols will store the cost to reach the destination along the shortest path. Other routing protocols may store information about the number of hops of the best path, its lifetime or the number of sub paths. A *routing table* may store multipath paths towards a given destination and flag one of them as the best one. The *routing table* is a software data structure which is updated by (one or more) routing protocols. The *routing table* is usually not directly used when forwarding packets. Packet forwarding relies on a more compact data structure which is the *forwarding table*. On high-end routers, the *forwarding table* is implemented directly in hardware while lower performance routers will use a software implementation. A *forwarding table* contains a subset of the information found in the *routing table*. It only contains the paths that are used to forward packets and no attributes. A *forwarding table* will typically associate each destination to an outgoing interface or nexthop router.

Link state routing

Link state routing is the second family of routing protocols. While distance vector routers use a distributed algorithm to compute their routing tables, link-state routers exchange messages to allow each router to learn the entire network topology. Based on this learned topology, each router is then able to compute its routing table by using a shortest path computation [Dijkstra1959].

For link-state routing, a network is modelled as a *directed weighted graph*. Each router is a node, and the links between routers are the edges in the graph. A positive weight is associated to each directed edge and routers use the shortest path to reach each destination. In practice, different types of weight can be associated to each directed edge :

- unit weight. If all links have a unit weight, shortest path routing prefers the paths with the least number of intermediate routers.

- weight proportional to the propagation delay on the link. If all link weights are configured this way, shortest path routing uses the paths with the smallest propagation delay.
- $weight = \frac{C}{bandwidth}$ where C is a constant larger than the highest link bandwidth in the network. If all link weights are configured this way, shortest path routing prefers higher bandwidth paths over lower bandwidth paths

Usually, the same weight is associated to the two directed edges that correspond to a physical link (i.e. $R1 \rightarrow R2$ and $R2 \rightarrow R1$). However, nothing in the link state protocols requires this. For example, if the weight is set in function of the link bandwidth, then an asymmetric ADSL link could have a different weight for the upstream and downstream directions. Other variants are possible. Some networks use optimisation algorithms to find the best set of weights to minimize congestion inside the network for a given traffic demand [FRT2002].

When a link-state router boots, it first needs to discover to which routers it is directly connected. For this, each router sends a HELLO message every N seconds on all of its interfaces. This message contains the router's address. Each router has a unique address. As its neighbouring routers also send HELLO messages, the router automatically discovers to which neighbours it is connected. These HELLO messages are only sent to neighbours who are directly connected to a router, and a router never forwards the HELLO messages that they receive. HELLO messages are also used to detect link and router failures. A link is considered to have failed if no HELLO message has been received from the neighbouring router for a period of $k \times N$ seconds.

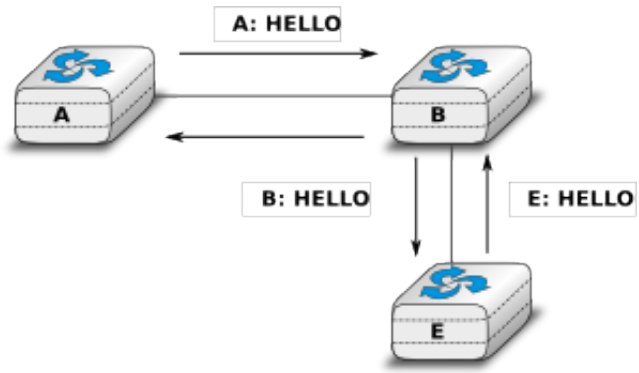


Figure 2.23: The exchange of HELLO messages

Once a router has discovered its neighbours, it must reliably distribute its local links to all routers in the network to allow them to compute their local view of the network topology. For this, each router builds a *link-state packet* (LSP) containing the following information :

- LSP.Router : identification (address) of the sender of the LSP
- LSP.age : age or remaining lifetime of the LSP
- LSP.seq : sequence number of the LSP
- LSP.Links[] : links advertised in the LSP. Each directed link is represented with the following information : - LSP.Links[i].Id : identification of the neighbour - LSP.Links[i].cost : cost of the link

These LSPs must be reliably distributed inside the network without using the router's routing table since these tables can only be computed once the LSPs have been received. The *Flooding* algorithm is used to efficiently distribute the LSPs of all routers. Each router that implements *flooding* maintains a *link state database* (LSDB) containing the most recent LSP sent by each router. When a router receives an LSP, it first verifies whether this LSP is already stored inside its LSDB. If so, the router has already distributed the LSP earlier and it does not need to forward it. Otherwise, the router forwards the LSP on all links except the link over which the LSP was received. Flooding can be implemented by using the following pseudo-code.

```
# links is the set of all links on the router
# Router R's LSP arrival on link l
if newer(LSP, LSDB(LSP.Router)) :
  LSDB.add(LSP)
  for i in links :
    if i!=l :
```

```
        send(LSP,i)
else:
 # LSP has already been flooded
```

In this pseudo-code, *LSDB(r)* returns the most recent *LSP* originating from router *r* that is stored in the *LSDB*. *newer(lsp1,lsp2)* returns true if *lsp1* is more recent than *lsp2*. See the note below for a discussion on how *newer* can be implemented.

Note: Which is the most recent LSP ?

A router that implements flooding must be able to detect whether a received LSP is newer than the stored LSP. This requires a comparison between the sequence number of the received LSP and the sequence number of the LSP stored in the link state database. The ARPANET routing protocol [MRR1979] used a 6 bits sequence number and implemented the comparison as follows **RFC 789**

```
def newer( lsp1, lsp2 ):
  return ( ( ( lsp1.seq > lsp2.seq) and ( (lsp1.seq-lsp2.seq)<=32) ) or
           ( ( lsp1.seq < lsp2.seq) and ( (lsp2.seq-lsp1.seq)> 32) )    )
```

This comparison takes into account the modulo 2^6 arithmetic used to increment the sequence numbers. Intuitively, the comparison divides the circle of all sequence numbers into two halves. Usually, the sequence number of the received LSP is equal to the sequence number of the stored LSP incremented by one, but sometimes the sequence numbers of two successive LSPs may differ, e.g. if one router has been disconnected from the network for some time. The comparison above worked well until October 27, 1980. On this day, the ARPANET crashed completely. The crash was complex and involved several routers. At one point, LSP *40* and LSP *44* from one of the routers were stored in the LSDB of some routers in the ARPANET. As LSP *44* was the newest, it should have replaced LSP *40* on all routers. Unfortunately, one of the ARPANET routers suffered from a memory problem and sequence number *40* (*101000* in binary) was replaced by *8* (*001000* in binary) in the buggy router and flooded. Three LSPs were present in the network and *44* was newer than *40* which is newer than *8*, but unfortunately *8* was considered to be newer than *44*... All routers started to exchange these three link state packets for ever and the only solution to recover from this problem was to shutdown the entire network **RFC 789**.

Current link state routing protocols usually use 32 bits sequence numbers and include a special mechanism in the unlikely case that a sequence number reaches the maximum value (using a 32 bits sequence number space takes 136 years if a link state packet is generated every second).

To deal with the memory corruption problem, link state packets contain a checksum. This checksum is computed by the router that generates the LSP. Each router must verify the checksum when it receives or floods an LSP. Furthermore, each router must periodically verify the checksums of the LSPs stored in its LSDB.

Flooding is illustrated in the figure below. By exchanging HELLO messages, each router learns its direct neighbours. For example, router *E* learns that it is directly connected to routers *D*, *B* and *C*. Its first LSP has sequence number *0* and contains the directed links *E->D*, *E->B* and *E->C*. Router *E* sends its LSP on all its links and routers *D*, *B* and *C* insert the LSP in their LSDB and forward it over their other links.

Flooding allows LSPs to be distributed to all routers inside the network without relying on routing tables. In the example above, the LSP sent by router *E* is likely to be sent twice on some links in the network. For example, routers *B* and *C* receive *E*'s LSP at almost the same time and forward it over the *B-C* link. To avoid sending the same LSP twice on each link, a possible solution is to slightly change the pseudo-code above so that a router waits for some random time before forwarding a LSP on each link. The drawback of this solution is that the delay to flood an LSP to all routers in the network increases. In practice, routers immediately flood the LSPs that contain new information (e.g. addition or removal of a link) and delay the flooding of refresh LSPs (i.e. LSPs that contain exactly the same information as the previous LSP originating from this router) [FFEB2005].

To ensure that all routers receive all LSPs, even when there are transmissions errors, link state routing protocols use *reliable flooding*. With *reliable flooding*, routers use acknowledgements and if necessary retransmissions to ensure that all link state packets are successfully transferred to all neighbouring routers. Thanks to reliable flooding, all routers store in their LSDB the most recent LSP sent by each router in the network. By combining the received LSPs with its own LSP, each router can compute the entire network topology.

Note: Static or dynamic link metrics ?

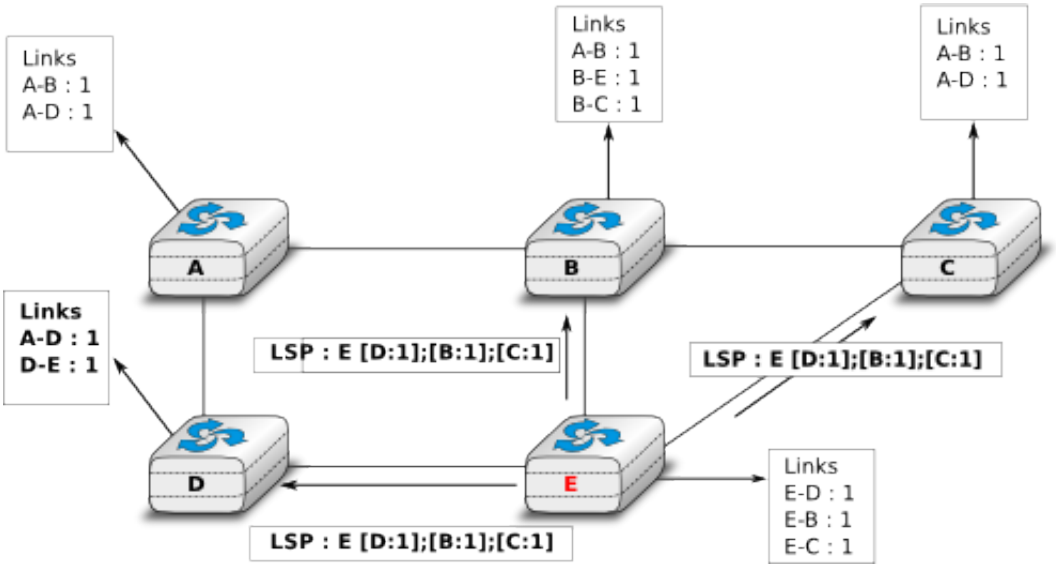


Figure 2.24: Flooding : example

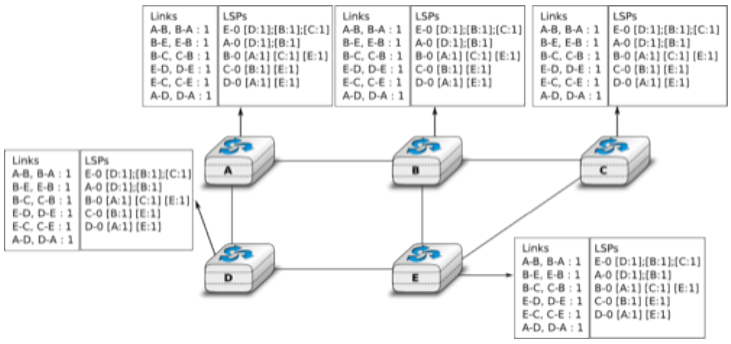


Figure 2.25: Link state databases received by all routers

As link state packets are flooded regularly, routers are able to measure the quality (e.g. delay or load) of their links and adjust the metric of each link according to its current quality. Such dynamic adjustments were included in the ARPANET routing protocol [MRR1979] . However, experience showed that it was difficult to tune the dynamic adjustments and ensure that no forwarding loops occur in the network [KZ1989]. Today's link state routing protocols use metrics that are manually configured on the routers and are only changed by the network operators or network management tools [FRT2002].

When a link fails, the two routers attached to the link detect the failure by the lack of HELLO messages received in the last $k \times N$ seconds. Once a router has detected a local link failure, it generates and floods a new LSP that no longer contains the failed link and the new LSP replaces the previous LSP in the network. As the two routers attached to a link do not detect this failure exactly at the same time, some links may be announced in only one direction. This is illustrated in the figure below. Router *E* has detected the failures of link *E-B* and flooded a new LSP, but router *B* has not yet detected the failure.

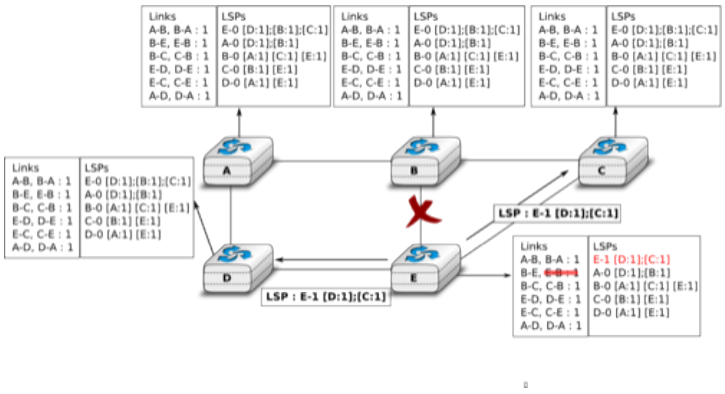


Figure 2.26: The two-way connectivity check

When a link is reported in the LSP of only one of the attached routers, routers consider the link as having failed and they remove it from the directed graph that they compute from their LSDB. This is called the *two-way connectivity check*. This check allows link failures to be flooded quickly as a single LSP is sufficient to announce such bad news. However, when a link comes up, it can only be used once the two attached routers have sent their LSPs. The *two-way connectivity check* also allows for dealing with router failures. When a router fails, all its links fail by definition. Unfortunately, it does not, of course, send a new LSP to announce its failure. The *two-way connectivity check* ensures that the failed router is removed from the graph.

When a router has failed, its LSP must be removed from the LSDB of all routers [3]. This can be done by using the *age* field that is included in each LSP. The *age* field is used to bound the maximum lifetime of a link state packet in the network. When a router generates a LSP, it sets its lifetime (usually measured in seconds) in the *age* field. All routers regularly decrement the *age* of the LSPs in their LSDB and a LSP is discarded once its *age* reaches *0*. Thanks to the *age* field, the LSP from a failed router does not remain in the LSDBs forever.

To compute its forwarding table, each router computes the spanning tree rooted at itself by using Dijkstra's shortest path algorithm [Dijkstra1959]. The forwarding table can be derived automatically from the spanning as shown in the figure below.

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=3

[3] It should be noted that link state routing assumes that all routers in the network have enough memory to store the entire LSDB. The routers that do not have enough memory to store the entire LSDB cannot participate in link state routing. Some link state routing protocols allow routers to report that they do not have enough memory and must be removed from the graph by the other routers in the network.

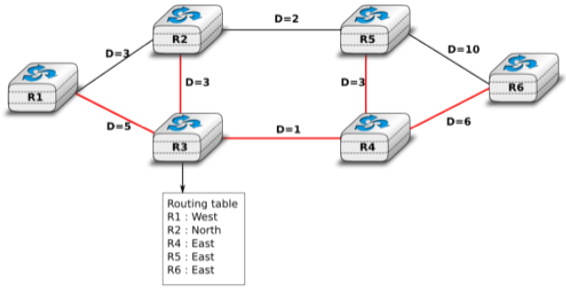


Figure 2.27: Computation of the forwarding table

2.3 Applications

The are two important models used to organise a networked application. The first and oldest model is the client-server model. In this model, a server provides services to clients that exchange information with it. This model is highly asymmetrical : clients send requests and servers perform actions and return responses. It is illustrated in the figure below.

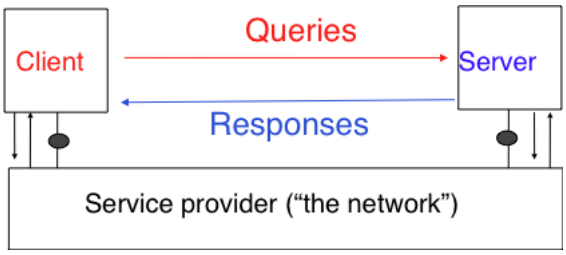


Figure 2.28: The client-server model

The client-server model was the first model to be used to develop networked applications. This model comes naturally from the mainframes and minicomputers that were the only networked computers used until the 1980s. A minicomputer is a multi-user system that is used by tens or more users at the same time. Each user interacts with the minicomputer by using a terminal. Those terminals, were mainly a screen, a keyboard and a cable directly connected to the minicomputer.

There are various types of servers as well as various types of clients. A web server provides information in response to the query sent by its clients. A print server prints documents sent as queries by the client. An email server will forward towards their recipient the email messages sent as queries while a music server will deliver the music requested by the client. From the viewpoint of the application developer, the client and the server applications directly exchange messages (the horizontal arrows labelled *Queries* and *Responses* in the above figure), but in practice these messages are exchanged thanks to the underlying layers (the vertical arrows in the above figure). In this chapter, we focus on these horizontal exchanges of messages.

Networked applications do not exchange random messages. In order to ensure that the server is able to understand the queries sent by a client, and also that the client is able to understand the responses sent by the server, they must both agree on a set of syntactical and semantic rules. These rules define the format of the messages exchanged as well as their ordering. This set of rules is called an application-level *protocol*.

An *application-level protocol* is similar to a structured conversation between humans. Assume that Alice wants to know the current time but does not have a watch. If Bob passes close by, the following conversation could take place :

- Alice : *Hello*
- Bob : *Hello*
- Alice : *What time is it ?*

- Bob : *11:55*
- Alice : *Thank you*
- Bob : *You're welcome*

Such a conversation succeeds if both Alice and Bob speak the same language. If Alice meets Tchang who only speaks Chinese, she won't be able to ask him the current time. A conversation between humans can be more complex. For example, assume that Bob is a security guard whose duty is to only allow trusted secret agents to enter a meeting room. If all agents know a secret password, the conversation between Bob and Trudy could be as follows :

- Bob : *What is the secret password ?*
- Trudy : *1234*
- Bob : *This is the correct password, you're welcome*

If Alice wants to enter the meeting room but does not know the password, her conversation could be as follows :

- Bob : *What is the secret password ?*
- Alice : *3.1415*
- Bob : *This is not the correct password.*

Human conversations can be very formal, e.g. when soldiers communicate with their hierarchy, or informal such as when friends discuss. Computers that communicate are more akin to soldiers and require well-defined rules to ensure an successful exchange of information. There are two types of rules that define how information can be exchanged between computers :

- syntactical rules that precisely define the format of the messages that are exchanged. As computers only process bits, the syntactical rules specify how information is encoded as bit strings
- organisation of the information flow. For many applications, the flow of information must be structured and there are precedence relationships between the different types of information. In the time example above, Alice must greet Bob before asking for the current time. Alice would not ask for the current time first and greet Bob afterwards. Such precedence relationships exist in networked applications as well. For example, a server must receive a username and a valid password before accepting more complex commands from its clients.

Let us first discuss the syntactical rules. We will later explain how the information flow can be organised by analysing real networked applications.

Application-layer protocols exchange two types of messages. Some protocols such as those used to support electronic mail exchange messages expressed as strings or lines of characters. As the transport layer allows hosts to exchange bytes, they need to agree on a common representation of the characters. The first and simplest method to encode characters is to use the *ASCII* table. **RFC 20** provides the ASCII table that is used by many protocols on the Internet. For example, the table defines the following binary representations :

- *A* : *1000011b*
- *0* : *0110000b*
- *z* : *1111010b*
- *@* : *1000000b*
- *space* : *0100000b*

In addition, the *ASCII* table also defines several non-printable or control characters. These characters were designed to allow an application to control a printer or a terminal. These control characters include *CR* and *LF*, that are used to terminate a line, and the *Bell* character which causes the terminal to emit a sound.

- *carriage return* (*CR*) : *0001101b*
- *line feed* (*LF*) : *0001010b*
- *Bell*: *0000111b*

The *ASCII* characters are encoded as a seven bits field, but transmitted as an eight-bits byte whose high order bit is usually set to *0*. Bytes are always transmitted starting from the high order or most significant bit.

Most applications exchange strings that are composed of fixed or variable numbers of characters. A common solution to define the character strings that are acceptable is to define them as a grammar using a Backus-Naur Form (*BNF*) such as the Augmented BNF defined in **RFC 5234**. A BNF is a set of production rules that generate all valid character strings. For example, consider a networked application that uses two commands, where the user can supply a username and a password. The BNF for this application could be defined as shown in the figure below.

```
command         = usercommand / passworcommand
usercommand     = "user" SP username CRLF
passwordcommand = "pass" SP password CRLF
username        = 1*8ALPHA
password        = (ALPHA) *(ALPHA/DIGIT)
ALPHA           =  %x41-5A / %x61-7A
CR              =  %x0D
CRLF            =  CR LF
DIGIT           =  "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
LF              =  %x0A
SP              =  %x20 / %x09
```

Figure 2.29: A simple BNF specification

The example above defines several terminals and two commands : *usercommand* and *passwordcommand*. The *ALPHA* terminal contains all letters in upper and lower case. In the *ALPHA* rule, *%x41* corresponds to ASCII character code 41 in hexadecimal, i.e. capital *A*. The *CR* and *LF* terminals correspond to the carriage return and linefeed control characters. The *CRLF* rule concatenates these two terminals to match the standard end of line termination. The *DIGIT* terminal contains all digits. The *SP* terminal corresponds to the white space characters. The *usercommand* is composed of two strings separated by white space. In the ABNF rules that define the messages used by Internet applications, the commands are case-insensitive. The rule *"user"* corresponds to all possible cases of the letters that compose the word between brackets, e.g. *user*, *uSeR*, *USER*, *usER*, ... A *username* contains at least one letter and up to 8 letters. User names are case-sensitive as they are not defined as a string between brackets. The *password* rule indicates that a password starts with a letter and can contain any number of letters or digits. The white space and the control characters cannot appear in a *password* defined by the above rule.

Besides character strings, some applications also need to exchange 16 bits and 32 bits fields such as integers. A naive solution would have been to send the 16- or 32-bits field as it is encoded in the host's memory. Unfortunately, there are different methods to store 16- or 32-bits fields in memory. Some CPUs store the most significant byte of a 16-bits field in the first address of the field while others store the least significant byte at this location. When networked applications running on different CPUs exchange 16 bits fields, there are two possibilities to transfer them over the transport service :

- send the most significant byte followed by the least significant byte
- send the least significant byte followed by the most significant byte

The first possibility was named *big-endian* in a note written by Cohen [Cohen1980] while the second was named *little-endian*. Vendors of CPUs that used *big-endian* in memory insisted on using *big-endian* encoding in networked applications while vendors of CPUs that used *little-endian* recommended the opposite. Several studies were written on the relative merits of each type of encoding, but the discussion became almost a religious issue [Cohen1980]. Eventually, the Internet chose the *big-endian* encoding, i.e. multi-byte fields are always transmitted by sending the most significant byte first, **RFC 791** refers to this encoding as the *network-byte order*. Most libraries [4] used to write networked applications contain functions to convert multi-byte fields from memory to the network byte order and vice versa.

Besides 16 and 32 bit words, some applications need to exchange data structures containing bit fields of various lengths. For example, a message may be composed of a 16 bits field followed by eight, one bit flags, a 24 bits field and two 8 bits bytes. Internet protocol specifications will define such a message by using a representation such as the one below. In this representation, each line corresponds to 32 bits and the vertical lines are used to delineate fields. The numbers above the lines indicate the bit positions in the 32-bits word, with the high order bit at position *0*.

[4] For example, the `htonl(3)` (resp. `ntohl(3)`) function the standard C library converts a 32-bits unsigned integer from the byte order used by the CPU to the network byte order (resp. from the network byte order to the CPU byte order). Similar functions exist in other

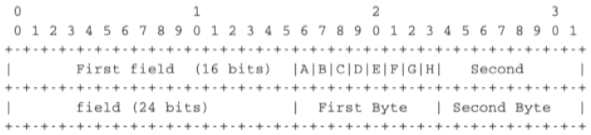


Figure 2.30: Message format

The message mentioned above will be transmitted starting from the upper 32-bits word in network byte order. The first field is encoded in 16 bits. It is followed by eight one bit flags (*A*-*H*), a 24 bits field whose high order byte is shown in the first line and the two low order bytes appear in the second line followed by two one byte fields. This ASCII representation is frequently used when defining binary protocols. We will use it for all the binary protocols that are discussed in this book.

The peer-to-peer model emerged during the last ten years as another possible architecture for networked applications. In the traditional client-server model, hosts act either as servers or as clients and a server serves a large number of clients. In the peer-to-peer model, all hosts act as both servers and clients and they play both roles. The peer-to-peer model has been used to develop various networked applications, ranging from Internet telephony to file sharing or Internet-wide filesystems. A detailed description of peer-to-peer applications may be found in [BYL2008]. Surveys of peer-to-peer protocols and applications may be found in [AS2004] and [LCP2005].

2.4 The transport layer

A network is always designed and built to enable applications running on hosts to exchange information. In the previous chapter, we have explained the principles of the *network layer* that enables hosts connected to different types of datalink layers to exchange information through routers. These routers act as relays in the network layer and ensure the delivery of packets between any pair of hosts attached to the network.

The network layer ensures the delivery of packets on a hop-by-hop basis through intermediate nodes. As such, it provides a service to the upper layer. In practice, this layer is usually the *transport layer* that improves the service provided by the *network layer* to make it useable by applications.

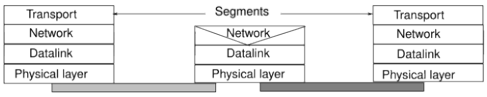


Figure 2.31: The transport layer

Most networks use a datagram organisation and provide a simple service which is called the *connectionless service*.

The figure below provides a representation of the connectionless service as a *time-sequence diagram*. The user on the left, having address *S*, issues a *Data.request* primitive containing Service Data Unit (SDU) *M* that must be delivered by the service provider to destination *D*. The dashed line between the two primitives indicates that the *Data.indication* primitive that is delivered to the user on the right corresponds to the *Data.request* primitive sent by the user on the left.

There are several possible implementations of the connectionless service. Before studying these realisations, it is useful to discuss the possible characteristics of the connectionless service. A *reliable connectionless service* is a service where the service provider guarantees that all SDUs submitted in *Data.requests* by a user will eventually be delivered to their destination. Such a service would be very useful for users, but guaranteeing perfect delivery is difficult in practice. For this reason, network layers usually support an *unreliable connectionless service*.

An *unreliable connectionless* service may suffer from various types of problems compared to a *reliable connectionless service*. First of all, an *unreliable connectionless service* does not guarantee the delivery of all SDUs. This can be expressed graphically by using the time-sequence diagram below.

programming languages.

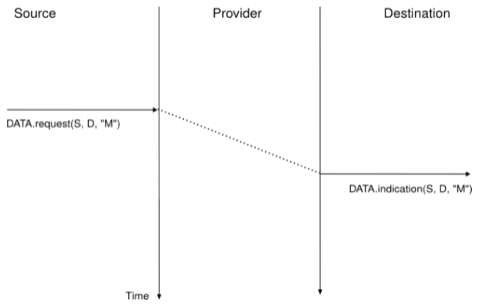


Figure 2.32: A simple connectionless service

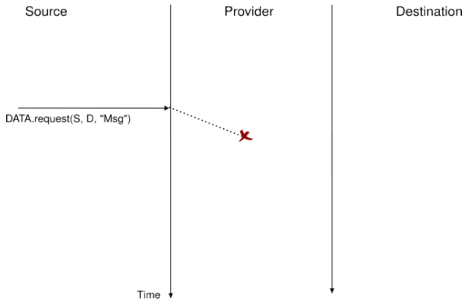


Figure 2.33: An unreliable connectionless service may loose SDUs

In practice, an *unreliable connectionless service* will usually deliver a large fraction of the SDUs. However, since the delivery of SDUs is not guaranteed, the user must be able to recover from the loss of any SDU.

A second imperfection that may affect an *unreliable connectionless service* is that it may duplicate SDUs. Some packets may be duplicated in a network and be delivered twice to their destination. This is illustrated by the time-sequence diagram below.

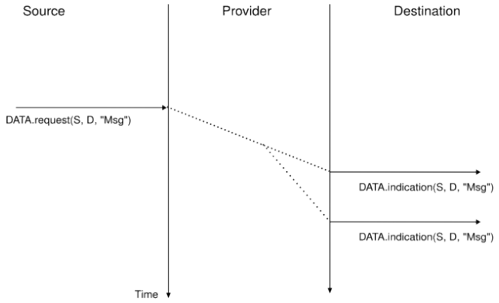


Figure 2.34: An unreliable connectionless service may duplicate SDUs

Finally, some unreliable connectionless service providers may deliver to a destination a different SDU than the one that was supplied in the *Data.request*. This is illustrated in the figure below.

As the transport layer is built on top of the network layer, it is important to know the key features of the network layer service. In this book, we only consider the *connectionless network layer service* which is the most widespread. Its main characteristics are :

- the *connectionless network layer service* can only transfer SDUs of *limited size*
- the *connectionless network layer service* may discard SDUs
- the *connectionless network layer service* may corrupt SDUs
- the *connectionless network layer service* may delay, reorder or even duplicate SDUs

These imperfections of the *connectionless network layer service* are caused by the operations of the *network layer*. This *layer* is able to deliver packets to their intended destination, but it cannot guarantee this delivery. The main

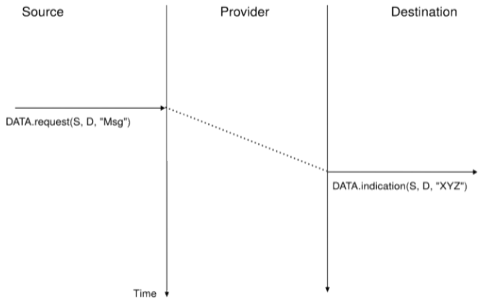


Figure 2.35: An unreliable connectionless service may deliver erroneous SDUs

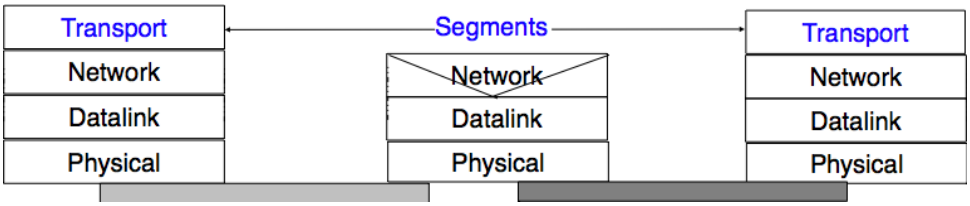


Figure 2.36: The transport layer

cause of packet losses and errors are the buffers used on the network nodes. If the buffers of one of these nodes becomes full, all arriving packets must be discarded. This situation happens frequently in practice. Transmission errors can also affect packet transmissions on links where reliable transmission techniques are not enabled or because of errors in the buffers of the network nodes.

2.4.1 Transport layer services

When two applications need to communicate, they need to structure their exchange of information. Structuring this exchange of information requires solving two different problems. The first problem is how to represent the information being exchanged knowing that the two applications may be running on hosts that use different operating systems, different processors and have different conventions to store information. This requires a common syntax to transfer the information between the two applications. For this chapter, let us assume that this syntax exists and that the two applications simply need to exchange bytes. We will discuss later how more complex data can be encoded as sequences of bytes to be exchanged. The second problem is how to organise the interactions between the application and the underlying network. From the application's viewpoint, the *network* will appear as the *transport layer* service. This *transport layer* can provide three types of services to the applications :

- the *connectionless service*
- the *connection oriented service*
- the *request-response service*

The connectionless service

The *connectionless service* that we have described earlier is frequently used by users who need to exchange small SDUs. It can be easily built on top of the connectionless network layer service that we have described earlier. Users needing to either send or receive several different and potentially large SDUs, or who need structured exchanges often prefer the *connection-oriented service*.

The connection-oriented service

An invocation of the *connection-oriented service* is divided into three phases. The first phase is the establishment of a *connection*. A *connection* is a temporary association between two users through a service provider. Several connections may exist at the same time between any pair of users. Once established, the connection is used to

transfer SDUs. *Connections* usually provide one bidirectional stream supporting the exchange of SDUs between the two users that are associated through the *connection*. This stream is used to transfer data during the second phase of the connection called the *data transfer* phase. The third phase is the termination of the connection. Once the users have finished exchanging SDUs, they request to the service provider to terminate the connection. As we will see later, there are also some cases where the service provider may need to terminate a connection itself.

The establishment of a connection can be modelled by using four primitives : *Connect.request*, *Connect.indication*, *Connect.response* and *Connect.confirm*. The *Connect.request* primitive is used to request the establishment of a connection. The main parameter of this primitive is the *address* of the destination user. The service provider delivers a *Connect.indication* primitive to inform the destination user of the connection attempt. If it accepts to establish a connection, it responds with a *Connect.response* primitive. At this point, the connection is considered to be established and the destination user can start sending SDUs over the connection. The service provider processes the *Connect.response* and will deliver a *Connect.confirm* to the user who initiated the connection. The delivery of this primitive terminates the connection establishment phase. At this point, the connection is considered to be open and both users can send SDUs. A successful connection establishment is illustrated below.

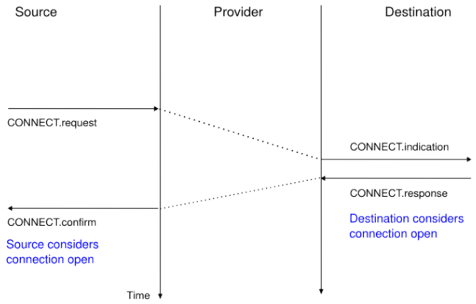


Figure 2.37: Connection establishment

The example above shows a successful connection establishment. However, in practice not all connections are successfully established. One reason is that the destination user may not agree, for policy or performance reasons, to establish a connection with the initiating user at this time. In this case, the destination user responds to the *Connect.indication* primitive by a *Disconnect.request* primitive that contains a parameter to indicate why the connection has been refused. The service provider will then deliver a *Disconnect.indication* primitive to inform the initiating user. A second reason is when the service provider is unable to reach the destination user. This might happen because the destination user is not currently attached to the network or due to congestion. In these cases, the service provider responds to the *Connect.request* with a *Disconnect.indication* primitive whose *reason* parameter contains additional information about the failure of the connection.

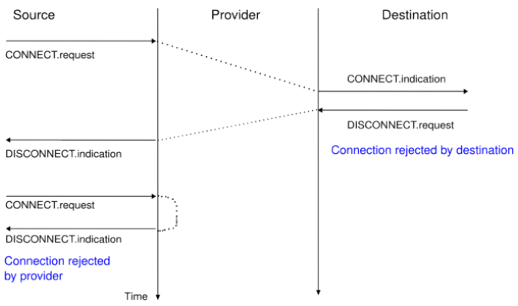


Figure 2.38: Two types of rejection for a connection establishment attempt

Once the connection has been established, the service provider supplies two data streams to the communicating users. The first data stream can be used by the initiating user to send SDUs. The second data stream allows the responding user to send SDUs to the initiating user. The data streams can be organised in different ways. A first organisation is the *message-mode* transfer. With the *message-mode* transfer, the service provider guarantees that one and only one *Data.indication* will be delivered to the endpoint of the data stream for each *Data.request* primitive issued by the other endpoint. The *message-mode* transfer is illustrated in the figure below. The main advantage of the *message-transfer* mode is that the recipient receives exactly the SDUs that were sent by the other

user. If each SDU contains a command, the receiving user can process each command as soon as it receives a SDU.

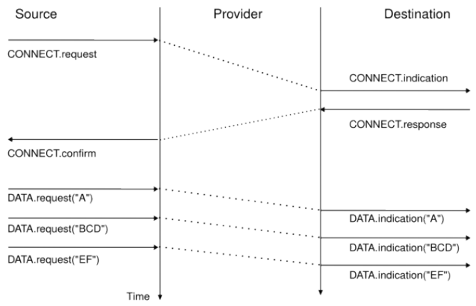


Figure 2.39: Message-mode transfer in a connection oriented service

Unfortunately, the *message-mode* transfer is not widely used on the Internet. On the Internet, the most popular connection-oriented service transfers SDUs in *stream-mode*. With the *stream-mode*, the service provider supplies a byte stream that links the two communicating users. The sending user sends bytes by using *Data.request* primitives that contain sequences of bytes as SDUs. The service provider delivers SDUs containing consecutive bytes to the receiving user by using *Data.indication* primitives. The service provider ensures that all the bytes sent at one end of the stream are delivered correctly in the same order at the other endpoint. However, the service provider does not attempt to preserve the boundaries of the SDUs. There is no relation enforced by the service provider between the number of *Data.request* and the number of *Data.indication* primitives. The *stream-mode* is illustrated in the figure below. In practice, a consequence of the utilisation of the *stream-mode* is that if the users want to exchange structured SDUs, they will need to provide the mechanisms that allow the receiving user to separate successive SDUs in the byte stream that it receives. Application layer protocols often use specific delimiters such as the end of line character to delineate SDUs in a bytestream.

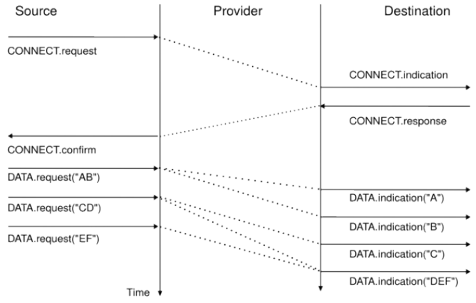


Figure 2.40: Stream-mode transfer in a connection oriented service

The third phase of a connection is when it needs to be released. As a connection involves three parties (two users and one service provider), any of them can request the termination of the connection. Usually, connections are terminated upon request of one user once the data transfer is finished. However, sometimes the service provider may be forced to terminate a connection. This can be due to lack of resources inside the service provider or because one of the users is not reachable anymore through the network. In this case, the service provider will issue *Disconnect.indication* primitives to both users. These primitives will contain, as parameter, some information about the reason for the termination of the connection. Unfortunately, as illustrated in the figure below, when a service provider is forced to terminate a connection it cannot guarantee that all SDUs sent by each user have been delivered to the other user. This connection release is said to be abrupt as it can cause losses of data.

An abrupt connection release can also be triggered by one of the users. If a user needs, for any reason, to terminate a connection quickly, it can issue a *Disconnect.request* primitive and to request an abrupt release. The service provider will process the request, stop the two data streams and deliver the *Disconnect.indication* primitive to the remote user as soon as possible. As illustrated in the figure below, this abrupt connection release may cause losses of SDUs.

To ensure a reliable delivery of the SDUs sent by each user over a connection, we need to consider the two streams that compose a connection as independent. A user should be able to release the stream that it uses to send SDUs

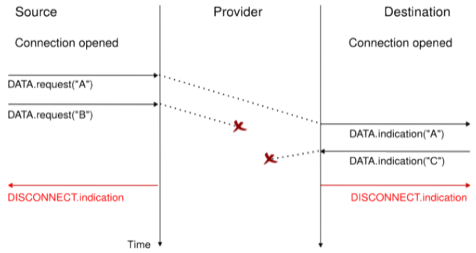


Figure 2.41: Abrupt connection release initiated by the service provider

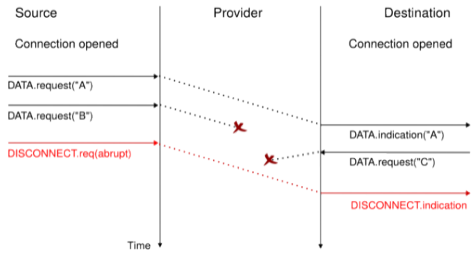


Figure 2.42: Abrupt connection release initiated by a user

once it has sent all the SDUs that it planned to send over this connection, but still continue to receive SDUs over the opposite stream. This *graceful* connection release is usually performed as shown in the figure below. One user issues a *Disconnect.request* primitive to its provider once it has issued all its *Data.request* primitives. The service provider will wait until all *Data.indication* primitives have been delivered to the receiving user before issuing the *Disconnnect.indication* primitive. This primitive informs the receiving user that it will no longer receive SDUs over this connection, but it is still able to issue *Data.request* primitives on the stream in the opposite direction. Once the user has issued all of its *Data.request* primitives, it issues a *Disconnnect.request* primitive to request the termination of the remaining stream. The service provider will process the request and deliver the corresponding *Disconnect.indication* to the other user once it has delivered all the pending *Data.indication* primitives. At this point, all data has been delivered, the two streams have been released successfully and the connection is completely closed.

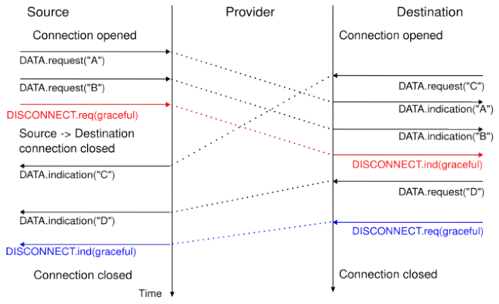


Figure 2.43: Graceful connection release

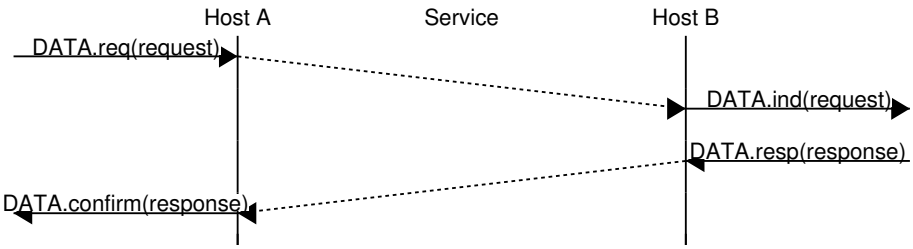
Note: Reliability of the connection-oriented service

An important point to note about the connection-oriented service is its reliability. A *connection-oriented* service can only guarantee the correct delivery of all SDUs provided that the connection has been released gracefully. This implies that while the connection is active, there is no guarantee for the actual delivery of the SDUs exchanged as the connection may need to be released abruptly at any time.

The request-response service

The *request-response service* is a compromise between the *connectionless service* and the *connection-oriented service*. Many applications need to send a small amount of data and receive a small amount of information back. This is similar to procedure calls in programming languages. A call to a procedure takes a few arguments and returns a simple answer. In a network, it is sometimes useful to execute a procedure on a different host and receive the result of the computation. Executing a procedure on another host is often called Remote Procedure Call. It is possible to use the *connectionless service* for this application. However, since this service is usually unreliable, this would force the application to deal with any type of error that could occur. Using the *connection oriented service* is another alternative. This service ensures the reliable delivery of the data, but a connection must be created before the beginning of the data transfert. This overhead can be important for applications that only exchange a small amount of data.

The *request-response service* allows to efficiently exchange small amounts of information in a request and associate it with the corresponding response. This service can be depicted by using the time-sequence diagram below.



Note: Services and layers

In the previous sections, we have described services that are provided by the transport layer. However, it is important to note that the notion of service is more general than in the transport layer. As explained earlier, the network layer also provides a service, which in most networks is an unreliable connectionless service. There are network layers that provide a connection-oriented service. Similarly, the datalink layer also provides services. Some datalink layers will provide a connectionless service. This will be the case in Local Area Networks for examples. Other datalink layers, e.g. in public networks, provide a connection oriented service.

2.4.2 The transport layer

The transport layer entity interacts with both a user in the application layer and the network layer. It improves the network layer service to make it useable by applications. From the application's viewpoint, the main limitations of the network layer service that its service is unreliable:

- the network layer may corrupt data
- the network layer may loose data
- the network layer may not deliver data in-order
- the network layer has an upper bound on maximum length of the data
- the network layer may duplicate data

To deal with these issues, the transport layer includes several mechanisms that depend on the service that it provides. It interacts with both the applications and the underlying network layer.

We have already described in the datalink layers mechanisms to deal with data losses and transmission errors. These techniques are also used in the transport layer.

Connectionless transport

The simplest service that can be provided in the transport layer is the connectionless transport service. Compared to the connectionless network layer service, this transport service includes two additional features :

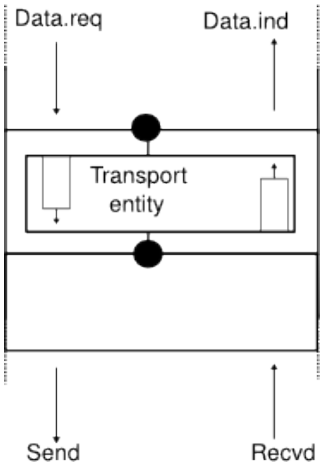


Figure 2.44: Interactions between the transport layer, its user, and its network layer provider

- an *error detection* mechanism that allows to detect corrupted data
- a *multiplexing technique* that enables several applications running on one host to exchange information with another host

To exchange data, the transport protocol encapsulates the SDU produced by its user inside a *segment*. The *segment* is the unit of transfert of information in the transport layer. Transport layer entities always exchange segments. When a transport layer entity creates a segment, this segment is encapsulated by the network layer into a packet which contains the segment as its payload and a network header. The packet is then encapsulated in a frame to be transmitted in the datalink layer.

A *segment* also contains control information, usually stored inside a *header* and the payload that comes from the application. To detect transmission errors, transport protocols rely on checksums or CRCs like the datalink layer protocols.

Compared to the connectionless network layer service, the transport layer service allows several applications running on a host to exchange SDUs with several other applications running on remote hosts. Let us consider two hosts, e.g. a client and a server. The network layer service allows the client to send information to the server, but if an application running on the client wants to contact a particular application running on the server, then an additional addressing mechanism is required other than the network layer address that identifies a host, in order to differentiate the application running on a host. This additional addressing can be provided by using *port numbers*. When a server application is launched on a host, it registers a *port number*. This *port number* will be used by the clients to contact the server process.

The figure below shows a typical usage of port numbers. The client process uses port number *1234* while the server process uses port number *5678*. When the client sends a request, it is identified as originating from port number *1234* on the client host and destined to port number *5678* on the server host. When the server process replies to this request, the server's transport layer returns the reply as originating from port *5678* on the server host and destined to port *1234* on the client host.

To support the connection-oriented service, the transport layer needs to include several mechanisms to enrich the connectionless network-layer service. We discuss these mechanisms in the following sections.

Connection establishment

Like the connectionless service, the connection-oriented service allows several applications running on a given host to exchange data with other hosts. The port numbers described above for the connectionless service are also used by the connection-oriented service to multiplex several applications. Similarly, connection-oriented protocols used checksums/CRCs to detect transmission errors and discard segments containing an invalid checksum/CRC.

An important difference between the connectionless service and the connection-oriented one is that the transport

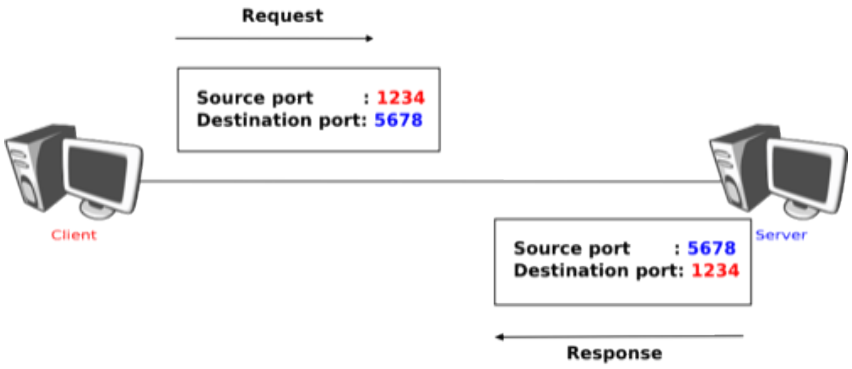


Figure 2.45: Utilisation of port numbers

entities in the latter maintain some state during lifetime of the connection. This state is created when a connection is established and is removed when it is released.

The simplest approach to establish a transport connection would be to define two special control segments : *CR* and *CA*. The *CR* segment is sent by the transport entity that wishes to initiate a connection. If the remote entity wishes to accept the connection, it replies by sending a *CA* segment. The *CR* and *CA* segments contain *port numbers* that allow to identify the communicating applications. The transport connection is considered to be established once the *CA* segment has been received. At that point, data segments can be sent in both directions.

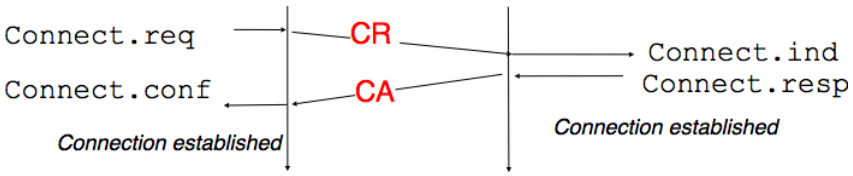


Figure 2.46: Naive transport connection establishment

Unfortunately, this scheme is not sufficient given the unreliable network layer. Since the network layer is imperfect, the *CR* or *CA* segments can be lost, delayed, or suffer from transmission errors. To deal with these problems, the control segments must be protected by using a CRC or checksum to detect transmission errors. Furthermore, since the *CA* segment acknowledges the reception of the *CR* segment, the *CR* segment can be protected by using a retransmission timer.

Unfortunately, this scheme is not sufficient to ensure the reliability of the transport service. Consider for example a short-lived transport connection where a single, but important transfer (e.g. money transfer from a bank account) is sent. Such a short-lived connection starts with a *CR* segment acknowledged by a *CA* segment, then the data segment is sent, acknowledged and the connection terminates. Unfortunately, as the network layer service is unreliable, delays combined to retransmissions may lead to the situation depicted in the figure below, where a delayed *CR* and data segments from a former connection are accepted by the receiving entity as valid segments, and the corresponding data is delivered to the user. Duplicating SDUs is not acceptable, and the transport protocol must solve this problem.

To avoid these duplicates, transport protocols require the network layer to bound the *Maximum Segment Lifetime (MSL)*. The organisation of the network must guarantee that no segment remains in the network for longer than *MSL* seconds. For example, on today's Internet, *MSL* is expected to be 2 minutes. To avoid duplicate transport connections, transport protocol entities must be able to safely distinguish between a duplicate *CR* segment and a new *CR* segment, without forcing each transport entity to remember all the transport connections that it has established in the past.

A classical solution to avoid remembering the previous transport connections to detect duplicates is to use a clock inside each transport entity. This *transport clock* has the following characteristics :

- the *transport clock* is implemented as a k bits counter and its clock cycle is such that $2^k \times cycle >> MSL$. Furthermore, the *transport clock* counter is incremented every clock cycle and after each connection establishment. This clock is illustrated in the figure below.

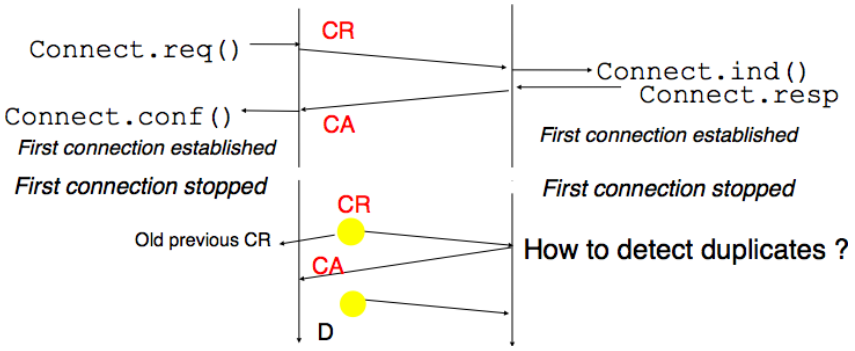


Figure 2.47: Duplicate transport connections ?

- the *transport clock* must continue to be incremented even if the transport entity stops or reboots

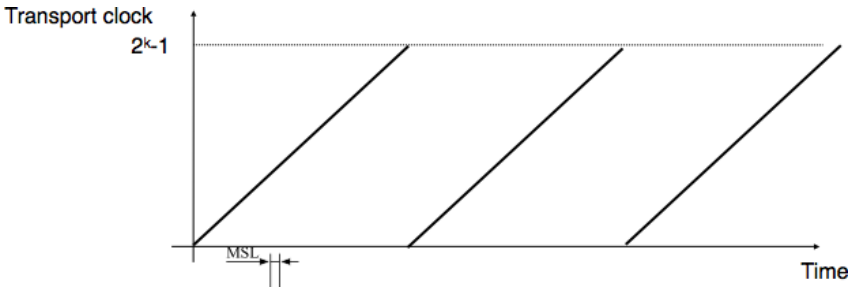


Figure 2.48: Transport clock

It should be noted that *transport clocks* do not need and usually are not synchronised to the real-time clock. Precisely synchronising real-time clocks is an interesting problem, but it is outside the scope of this document. See [Mills2006] for a detailed discussion on synchronizing the real-time clock.

This *transport clock* can now be combined with an exchange of three segments, called the *three way handshake*, to detect duplicates. This *three way handshake* occurs as follows :

1. The initiating transport entity sends a *CR* segment. This segment requests the establishment of a transport connection. It contains a port number (not shown in the figure) and a sequence number (*seq=x* in the figure below) whose value is extracted from the *transport clock*. The transmission of the *CR* segment is protected by a retransmission timer.
2. The remote transport entity processes the *CR* segment and creates state for the connection attempt. At this stage, the remote entity does not yet know whether this is a new connection attempt or a duplicate segment. It returns a *CA* segment that contains an acknowledgement number to confirm the reception of the *CR* segment (*ack=x* in the figure below) and a sequence number (*seq=y* in the figure below) whose value is extracted from its transport clock. At this stage, the connection is not yet established.
3. The initiating entity receives the *CA* segment. The acknowledgement number of this segment confirms that the remote entity has correctly received the *CR* segment. The transport connection is considered to be established by the initiating entity and the numbering of the data segments starts at sequence number *x*. Before sending data segments, the initiating entity must acknowledge the received *CA* segments by sending another *CA* segment.
4. The remote entity considers the transport connection to be established after having received the segment that acknowledges its *CA* segment. The numbering of the data segments sent by the remote entity starts at sequence number *y*.

The three way handshake is illustrated in the figure below.

Thanks to the three way handshake, transport entities avoid duplicate transport connections. This is illustrated by considering the three scenarios below.

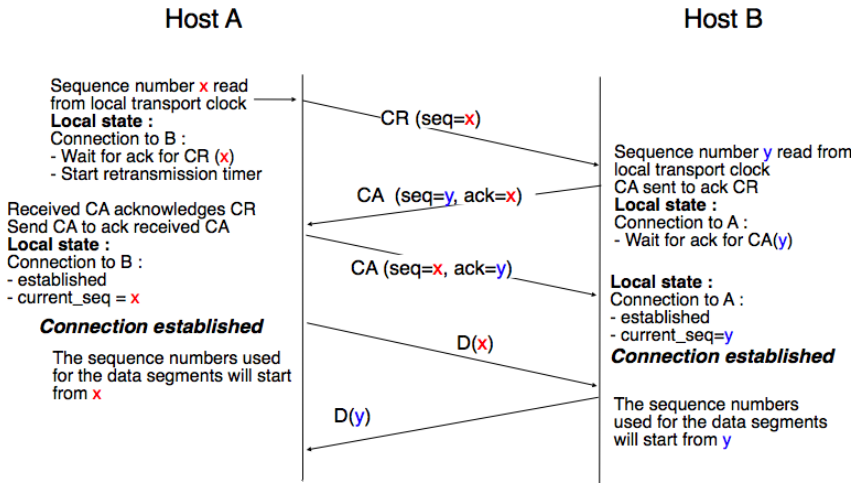


Figure 2.49: Three-way handshake

The first scenario is when the remote entity receives an old *CR* segment. It considers this *CR* segment as a connection establishment attempt and replies by sending a *CA* segment. However, the initiating host cannot match the received *CA* segment with a previous connection attempt. It sends a control segment (*REJECT* in the figure below) to cancel the spurious connection attempt. The remote entity cancels the connection attempt upon reception of this control segment.

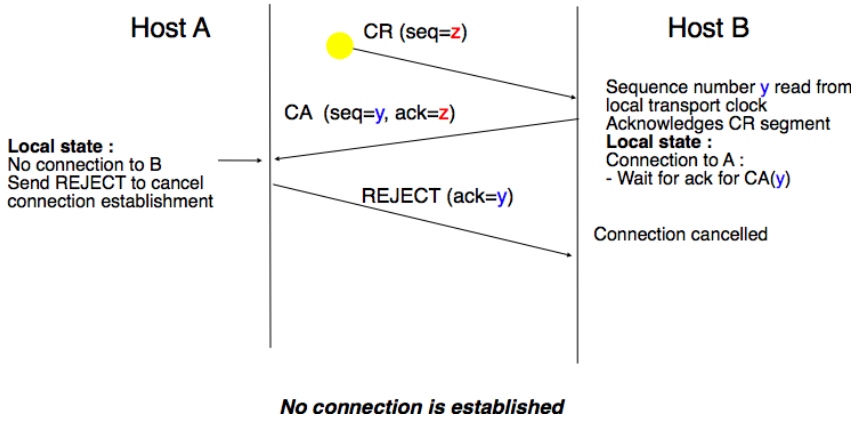


Figure 2.50: Three-way handshake : recovery from a duplicate *CR*

A second scenario is when the initiating entity sends a *CR* segment that does not reach the remote entity and receives a duplicate *CA* segment from a previous connection attempt. This duplicate *CA* segment cannot contain a valid acknowledgement for the *CR* segment as the sequence number of the *CR* segment was extracted from the transport clock of the initiating entity. The *CA* segment is thus rejected and the *CR* segment is retransmitted upon expiration of the retransmission timer.

The last scenario is less likely, but it it important to consider it as well. The remote entity receives an old *CR* segment. It notes the connection attempt and acknowledges it by sending a *CA* segment. The initiating entity does not have a matching connection attempt and replies by sending a *REJECT*. Unfortunately, this segment never reaches the remote entity. Instead, the remote entity receives a retransmission of an older *CA* segment that contains the same sequence number as the first *CR* segment. This *CA* segment cannot be accepted by the remote entity as a confirmation of the transport connection as its acknowledgement number cannot have the same value as the sequence number of the first *CA* segment.

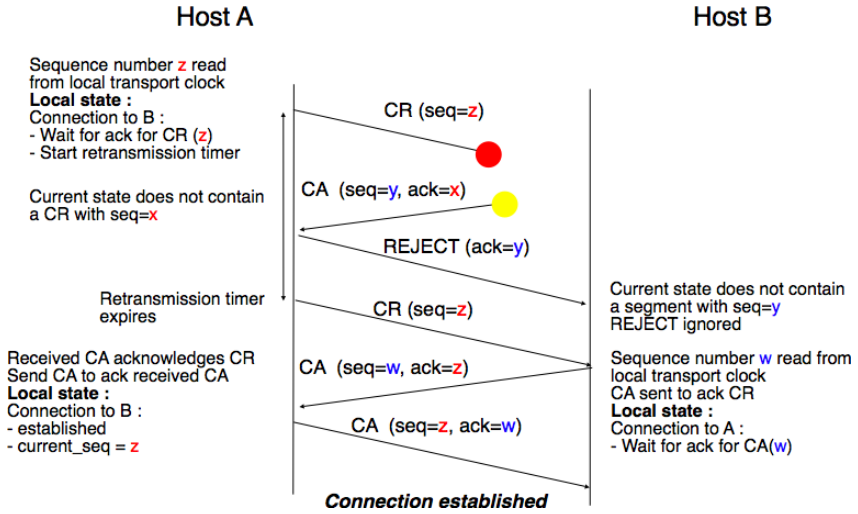


Figure 2.51: Three-way handshake : recovery from a duplicate *CA*

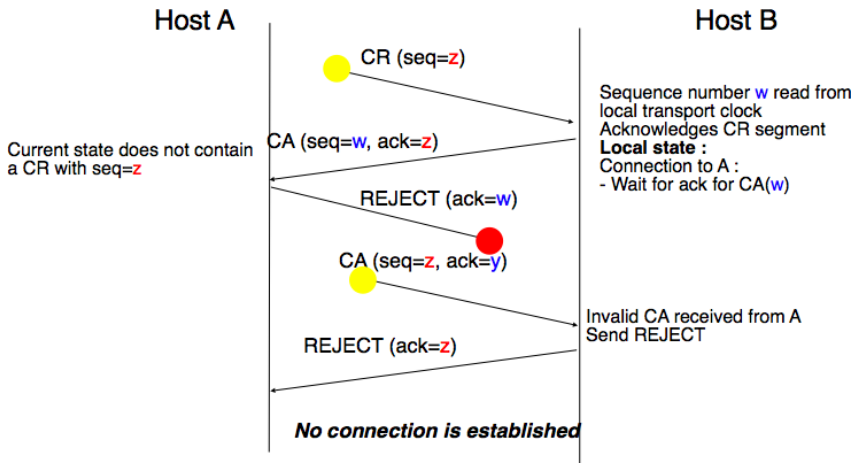


Figure 2.52: Three-way handshake : recovery from duplicates *CR* and *CA*

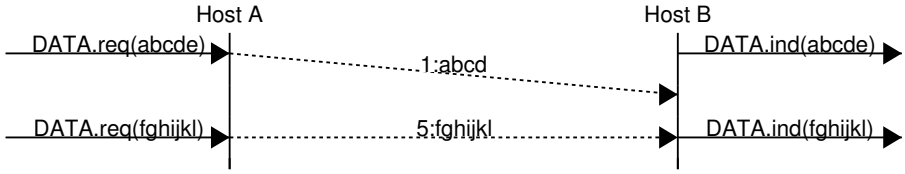
Data transfer

Now that the transport connection has been established, it can be used to transfer data. To ensure a reliable delivery of the data, the transport protocol will include sliding windows, retransmission timers and *go-back-n* or *selective repeat*. However, we cannot simply reuse the techniques from the datalink because a transport protocol needs to deal with more types of errors than a reliable protocol in datalink layer. The first difference between the two layers is the transport layer must face with more variable delays. In the datalink layer, when two hosts are connected by a link, the transmission delay or the round-trip-time over the link is almost fixed. In a network that can span the globe, the delays and the round-trip-times can vary significantly on a per packet basis. This variability can be caused by two factors. First, packets sent through a network do not necessarily follow the same path to reach their destination. Second, some packets may be queued in the buffers of routers when the load is high and these queueing delays can lead to increased end-to-end delays. A second difference between the datalink layer and the transport layer is that a network does not always deliver packets in sequence. This implies that packets may be reordered by the network. Furthermore, the network may sometimes duplicate packets. The last issue that needs to be dealt with in the transport layer is the transmission of large SDUs. In the datalink layer, reliable protocols transmit small frames. Applications could generate SDUs that are much larger than the maximum size of a packet in the network layer. The transport layer needs to include mechanisms to fragment and reassemble these large SDUs.

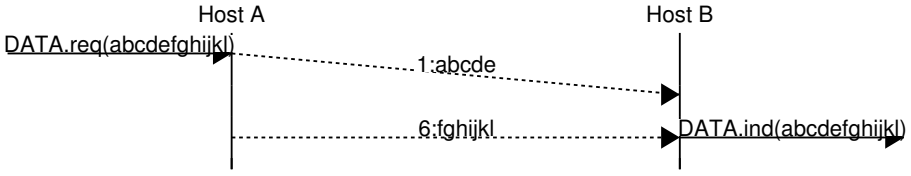
To deal with all these characteristics of the network layer, we need to adapt the techniques that we have introduced in the datalink layer.

The first point which is common between the two layers is that both use CRCs or checksum to detect transmission errors. Each segment contains a CRC/checksum which is computed over the entire segment (header and payload) by the sender and inserted in the header. The receiver recomputes the CRC/checksum for each received segment and discards all segments with an invalid CRC.

Reliable transport protocols also use sequence numbers and acknowledgement numbers. While reliable protocols in the datalink layer use one sequence number per frame, reliable transport protocols consider all the data transmitted as a stream of bytes. In these protocols, the sequence number placed in the segment header corresponds to the position of the first byte of the payload in the bytestream. This sequence number allows to detect losses but also enables the receiver to reorder the out-of-sequence segments. This is illustrated in the figure below.



Using sequence numbers to count bytes has also one advantage when the transport layer needs to fragment SDUs in several segments. The figure below shows the fragmentation of a large SDU in two segments. Upon reception of the segments, the receiver will use the sequence numbers to correctly reorder the data.



Compared to reliable protocols in the datalink layer, reliable transport protocols encode their sequence numbers in more bits. 32 bits and 64 bits sequence numbers are frequent in the transport layer while some datalink layer protocols encode their sequence numbers in an 8 bits field. This large sequence number space is motivated by two reasons. First, since the sequence number is incremented for each transmitted byte, a single segment may consume one or several thousands of sequence numbers. Second, a reliable transport protocol must be able to detect delayed segments. This can only be done if the number of bytes transmitted during the MSL period is smaller than the sequence number space. Otherwise, there is a risk of accepting duplicate segments.

Go-back-n and *selective repeat* can be used in the transport layer as in the datalink layer. Since the network layer does not guarantee an in-order delivery of the packets, a transport entity should always store the segments that it receives out-of-sequence. For this reason, most transport protocols will opt for some form of selective repeat

mechanism.

In the datalink layer, the sliding window has usually a fixed size which depends on the amount of buffers allocated to the datalink layer entity. Such a datalink layer entity usually serves one or a few network layer entities. In the transport layer, the situation is different. A single transport layer entity serves a large and varying number of application processes. Each transport layer entity manages a pool of buffers that needs to be shared between all these processes. Transport entity are usually implemented inside the operating system kernel and shares memory with other parts of the system. Furthermore, a transport layer entity must support several (possibly hundreds or thousands) of transport connections at the same time. This implies that the memory which can be used to support the sending or the receiving buffer of a transport connection may change during the lifetime of the connection [5] . Thus, a transport protocol must allow the sender and the receiver to adjust their window sizes.

To deal with this issue, transport protocols allow the receiver to advertise the current size of its receiving window in all the acknowledgements that it sends. The receiving window advertised by the receiver bounds the size of the sending buffer used by the sender. In practice, the sender maintains two state variables : *swin*, the size of its sending window (that may be adjusted by the system) and *rwin*, the size of the receiving window advertised by the receiver. At any time, the number of unacknowledged segments cannot be larger than *min(swin,rwin)* [6] . The utilisation of dynamic windows is illustrated in the figure below.

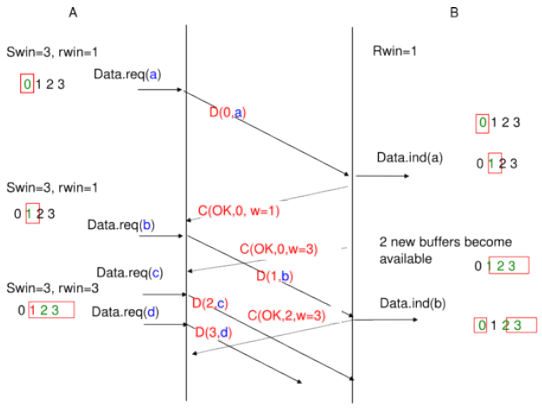


Figure 2.53: Dynamic receiving window

The receiver may adjust its advertised receive window based on its current memory consumption, but also to limit the bandwidth used by the sender. In practice, the receive buffer can also shrink as the application may not able to process the received data quickly enough. In this case, the receive buffer may be completely full and the advertised receive window may shrink to *0*. When the sender receives an acknowledgement with a receive window set to *0*, it is blocked until it receives an acknowledgement with a positive receive window. Unfortunately, as shown in the figure below, the loss of this acknowledgement could cause a deadlock as the sender waits for an acknowledgement while the receiver is waiting for a data segment.

To solve this problem, transport protocols rely on a special timer : the *persistence timer*. This timer is started by the sender whenever it receives an acknowledgement advertising a receive window set to *0*. When the timer expires, the sender retransmits an old segment in order to force the receiver to send a new acknowledgement, and hence send the current receive window size.

To conclude our description of the basic mechanisms found in transport protocols, we still need to discuss the impact of segments arriving in the wrong order. If two consecutive segments are reordered, the receiver relies on their sequence numbers to reorder them in its receive buffer. Unfortunately, as transport protocols reuse the same sequence number for different segments, if a segment is delayed for a prolonged period of time, it might still be accepted by the receiver. This is illustrated in the figure below where segment *D(1,b)* is delayed.

To deal with this problem, transport protocols combine two solutions. First, they use 32 bits or more to encode the sequence number in the segment header. This increases the overhead, but also increases the delay between the transmission of two different segments having the same sequence number. Second, transport protocols require the network layer to enforce a *Maximum Segment Lifetime (MSL)*. The network layer must ensure that no packet

[5] For a discussion on how the sending buffer can change, see e.g. [SMM1998]

[6] Note that if the receive window shrinks, it might happen that the sender has already sent a segment that is not anymore inside its window. This segment will be discarded by the receiver and the sender will retransmit it later.

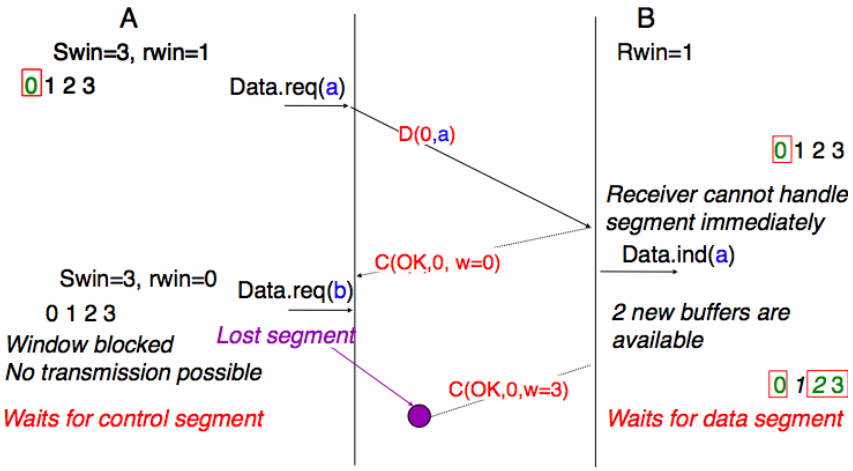


Figure 2.54: Risk of deadlock with dynamic windows

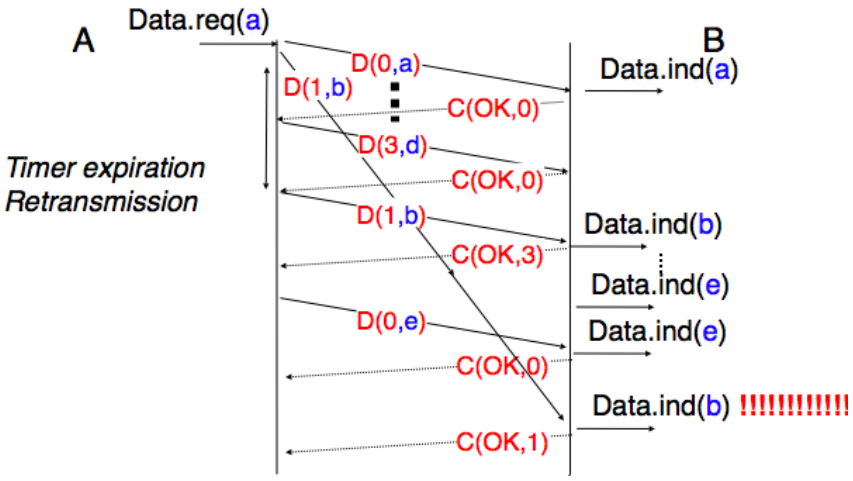


Figure 2.55: Ambiguities caused by excessive delays

remains in the network for more than MSL seconds. In the Internet the MSL is assumed [7] to be 2 minutes **RFC 793**. Note that this limits the maximum bandwidth of a transport protocol. If it uses *n* bits to encode its sequence numbers, then it cannot send more than 2^n segments every MSL seconds.

Connection release

When we discussed the connection-oriented service, we mentioned that there are two types of connection releases : *abrupt release* and *graceful release*.

The first solution to release a transport connection is to define a new control segment (e.g. the *DR* segment) and consider the connection to be released once this segment has been sent or received. This is illustrated in the figure below.

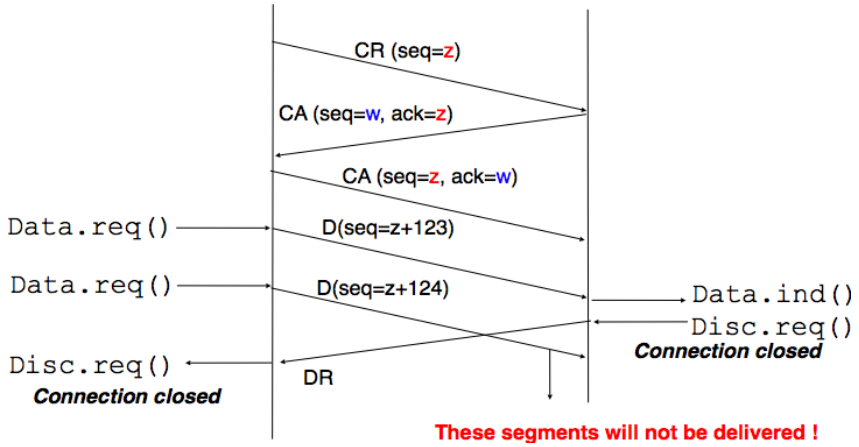


Figure 2.56: Abrupt connection release

As the entity that sends the *DR* segment cannot know whether the other entity has already sent all its data on the connection, SDUs can be lost during such an *abrupt connection release*.

The second method to release a transport connection is to release independently the two directions of data transfer. Once a user of the transport service has sent all its SDUs, it performs a *DISCONNECT.req* for its direction of data transfer. The transport entity sends a control segment to request the release of the connection *after* the delivery of all previous SDUs to the remote user. This is usually done by placing in the *DR* the next sequence number and by delivering the *DISCONNECT.ind* only after all previous *DATA.ind*. The remote entity confirms the reception of the *DR* segment and the release of the corresponding direction of data transfer by returning an acknowledgement. This is illustrated in the figure below.

2.5 Naming and addressing

The network and the transport layers rely on addresses that are encoded as fixed size bit strings. A network layer uniquely identifies a host. Several transport layer entities can use the service of the same network layer. For example, a reliable transport protocol and a connectionless transport protocol can coexist on the same host. In this case, the network layer multiplexes the segments produced by the two protocols. This multiplexing is usually achieved by placing in the network packet header a field that indicates which transport protocol produced and should process the segment. Given that there are few different transport protocols, this field does not need to be long. The port numbers play a similar role in the transport layer since they enable it to multiplex data from several application processes.

While addresses are natural for the network and transport layer entities, human users prefer to use names when interacting with servers. Names can be encoded as a character string and a mapping services allows applications

[7] In reality, the Internet does not strictly enforce this MSL. However, it is reasonable to expect that most packets on the Internet will not remain in the network during more than 2 minutes. There are a few exceptions to this rule, such as **RFC 1149** whose implementation is described in http://www.blug.linux.no/rfc1149/ but there are few real links supporting **RFC 1149** in the Internet.

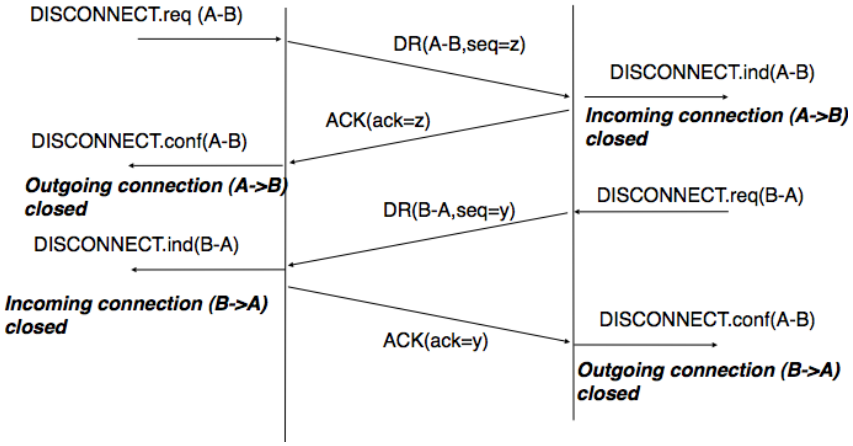


Figure 2.57: Graceful connection release

to map a name into the corresponding address. Using names is friendlier for the human users than addresses, but it also provides a level of indirection which is very useful in various situations. Before looking at these benefits of names, it is useful to discuss how names are used on the Internet.

In the early days of the Internet, there were only a few number of hosts (mainly minicomputers) connected to the network. The most popular applications were remote login and file transfer. By 1983, there were already five hundred hosts attached to the Internet. Each of these hosts were identified by a unique IPv4 address. Forcing human users to remember the IPv4 addresses of the remote hosts that they want to use was not user-friendly. Human users prefer to remember names, and use them when needed. Using names as aliases for addresses is a common technique in Computer Science. It simplifies the development of applications and allows the developer to ignore the low level details. For example, by using a programming language instead of writing machine code, a developer can write software without knowing whether the variables that it uses are stored in memory or inside registers.

Because names are at a higher level than addresses, they allow (both in the example of programming above, and on the Internet) to treat addresses as mere technical identifiers, which can change at will. Only the names are stable.

The first solution that allowed applications to use names was the *hosts.txt* file. This file is similar to the symbol table found in compiled code. It contains the mapping between the name of each Internet host and its associated IP address [8]. It was maintained by SRI International that coordinated the Network Information Center (NIC). When a new host was connected to the network, the system administrator had to register its name and IP address at the NIC. The NIC updated the *hosts.txt* file on its server. All Internet hosts regularly retrieved the updated *hosts.txt* file from the server maintained by SRI. This file was stored at a well-known location on each Internet host (see **RFC 952**) and networked applications could use it to find the IP address corresponding to a name.

A *hosts.txt* file can be used when there are up to a few hundred hosts on the network. However, it is clearly not suitable for a network containing thousands or millions of hosts. A key issue in a large network is to define a suitable naming scheme. The ARPANet initially used a flat naming space, i.e. each host was assigned a unique name. To limit collisions between names, these names usually contained the name of the institution and a suffix to identify the host inside the institution (a kind of poor man's hierarchical naming scheme). On the ARPANet few institutions had several hosts connected to the network.

However, the limitations of a flat naming scheme became clear before the end of the ARPANet and **RFC 819** proposed a hierarchical naming scheme. While **RFC 819** discussed the possibility of organising the names as a directed graph, the Internet opted eventually for a tree structure capable of containing all names. In this tree, the top-level domains are those that are directly attached to the root. The first top-level domain was *.arpa* [9]. This top-level name was initially added as a suffix to the names of the hosts attached to the ARPANet and listed in the *hosts.txt* file. In 1984, the *.gov*, *.edu*, *.com*, *.mil* and *.org* generic top-level domain names were added and **RFC 1032** proposed the utilisation of the two letter *ISO-3166* country codes as top-level domain names. Since

[8] The *hosts.txt* file is not maintained anymore. A historical snapshot retrieved on April 15th, 1984 is available from http://ftp.univie.ac.at/netinfo/netinfo/hosts.txt

[9] See http://www.donelan.com/dnstimeline.html for a time line of DNS related developments.

ISO-3166 defines a two letter code for each country recognised by the United Nations, this allowed all countries to automatically have a top-level domain. These domains include *.be* for Belgium, *.fr* for France, *.us* for the USA, *.ie* for Ireland or *.tv* for Tuvalu, a group of small islands in the Pacific and *.tm* for Turkmenistan. Today, the set of top-level domain-names is managed by the Internet Corporation for Assigned Names and Numbers (*ICANN*). Recently, *ICANN* added a dozen of generic top-level domains that are not related to a country and the *.cat* top-level domain has been registered for the Catalan language. There are ongoing discussions within *ICANN* to increase the number of top-level domains.

Each top-level domain is managed by an organisation that decides how sub-domain names can be registered. Most top-level domain names use a first-come first served system, and allow anyone to register domain names, but there are some exceptions. For example, *.gov* is reserved for the US government, *.int* is reserved for international organisations and names in the *.ca* are mainly reserved for companies or users who are present in Canada.

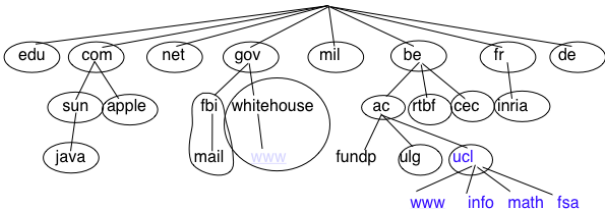


Figure 2.58: The tree of domain names

RFC 1035 recommended the following *BNF* for fully qualified domain names, to allow host names with a syntax which works with all applications (the domain names themselves have a much richer syntax).

```
domain ::= subdomain | " "
subdomain ::= label | subdomain "." label
label ::= letter [ [ ldh-str ] let-dig ]
ldh-str ::= let-dig-hyp | let-dig-hyp ldh-str
let-dig-hyp ::= let-dig | "-"
let-dig ::= letter | digit
letter ::= any one of the 52 alphabetic characters A through Z in upper case and a through z in lower case
digit ::= any one of the ten digits 0 through 9
```

Figure 2.59: BNF of the fully qualified host names

This grammar specifies that a host name is an ordered list of labels separated by the dot (.) character. Each label can contain letters, numbers and the hyphen character (-) [10]. Fully qualified domain names are read from left to right. The first label is a hostname or a domain name followed by the hierarchy of domains and ending with the root implicitly at the right. The top-level domain name must be one of the registered TLDs [11]. For example, in the above figure, *www.whitehouse.gov* corresponds to a host named *www* inside the *whitehouse* domain that belongs to the *gov* top-level domain. *info.ucl.ac.be* corresponds to the *info* domain inside the *ucl* domain that is included in the *ac* sub-domain of the *be* top-level domain.

This hierarchical naming scheme is a key component of the Domain Name System (DNS). The DNS is a distributed database that contains mappings between fully qualified domain names and IP addresses. The DNS uses the client-server model. The clients are hosts that need to retrieve the mapping for a given name. Each *nameserver* stores part of the distributed database and answers the queries sent by clients. There is at least one *nameserver* that is responsible for each domain. In the figure below, domains are represented by circles and there are three hosts inside domain *dom* (*h1*, *h2* and *h3*) and three hosts inside domain *a.sdom1.dom*. As shown in the figure below, a sub-domain may contain both host names and sub-domains.

A *nameserver* that is responsible for domain *dom* can directly answer the following queries :

- the IP address of any host residing directly inside domain *dom* (e.g. *h2.dom* in the figure above)
- the nameserver(s) that are responsible for any direct sub-domain of domain *dom* (i.e. *sdom1.dom* and *sdom2.dom* in the figure above, but not *z.sdom1.dom*)

[10] This specification evolved later to support domain names written by using other character sets than us-ASCII **RFC 5890**. This extension is important to support languages other than English, but a detailed discussion is outside the scope of this document.

[11] The official list of top-level domain names is maintained by *IANA* at http://data.iana.org/TLD/tlds-alpha-by-domain.txt Additional information about these domains may be found at http://en.wikipedia.org/wiki/List_of_Internet_top-level_domains

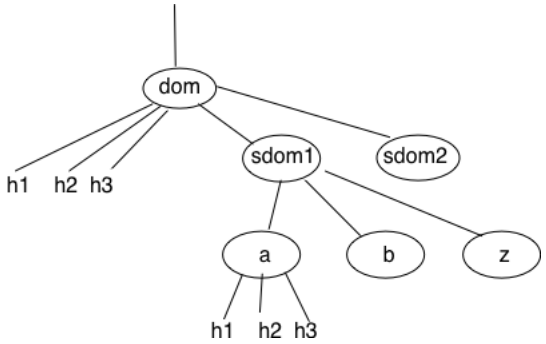


Figure 2.60: A simple tree of domain names

To retrieve the mapping for host *h2.dom*, a client sends its query to the name server that is responsible for domain *.dom*. The name server directly answers the query. To retrieve a mapping for *h3.a.sdom1.dom* a DNS client first sends a query to the name server that is responsible for the *.dom* domain. This nameserver returns the nameserver that is responsible for the *sdom1.dom* domain. This nameserver can now be contacted to obtain the nameserver that is responsible for the *a.sdom1.dom* domain. This nameserver can be contacted to retrieve the mapping for the *h3.a.sdom1.dom* name. Thanks to this organisation of the nameservers, it is possible for a DNS client to obtain the mapping of any host inside the *.dom* domain or any of its subdomains. To ensure that any DNS client will be able to resolve any fully qualified domain name, there are special nameservers that are responsible for the root of the domain name hierarchy. These nameservers are called *root nameserver*. There are currently about a dozen root nameservers.

Each root nameserver maintains the list [12] of all the nameservers that are responsible for each of the top-level domain names and their IP addresses [13]. All root nameservers are synchronised and provide the same answers. By querying any of the root nameservers, a DNS client can obtain the nameserver that is responsible for any top-level-domain name. From this nameserver, it is possible to resolve any domain name.

To be able to contact the root nameservers, each DNS client must know their IP addresses. This implies, that DNS clients must maintain an up-to-date list of the IP addresses of the root nameservers. Without this list, it is impossible to contact the root nameservers. Forcing all Internet hosts to maintain the most recent version of this list would be difficult from an operational point of view. To solve this problem, the designers of the DNS introduced a special type of DNS server : the DNS resolvers. A *resolver* is a server that provides the name resolution service for a set of clients. A network usually contains a few resolvers. Each host in these networks is configured to send all its DNS queries via one of its local resolvers. These queries are called *recursive queries* as the *resolver* must recurse through the hierarchy of nameservers to obtain the *answer*.

DNS resolvers have several advantages over letting each Internet host query directly nameservers. Firstly, regular Internet hosts do not need to maintain the up-to-date list of the IP addresses of the root servers. Secondly, regular Internet hosts do not need to send queries to nameservers all over the Internet. Furthermore, as a DNS resolver serves a large number of hosts, it can cache the received answers. This allows the resolver to quickly return answers for popular DNS queries and reduces the load on all DNS servers [JSBM2002].

2.5.1 Benefits of names

Using names instead of addresses inside applications has several important benefits in addition to being more human friendly. To understand these benefits, let us consider a popular application that provides information stored on servers. This application involves clients and servers. The server processes provide information upon requests from client processes running on remote hosts. A first deployment of this application would be to rely only on addresses. In this case, the server process would be installed on one host and the clients would connect to this server to retrieve information. Such a deployment has several drawbacks :

[12] A copy of the information maintained by each root nameserver is available at http://www.internic.net/zones/root.zone

[13] Until February 2008, the root DNS servers only had IPv4 addresses. IPv6 addresses were added to the root DNS servers slowly to avoid creating problems as discussed in http://www.icann.org/en/committees/security/sac018.pdf In 2013, several DNS root servers are still not reachable by using IPv6. The full list is available at http://www.root-servers.org/

- if the server process moves to another physical server, all clients must be informed about the new server address
- if there are many concurrent clients, the load of the server will increase without any possibility of adding another server without changing the server addresses user by the clients

Using names solves these problems and provide additional benefits. If clients are configured with the name of the server, they will query the name service before connecting to the server. The name service will resolve the name into the corresponding address. If a server process needs to move from one physical server to another, it suffices to update the name to address mapping of the name service to allow all clients to connect to the new server. The name service also enables the servers to better sustain be load. Assume a very popular server which is accessed by millions of user. This service cannot be provided by a single physical server due to performance limitations. Thanks to the utilisation of names, it is possible to scale this service by mapping a given name to a set of addresses. When a client queries the name service for the server's name, the name service returns one of the addresses in the set. Various strategies can be used to select one particular address inside the set of addresses. A first strategy is to select a random address in the set. A second strategy is to maintain information about the load on the servers and return the address of the less loaded server. Note that the list of server addresses does not need to remain fixed. It is possible to add and remove addresses from the list to cope with load fluctuations. Another strategy is to infer the location of the client from the name request and return the address of the closest server.

Mapping a single name onto a set of addresses allow popular servers to scale dynamically. There are also benefits in mapping multiple names, possibly a large number of them, onto a single address. Consider the case of information servers run by individuals or SMEs. Some of these servers attract only a few clients per day. Using a single physical server for each of these services would be a waste of resources. A better approach is to use a single server for a set of services that are all identified by different names. This enables service providers to support a large number of servers, identifiied by different names, onto a single physical server. If one of these servers becomes very popular, it will be possible to map its name onto a set of addresses to be able to sustain the load. There are some deployments where this mapping is done dynamically in function of the load.

Names provide a lot of flexibility compared to addresses. For the network, they play a similar role as variables in programming languages. No programmer using a high-level programming language would consider using addresses instead of variables. For the same reasons, all networked applications should depend on names and avoid dealing with addresses as much as possible.

2.6 Sharing resources

A network is designed to support a potentially large number of users that exchange information with each other. These users produce and consume information which is exchanged through the network. To support its users, a network uses several types of resources. It is important to keep in mind the different resources that are shared inside the network.

The first and more important resource inside a network is the link bandwidth. There are two situations where link bandwidth needs to be shared between different users. The first situation is when several hosts are attached to the same physical link. This situation mainly occurs in Local Area Networks (LAN). A LAN is a network that efficiently interconnects several hosts (usually a few dozens to a few hundreds) in the same room, building or campus. Consider for a example a network with five hosts. Any of these hosts needs to be able to exchange information with any of the other five hosts. A first organisation for this LAN is the full-mesh.

The full-mesh is the most reliable and highest performing network to interconnect these five hosts. However, this network organisation has two important drawbacks. First, if a network contains *n* hosts, then $\frac{n \times (n-1)}{2}$ links are required. If the network contains more than a few hosts, it becomes impossible to lay down the required physical links. Second, if the network contains *n* hosts, then each host must have $n-1$ interfaces to terminante $n-1$ links. This is beyond the capabilities of most hosts. Furthermore, if a new host is added to the network, new links have to be laid down and one interface has to be added to each participating host. However, full-mesh has the advantage of providing the lowest delay between the hosts and the best resiliency against link failures. In practice, full-mesh networks are rarely used expected when there are few network nodes and resiliency is key.

The second possible physical organisation, which is also used inside computers to connect different extension cards, is the bus. In a bus network, all hosts are attached to a shared medium, usually a cable through a single interface. When one host sends an electrical signal on the bus, the signal is received by all hosts attached to the bus.

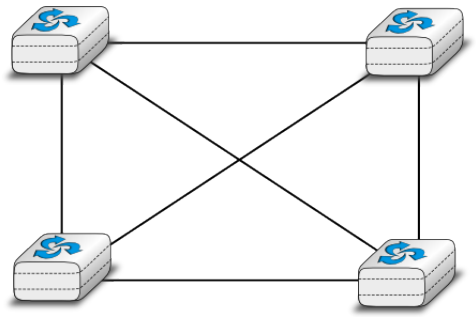


Figure 2.61: A Full mesh network

A drawback of bus-based networks is that if the bus is physically cut, then the network is split into two isolated networks. For this reason, bus-based networks are sometimes considered to be difficult to operate and maintain, especially when the cable is long and there are many places where it can break. Such a bus-based topology was used in early Ethernet networks.

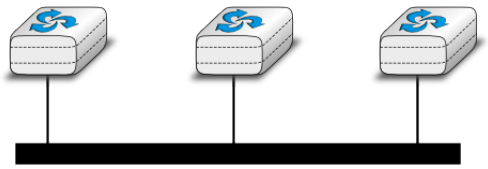


Figure 2.62: A network organized as a Bus

A third organisation of a computer network is a star topology. In such topologies, hosts have a single physical interface and there is one physical link between each host and the center of the star. The node at the center of the star can be either a piece of equipment that amplifies an electrical signal, or an active device, such as a piece of equipment that understands the format of the messages exchanged through the network. Of course, the failure of the central node implies the failure of the network. However, if one physical link fails (e.g. because the cable has been cut), then only one node is disconnected from the network. In practice, star-shaped networks are easier to operate and maintain than bus-shaped networks. Many network administrators also appreciate the fact that they can control the network from a central point. Administered from a Web interface, or through a console-like connection, the center of the star is a useful point of control (enabling or disabling devices) and an excellent observation point (usage statistics).

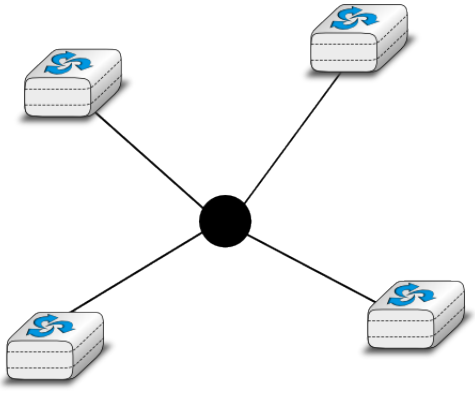


Figure 2.63: A network organised as a Star

A fourth physical organisation of a network is the ring topology. Like the bus organisation, each host has a single physical interface connecting it to the ring. Any signal sent by a host on the ring will be received by all hosts attached to the ring. From a redundancy point of view, a single ring is not the best solution, as the signal only travels in one direction on the ring; thus if one of the links composing the ring is cut, the entire network fails. In practice, such rings have been used in local area networks, but are now often replaced by star-shaped networks.

In metropolitan networks, rings are often used to interconnect multiple locations. In this case, two parallel links, composed of different cables, are often used for redundancy. With such a dual ring, when one ring fails all the traffic can be quickly switched to the other ring.

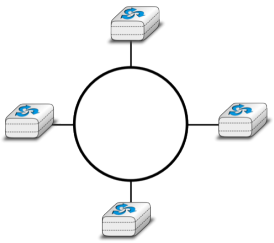


Figure 2.64: A network organised as a ring

A fifth physical organisation of a network is the tree. Such networks are typically used when a large number of customers must be connected in a very cost-effective manner. Cable TV networks are often organised as trees.

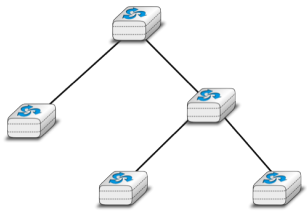


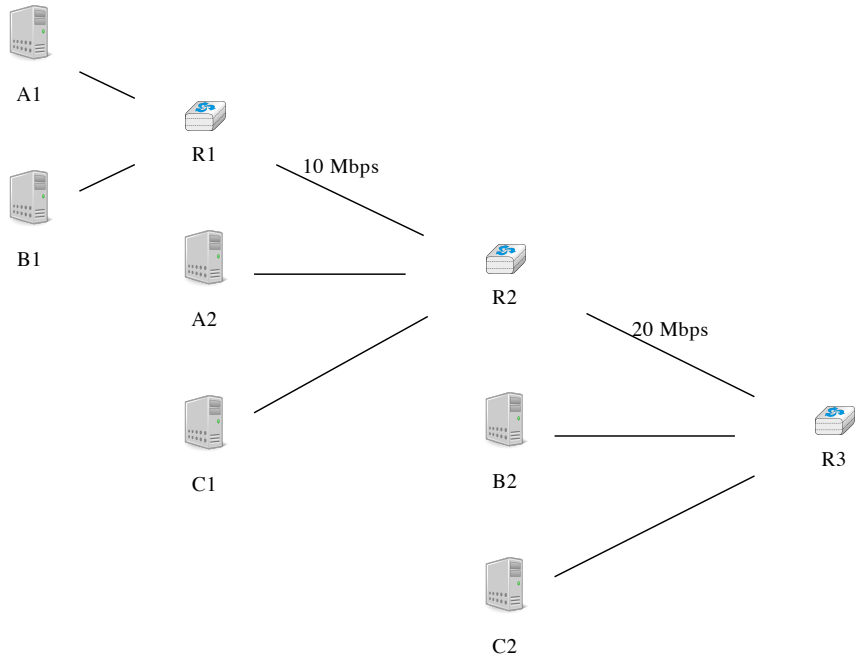
Figure 2.65: A network organized as a Tree

2.6.1 Sharing bandwidth

In all these networks, except the full-mesh, the link bandwidth is shared among all connected hosts. Various algorithms have been proposed and are used to efficiently share the access to this resource. We explain several of them in the Medium Access Control section below.

Note: Fairness in computer networks

Sharing resources is important to ensure that the network efficiently serves its user. In practice, there are many ways to share resources. Some resource sharing schemes consider that some users are more important than others and should obtain more resources. For example, on the highways, police cars and ambulances have priority to use the highways. In some cities, traffic lanes are reserved for buses to promote public services, ... In computer networks, the same problem arise. Given that resources are limited, the network needs to enable users to efficiently share them. Before designing an efficient resource sharing scheme, one needs to first formalize its objectives. In computer networks, the most popular objective for resource sharing schemes is that they must be *fair*. In a simple situation, for example two hosts using a shared 2 Mbps link, the sharing scheme should allocate the same bandwidth to each user, in this case 1 Mbps. However, in a large networks, simply dividing the available resources by the number of users is not sufficient. Consider the network shown in the figure below where *A1* sends data to *A2*, *B1* to *B2*, ... In this network, how should we divide the bandwidth among the different flows ? A first approach would be to allocate the same bandwidth to each flow. In this case, each flow would obtain 5 Mbps and the link between *R2* and *R3* would not be fully loaded. Another approach would be to allocate 10 Mbps to *A1*-*A2*, 20 Mbps to *C1*-*C2* and nothing to *B1*-*B2*. This is clearly unfair.

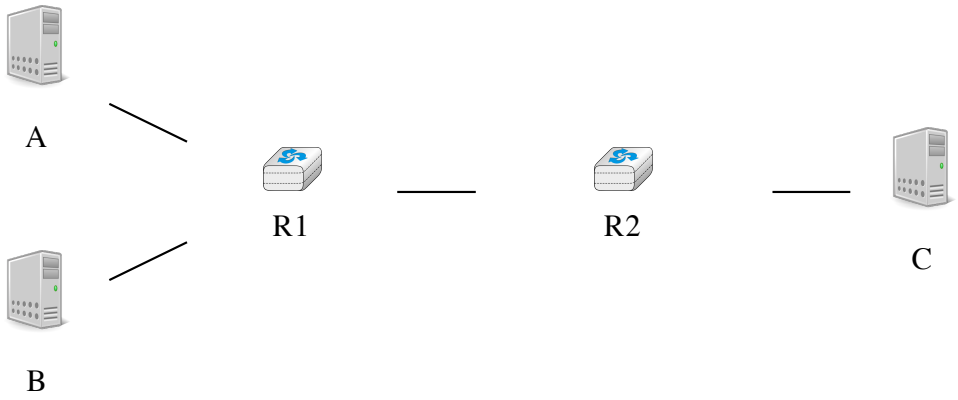


In large networks, fairness is always a compromise. The most widely used definition of fairness is the *max-min fairness*. A bandwidth allocation in a network is said to be *max-min fair* if it is such that it is impossible to allocate more bandwidth to one of the flows without reducing the bandwidth of a flow that already has a smaller allocation than the flow that we want to increase. If the network is completely known, it is possible to derive a *max-min fair* allocation as follows. Initially, all flows have a null bandwidth and they are placed in the candidate set. The bandwidth allocation of all flows in the candidate set is increased until one link becomes congested. At this point, the flows that use the congested link have reached their maximum allocation. They are removed from the candidate set and the process continues until the candidate set becomes empty.

In the above network, the allocation of all flows would grow until *A1-A2* and *B1-B2* reach 5 Mbps. At this point, link *R1-R2* becomes congested and these two flows have reached their maximum. The allocation for flow *C1-C2* can increase until reaching 15 Mbps. At this point, link *R2-R3* is congested. To increase the bandwidth allocated to *C1-C2*, one would need to reduce the allocation to flow *B1-B2*. Similarly, the only way to increase the allocation to flow *B1-B2* would require a decrease of the allocation to *A1-A2*.

2.6.2 Network congestion

Sharing bandwidth among the hosts directly attached to a link is not the only sharing problem that occurs in computer networks. To understand the general problem, let us consider a very simple network which contains only point-to-point links. This network contains three hosts and two network nodes. All links inside the network have the same capacity. For example, let us assume that all links have a bandwidth of 1000 bits per second and that the hosts send packets containing exactly one thousand bits.



In the network above, consider the case where host *A* is transmitting packets to destination *C*. *A* can send one packet per second and its packets will be delivered to *C*. Now, let us explore what happens when host *B* also starts to transmit a packet. Node *R1* will receive two packets that must be forwarded to *R2*. Unfortunately, due to the limited bandwidth on the *R1-R2* link, only one of these two packets can be transmitted. The outcome of the second packet will depend on the available buffers on *R1*. If *R1* has one available buffer, it could store the packet that has not been transmitted on the *R1-R2* link until the link becomes available. If *R1* does not have available buffers, then the packet needs to be discarded.

Besides the link bandwidth, the buffers on the network nodes are the second type of resource that needs to be shared inside the network. The node buffers play an important role in the operation of the network because that can be used to absorb transient traffic peaks. Consider again the example above. Assume that one average host *A* and host *B* send a group of three packets every ten seconds. Their combined transmission rate (0.6 packets per second) is, on average, lower than the network capacity (1 packet per second). However, if they both start to transmit at the same time, node *R1* will have to absorb a burst of packets. This burst of packets is a small *network congestion*. We will say that a network is congested, when the sum of the traffic demand from the hosts is larger than the network capacity $\sum demand > capacity$. This *network congestion* problem is one of the most difficult resource sharing problem in computer networks. *Congestion* occurs in almost all networks. Minimizing the amount of congestion is a key objective for many network operators. In most cases, they will have to accept transient congestion, i.e. congestion lasting a few seconds or perhaps minutes, but will want to prevent congestion that lasts days or months. For this, they can rely on a wide range of solutions. We briefly present some of these in the paragraphs below.

If *R1* has enough buffers, it will be able to absorb the load without having to discard packets. The packets sent by hosts *A* and *B* will reach their final destination *C*, but will experience a longer delay than when they are transmitting alone. The amount of buffering on the network node is the first paper that a network operator can tune to control congestion inside his network. Given the decreasing cost of memory, one could be tempted to put as many buffers [14] as possible on the network nodes. Let us consider this case in the network above and assume that *R1* has infinite buffers. Assume now that hosts *A* and *B* try to transmit a file that corresponds to one thousand packets each. Both are using a reliable protocol that relies on go-back-n to recover from transmission errors. The transmission starts and packets start to accumulate in *R1*'s buffers. These presence of these packets in the buffers increases the delay between the transmission of a packet by *A* and the return of the corresponding acknowledgement. Given the increasing delay, host *A* (and *B* as well) will consider that some of the packets that it sent have been lost. These packets will be retransmitted and will enter the buffers of *R1*. The occupancy of the buffers of *R1* will continue to increase and the delays as well. This will cause new retransmissions, ... In the end, several copies of the same packet will be transmitted over the *R1-R2*, but only one file will be delivered (very slowly) to the destination. This is known as the *congestion collapse* problem **RFC 896**. Congestion collapse is the nightmare for network operators. When it happens, the network carries packets without delivering useful data to the end users.

[14] There are still some vendors that try to put as many buffers as possible on their network nodes. A recent example is the buffer bloat problem that plagues some low-end Internet routers [GN2011].

Note: Congestion collapse on the Internet

Congestion collapse is unfortunately not only an academic experience. Van Jacobson reports in [Jacobson1988] one of these events that affected him while he was working at the Lawrence Berkeley Laboratory (LBL). LBL was two network nodes away from the University of California in Berkeley. At that time, the link between the two sites had a bandwidth of 32 Kbps, but some hosts were already attached to 10 Mbps LANs. "In October 1986, the data throughput from LBL to UC Berkeley ... dropped from 32 Kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad." This work lead to the development of various congestion control techniques that have allowed the Internet to continue to grow without experiencing widespread congestion collapse events.

Besides bandwidth and memory, a third resource that needs to be shared inside a network is the (packet) processing capacity. To forward a packet, a network node needs bandwidth on the outgoing link, but it also needs to analyze the packet header to perform a lookup inside its forwarding table. Performing these lookup operations require resources such as CPU cycles or memory accesses. Network nodes are usually designed to be able to sustain a given packet processing rate, measured in packets per second.

Note: Packets per second versus bits per second

The performance of network nodes can be characterized by two key metrics :

- the node's capacity measured in bits per second
- the node's lookup performance measured in packets per second

The node's capacity in bits per second mainly depends on the physical interfaces that it uses and also on the capacity of the internal interconnection (bus, crossbar switch, ...) between the different interfaces inside the node. Many vendors, in particular for low-end devices will use the sum of the bandwidth of the nodes' interfaces as the node capacity in bits per second. Measurements do not always match this maximum theoretical capacity. A well designed network node will usually have a capacity in bits per second larger than the sum of its link capacities. Such nodes will usually reach this maximum capacity when forwarding large packets.

When a network node forwards small packets, its performance is usually limited by the number of lookup operations that it can perform every second. This lookup performance is measured in packets per second. The performance may depend on the length of the forwarded packets. The key performance factor is the number of minimal size packets that are forwarded by the node every second. This rate can lead to a capacity in bits per second which is much lower than the sum of the bandwidth of the node's links.

Let us now try to present a broad overview of the congestion problem in networks. We will assume that the network is composed of dedicated links having a fixed bandwidth [15]. A network contains hosts that generate and receive packets and nodes that forward packets. Assuming that each host is connected via a single link to the network, the largest demand is $\sum AccessLinks$. In practice, this largest demand is never reached and the network will be engineered to sustain a much lower traffic demand. The difference between the worst-case traffic demand and the sustainable traffic demand can be large, up to several orders of magnitude. Fortunately, the hosts are not completely dumb and they can adapt their traffic demand to the current state of the network and the available bandwidth. For this, the hosts need to *sense* the current level of congestion and adjust their own traffic demand based on the estimated congestion. Network nodes can react in different ways to network congestion and hosts can sense the level of congestion in different ways.

Let us first explore which mechanisms can be used inside a network to control congestion and how these mechanisms can influence the behavior of the end hosts.

As explained earlier, one of the first manifestation of congestion on network nodes is the saturation of the network links that leads to a growth in the occupancy of the buffers of the node. This growth of the buffer occupancy implies that some packets will spend more time in the buffer and thus in the network. If hosts measure the network delays (e.g. by measuring the round-trip-time between the transmission of a packet and the return of the corresponding acknowledgement) they could start to sense congestion. On low bandwidth links, a growth in the buffer occupancy can lead to an increase of the delays which can be easily measured by the end hosts. On high bandwidth links, a

[15] Some networking technologies allow to adjust dynamically the bandwidth of links. For example, some devices can reduce their bandwidth to preserve energy. We ignore these technologies in this basic course and assume that all links used inside the network have a fixed bandwidth.

few packets inside the buffer will cause a small variation in the delay which may not necessarily be larger that the natural fluctuations of the delay measurements.

If the buffer's occupancy continues to grow, it will overflow and packets will need to be discarded. Discarding packets during congestion is the second possible reaction of a network node to congestion. Before looking at how a node can discard packets, it is interesting to discuss qualitatively the impact of the buffer occupancy on the reliable delivery of data through a network. This is illustrated by the figure below, adapted from [Jain1990].

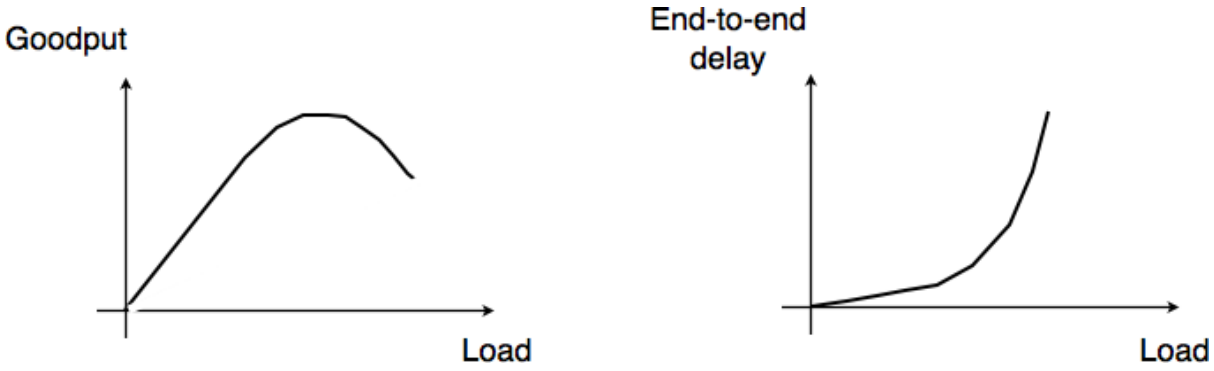


Figure 2.66: Network congestion

When the network load is low, buffer occupancy and link utilizations are low. The buffers on the network nodes are mainly used to absorb very short bursts of packets, but on average the traffic demand is lower than the network capacity. If the demand increases, the average buffer occupancy will increase as well. Measurements have shown that the total throughput increases as well. If the buffer occupancy is zero or very low, transmission opportunities on network links can be missed. This is not the case when the buffer occupancy is small but non zero. However, if the buffer occupancy continues to increase, the buffer becomes overloaded and the throughput does not increase anymore. When the buffer occupancy is close to the maximum, the throughput may decrease. This drop in throughput can be caused by excessive retransmissions of reliable protocols that incorrectly assume that previously sent packets have been lost while they are still waiting in the buffer. The network delay on the other hand increases with the buffer occupancy. In practice, a good operating point for a network buffer is a low occupancy to achieve high link utilization and also low delay for interactive applications.

Discarding packets is one of the signals that the network nodes can use to inform the hosts of the current level of congestion. Buffers on network nodes are usually used as FIFO queues to preserve packet ordering. Several *packet discard mechanisms* have been proposed for network nodes. These techniques basically answer two different questions :

- *What triggers a packet to be discarded ?* What are the conditions that lead a network node to decide to discard a packet. The simplest answer to this question is : *When the buffer is full*. Although this is a good congestion indication, it is probably not the best one from a performance viewpoint. An alternative is to discard packets when the buffer occupancy grows too much. In this case, it is likely that the buffer will become full shortly. Since packet discarding is an information that allows hosts to adapt their transmission rate, discarding packets early could allow hosts to react earlier and thus prevent congestion from happening.
- *Which packet(s) should be discarded ?* Once the network node has decided to discard packets, it needs to actually discard real packets.

By combining different answers to these questions, network researchers have developed different packet discard mechanisms.

- *tail drop* is the simplest packet discard technique. When a buffer is full, the arriving packet is discarded. *Tail drop* can be easily implemented. This is, by far, the most widely used packet discard mechanism. However, it suffers from two important drawbacks. First, since *tail drop* discards packets only when the buffer is full, buffers tend to be congested and realtime applications may suffer from the increased delays. Second, *tail drop* is blind when it discards a packet. It may discard a packet from a low bandwidth interactive flow while most of the buffer is used by large file transfers.
- *drop from front* is an alternative packet discard technique. Instead of removing the arriving packet, it removes the packet that was at the head of the queue. Discarding this packet instead of the arriving one can

have two advantages. First, it already stayed a long time in the buffer. Second, hosts should be able to detect the loss (and thus the congestion) earlier.

- *probabilistic drop*. Various random drop techniques have been proposed. Compared to the previous techniques. A frequently cited technique is *Random Early Discard* (RED) [FJ1993]. RED measures the average buffer occupancy and probabilistically discards packets when this average occupancy is too high. Compared to *tail drop* and *drop from front*, an advantage of *RED* is that thanks to the probabilistic drops, packets should be discarded from different flows in proportion of their bandwidth.

Discarding packets is a frequent reaction to network congestion. Unfortunately, discarding packets is not optimal since a packet which is discarded on a network node has already consumed resources on the upstream nodes. There are other ways for the network to inform the end hosts of the current congestion level. A first solution is to mark the packets when a node is congested. Several networking technologies have relied on this kind of packet marking.

In datagram networks, *Forward Explicit Congestion Notification* (FECN) can be used. One field of the packet header, typically one bit, is used to indicate congestion. When a host sends a packet, the congestion bit is reset. If the packet passes through a congested node, the congestion bit is set. The destination can then determine the current congestion level by measuring the fraction of the packets that it received with the congestion bit set. It may then return this information to the sending host to allow it to adapt its retransmission rate. Compared to packet discarding, the main advantage of FECN is that hosts can detect congestion explicitly without having to rely on packet losses.

In virtual circuit networks, packet marking can be improved if the return packets follow the reverse path of the forward packets. It this case, a network node can detect congestion on the forward path (e.g. due to the size of its buffer), but mark the packets on the return path. Marking the return packets (e.g. the acknowledgements used by reliable protocols) provides a faster feedback to the sending hosts compared to FECN. This technique is usually called *Backward Explicit Congestion Notification (BECN)*.

If the packet header does not contain any bit in the header to represent the current congestion level, an alternative is to allow the network nodes to send a control packet to the source to indicate the current congestion level. Some networking technologies use such control packets to explicitly regulate the transmission rate of sources. However, their usage is mainly restricted to small networks. In large networks, network nodes usually avoid using such control packets. These controlled packets are even considered to be dangerous in some networks. First, using them increases the network load when the network is congested. Second, while network nodes are optimized to forward packets, they are usually pretty slow at creating new packets.

Dropping and marking packets is not the only possible reaction of a router that becomes congested. A router could also selectively delay packets belonging to some flows. There are different algorithms that can be used by a router to delay packets. If the objective of the router is to fairly distribute to bandwidth of an output link among competing flows, one possibility is to organize the buffers of the router as a set of queues. For simplicity, let us assume that the router is capable of supporting a fixed number of concurrent flows, say *N*. One of the queues of the router is associated to each flow and when a packet arrives, it is placed at the tail of the corresponding queue. All the queues are controlled by a *scheduler*. A *scheduler* is an algorithm that is run each time there is an opportunity to transmit a packet on the outgoing link. Various schedulers have been proposed in the scientific literature and some are used in real routers.

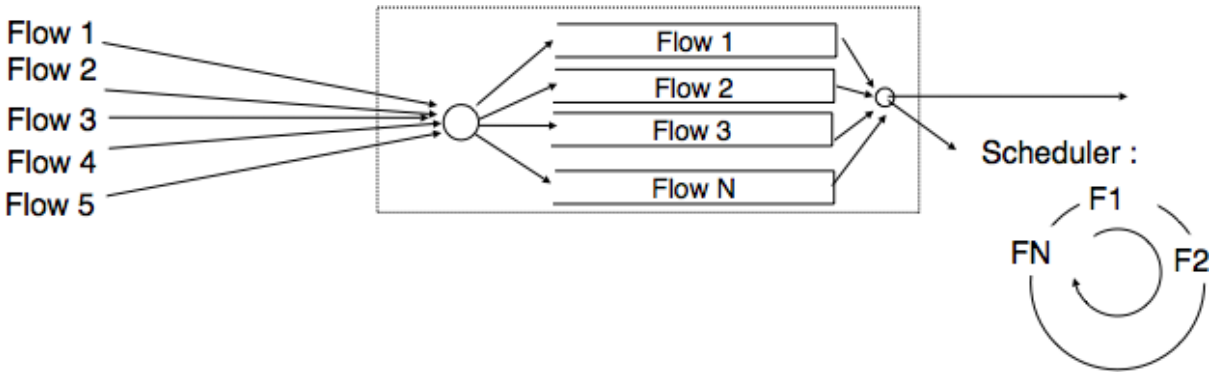


Figure 2.67: A round-robin scheduler

A very simple scheduler is the *round-robin scheduler*. This scheduler serves all the queues in a round-robin fashion. If all flows send packets of the same size, then the round-robin scheduler allocates the bandwidth fairly among the different flows. Otherwise, it favors flows that are using larger packets. Extensions to the *round-robin scheduler* have been proposed to provide a fair distribution of the bandwidth with variable-length packets [SV1995] but these are outside the scope of this chapter.

```
# N queues
# state variable : next_queue
next_queue=0
while (true) :
  if isEmpty(buffer) :
    wait
    # wait for next packet in buffer
  if !isEmpty(queue[next_queue]) :
      # Send packet at head of next_queue
      p=remove_packet(queue[next_queue])
     send(p)
  next_queue=(next_queue+1)%N
# end while
```

2.6.3 Distributing the load across the network

Delays, packet discards, packet markings and control packets are the main types of information that the network can exchange with the end hosts. Discarding packets is the main action that a network node can perform if the congestion is too severe. Besides tackling congestion at each node, it is also possible to change the divert some traffic flows from heavily loaded links to reduce congestion. Early routing algorithms [MRR1980] have used delay measurements to detect congestion between network nodes and update the link weights dynamically. By reflecting the delay perceived by applications in the link weights used for the shortest paths computation, these routing algorithms managed to dynamically change the forwarding paths in reaction to congestion. However, deployment experience showed that these dynamic routing algorithms could cause oscillations and did not necessarily lower congestion. Deployed datagram networks rarely use dynamic routing algorithms, except in some wireless networks. In datagram networks, the state of the art reaction to long term congestion, i.e. congestion lasting hours, days or more, is to measure the traffic demand and then select the link weights [FRT2002] that allow to minimize the maximum link loads. If the congestion lasts longer, changing the weights is not sufficient anymore and the network needs to be upgraded with few or faster links. However, in Wide Area Networks, adding new links can take months.

In virtual circuit networks, another way to manage or prevent congestion is to limit the number of circuits that use the network at any time. This technique is usually called *connection admission control*. When a host requests the creation of a new circuit in the network, it specifies the destination and in some networking technologies the required bandwidth. With this information, the network can check whether there are enough resources available to reach this particular destination. If yes, the circuit is established. However, the requested is denied and the host will have to defer the creation of its virtual circuit. *Connection admission control* schemes are widely used in the telephone networks. In these networks, a busy tone corresponds to an unavailable destination or a congested network.

In datagram networks, this technique cannot be easily used since the basic assumption of such a network is that a host can send any packet towards any destination at any time. A host does not need to request the authorization of the network to send packets towards a particular destination.

Based on the feedback received from the network, the hosts can adjust their transmission rate. We discuss in section *Congestion control* some techniques that allow hosts to react to congestion.

Another way to share the network resources is to distribute the load across multiple links. Many techniques have been designed to spread the load over the network. As an illustration, let us briefly consider how load can be shared when accessing some content. Consider a large and popular file such as the image of a Linux distribution or the upgrade of a commercial operating system that will be downloaded by many users. There are many ways to distribute this large file. A naive solution is to place one copy of the file on a server and allow all users to download this file from the server. If the file is popular and millions of users want to download it, the server will quickly become overloaded. There are two classes of solutions that can be used to serve a large number of users.

A first approach is to store the file on servers whose name is known by the clients. Before retrieving the file, each client will query the name service to obtain the address of the server. If the file is available from many servers, the name service can provide different addresses to different clients. This will automatically spread the load since different clients will download the file from different servers. Most large content providers use such a solution to distribute large files or videos.

There is another solution that allows to spread the load among many sources without relying on the name service. The popular bittorent service is an example of this approach. With this solution, each file is divided in blocks of a fixed size. To retrieve a file, a client needs to retrieve all the blocks that compose the file. However, nothing forces the client to retrieve all the blocks in sequence and from the same server. Each file is associated with metadata that indicates for each block a list of addresses of hosts that store this block. To retrieve a complete file, a client first downloads the metadata. Then, it tries to retrieve each block from one of the hosts that store the block. In practice, implementations often try to download several blocks in parallel. Once one block has been successfully downloaded, the next block can be requested. If a host is slow to provide one block or becomes unavailable, the client can contact another host listed in the metadata. Most deployments of bittorrent allow the clients to participate to the distribution of blocks. Once a client has downloaded one block, it contacts the server which stores the metadata to indicate that it can also provide this block. With this scheme, when a file is popular, its blocks are downloaded by many hosts that automatically participate in the distribution of the blocks. Thus, the number of *servers* that are capable of providing blocks from a popular file automatically increases with the file's popularity.

Now that we have provided a broad overview of the techniques that can be used to spread the load and allocate resources in the network, let us analyze two techniques in more details : Medium Access Control and Congestion control.

2.6.4 Medium Access Control algorithms

The common problem among Local Area Networks is how to efficiently share the available bandwidth. If two devices send a frame at the same time, the two electrical, optical or radio signals that correspond to these frames will appear at the same time on the transmission medium and a receiver will not be able to decode either frame. Such simultaneous transmissions are called *collisions*. A *collision* may involve frames transmitted by two or more devices attached to the Local Area Network. Collisions are the main cause of errors in wired Local Area Networks.

All Local Area Network technologies rely on a *Medium Access Control* algorithm to regulate the transmissions to either minimise or avoid collisions. There are two broad families of *Medium Access Control* algorithms :

1. *Deterministic* or *pessimistic* MAC algorithms. These algorithms assume that collisions are a very severe problem and that they must be completely avoided. These algorithms ensure that at any time, at most one device is allowed to send a frame on the LAN. This is usually achieved by using a distributed protocol which elects one device that is allowed to transmit at each time. A deterministic MAC algorithm ensures that no collision will happen, but there is some overhead in regulating the transmission of all the devices attached to the LAN.
2. *Stochastic* or *optimistic* MAC algorithms. These algorithms assume that collisions are part of the normal operation of a Local Area Network. They aim to minimise the number of collisions, but they do not try to avoid all collisions. Stochastic algorithms are usually easier to implement than deterministic ones.

We first discuss a simple deterministic MAC algorithm and then we describe several important optimistic algorithms, before coming back to a distributed and deterministic MAC algorithm.

Static allocation methods

A first solution to share the available resources among all the devices attached to one Local Area Network is to define, *a priori*, the distribution of the transmission resources among the different devices. If N devices need to share the transmission capacities of a LAN operating at b Mbps, each device could be allocated a bandwidth of $\frac{b}{N}$ Mbps.

Limited resources need to be shared in other environments than Local Area Networks. Since the first radio transmissions by Marconi more than one century ago, many applications that exchange information through radio signals have been developed. Each radio signal is an electromagnetic wave whose power is centered around a

given frequency. The radio spectrum corresponds to frequencies ranging between roughly 3 KHz and 300 GHz. Frequency allocation plans negotiated among governments reserve most frequency ranges for specific applications such as broadcast radio, broadcast television, mobile communications, aeronautical radio navigation, amateur radio, satellite, etc. Each frequency range is then subdivided into channels and each channel can be reserved for a given application, e.g. a radio broadcaster in a given region.

Frequency Division Multiplexing (FDM) is a static allocation scheme in which a frequency is allocated to each device attached to the shared medium. As each device uses a different transmission frequency, collisions cannot occur. In optical networks, a variant of FDM called *Wavelength Division Multiplexing* (WDM) can be used. An optical fiber can transport light at different wavelengths without interference. With WDM, a different wavelength is allocated to each of the devices that share the same optical fiber.

Time Division Multiplexing (TDM) is a static bandwidth allocation method that was initially defined for the telephone network. In the fixed telephone network, a voice conversation is usually transmitted as a 64 Kbps signal. Thus, a telephone conservation generates 8 KBytes per second or one byte every 125 microseconds. Telephone conversations often need to be multiplexed together on a single line. For example, in Europe, thirty 64 Kbps voice signals are multiplexed over a single 2 Mbps (E1) line. This is done by using *Time Division Multiplexing* (TDM). TDM divides the transmission opportunities into slots. In the telephone network, a slot corresponds to 125 microseconds. A position inside each slot is reserved for each voice signal. The figure below illustrates TDM on a link that is used to carry four voice conversations. The vertical lines represent the slot boundaries and the letters the different voice conversations. One byte from each voice conversation is sent during each 125 microseconds slot. The byte corresponding to a given conversation is always sent at the same position in each slot.

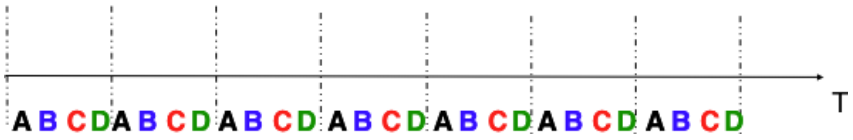


Figure 2.68: Time-division multiplexing

TDM as shown above can be completely static, i.e. the same conversations always share the link, or dynamic. In the latter case, the two endpoints of the link must exchange messages specifying which conversation uses which byte inside each slot. Thanks to these signalling messages, it is possible to dynamically add and remove voice conversations from a given link.

TDM and FDM are widely used in telephone networks to support fixed bandwidth conversations. Using them in Local Area Networks that support computers would probably be inefficient. Computers usually do not send information at a fixed rate. Instead, they often have an on-off behaviour. During the on period, the computer tries to send at the highest possible rate, e.g. to transfer a file. During the off period, which is often much longer than the on period, the computer does not transmit any packet. Using a static allocation scheme for computers attached to a LAN would lead to huge inefficiencies, as they would only be able to transmit at $\frac{1}{N}$ of the total bandwidth during their on period, despite the fact that the other computers are in their off period and thus do not need to transmit any information. The dynamic MAC algorithms discussed in the remainder of this chapter aim solve this problem.

ALOHA

In the 1960s, computers were mainly mainframes with a few dozen terminals attached to them. These terminals were usually in the same building as the mainframe and were directly connected to it. In some cases, the terminals were installed in remote locations and connected through a *modem* attached to a *dial-up line*. The university of Hawaii chose a different organisation. Instead of using telephone lines to connect the distant terminals, they developed the first *packet radio* technology [Abramson1970]. Until then, computer networks were built on top of either the telephone network or physical cables. ALOHANet showed that it was possible to use radio signals to interconnect computers.

The first version of ALOHANet, described in [Abramson1970], operated as follows: First, the terminals and the mainframe exchanged fixed-length frames composed of 704 bits. Each frame contained 80 8-bit characters, some control bits and parity information to detect transmission errors. Two channels in the 400 MHz range were reserved for the operation of ALOHANet. The first channel was used by the mainframe to send frames to all terminals.

The second channel was shared among all terminals to send frames to the mainframe. As all terminals share the same transmission channel, there is a risk of collision. To deal with this problem as well as transmission errors, the mainframe verified the parity bits of the received frame and sent an acknowledgement on its channel for each correctly received frame. The terminals on the other hand had to retransmit the unacknowledged frames. As for TCP, retransmitting these frames immediately upon expiration of a fixed timeout is not a good approach as several terminals may retransmit their frames at the same time leading to a network collapse. A better approach, but still far from perfect, is for each terminal to wait a random amount of time after the expiration of its retransmission timeout. This avoids synchronisation among multiple retransmitting terminals.

The pseudo-code below shows the operation of an ALOHANet terminal. We use this python syntax for all Medium Access Control algorithms described in this chapter. The algorithm is applied to each new frame that needs to be transmitted. It attempts to transmit a frame at most *max* times (*while loop*). Each transmission attempt is performed as follows: First, the frame is sent. Each frame is protected by a timeout. Then, the terminal waits for either a valid acknowledgement frame or the expiration of its timeout. If the terminal receives an acknowledgement, the frame has been delivered correctly and the algorithm terminates. Otherwise, the terminal waits for a random time and attempts to retransmit the frame.

```
# ALOHA
N=1
while N<= max :
   send(frame)
   wait(ack_on_return_channel or timeout)
   if (ack_on_return_channel):
       break  # transmission was successful
   else:
       # timeout
       wait(random_time)
       N=N+1
 else:
   # Too many transmission attempts
```

[Abramson1970] analysed the performance of ALOHANet under particular assumptions and found that ALOHANet worked well when the channel was lightly loaded. In this case, the frames are rarely retransmitted and the *channel traffic*, i.e. the total number of (correct and retransmitted) frames transmitted per unit of time is close to the *channel utilization*, i.e. the number of correctly transmitted frames per unit of time. Unfortunately, the analysis also reveals that the *channel utilization* reaches its maximum at $\frac{1}{2\times e} = 0.186$ times the channel bandwidth. At higher utilization, ALOHANet becomes unstable and the network collapses due to collided retransmissions.

Note: Amateur packet radio

Packet radio technologies have evolved in various directions since the first experiments performed at the University of Hawaii. The Amateur packet radio service developed by amateur radio operators is one of the descendants ALOHANet. Many amateur radio operators are very interested in new technologies and they often spend countless hours developing new antennas or transceivers. When the first personal computers appeared, several amateur radio operators designed radio modems and their own datalink layer protocols [KPD1985] [BNT1997]. This network grew and it was possible to connect to servers in several European countries by only using packet radio relays. Some amateur radio operators also developed TCP/IP protocol stacks that were used over the packet radio service. Some parts of the amateur packet radio network are connected to the global Internet and use the *44.0.0.0/8* prefix.

Many improvements to ALOHANet have been proposed since the publication of [Abramson1970], and this technique, or some of its variants, are still found in wireless networks today. The slotted technique proposed in [Roberts1975] is important because it shows that a simple modification can significantly improve channel utilization. Instead of allowing all terminals to transmit at any time, [Roberts1975] proposed to divide time into slots and allow terminals to transmit only at the beginning of each slot. Each slot corresponds to the time required to transmit one fixed size frame. In practice, these slots can be imposed by a single clock that is received by all terminals. In ALOHANet, it could have been located on the central mainframe. The analysis in [Roberts1975] reveals that this simple modification improves the channel utilization by a factor of two.

Carrier Sense Multiple Access

ALOHA and slotted ALOHA can easily be implemented, but unfortunately, they can only be used in networks that are very lightly loaded. Designing a network for a very low utilisation is possible, but it clearly increases the cost of the network. To overcome the problems of ALOHA, many Medium Access Control mechanisms have been proposed which improve channel utilization. Carrier Sense Multiple Access (CSMA) is a significant improvement compared to ALOHA. CSMA requires all nodes to listen to the transmission channel to verify that it is free before transmitting a frame [KT1975]. When a node senses the channel to be busy, it defers its transmission until the channel becomes free again. The pseudo-code below provides a more detailed description of the operation of CSMA.

```
# persistent CSMA
N=1
while N<= max :
  wait(channel_becomes_free)
  send(frame)
  wait(ack or timeout)
  if ack :
      break  # transmission was successful
  else :
      # timeout
      N=N+1
# end of while loop
  # Too many transmission attempts
```

The above pseudo-code is often called *persistent CSMA* [KT1975] as the terminal will continuously listen to the channel and transmit its frame as soon as the channel becomes free. Another important variant of CSMA is the *non-persistent CSMA* [KT1975]. The main difference between persistent and non-persistent CSMA described in the pseudo-code below is that a non-persistent CSMA node does not continuously listen to the channel to determine when it becomes free. When a non-persistent CSMA terminal senses the transmission channel to be busy, it waits for a random time before sensing the channel again. This improves channel utilization compared to persistent CSMA. With persistent CSMA, when two terminals sense the channel to be busy, they will both transmit (and thus cause a collision) as soon as the channel becomes free. With non-persistent CSMA, this synchronisation does not occur, as the terminals wait a random time after having sensed the transmission channel. However, the higher channel utilization achieved by non-persistent CSMA comes at the expense of a slightly higher waiting time in the terminals when the network is lightly loaded.

```
# Non persistent CSMA
N=1
while N<= max :
   listen(channel)
   if free(channel):
      send(frame)
      wait(ack or timeout)
      if received(ack) :
         break  # transmission was successful
      else :
         # timeout
         N=N+1
   else:
      wait(random_time)
 # end of while loop
   # Too many transmission attempts
```

[KT1975] analyzes in detail the performance of several CSMA variants. Under some assumptions about the transmission channel and the traffic, the analysis compares ALOHA, slotted ALOHA, persistent and non-persistent CSMA. Under these assumptions, ALOHA achieves a channel utilization of only 18.4% of the channel capacity. Slotted ALOHA is able to use 36.6% of this capacity. Persistent CSMA improves the utilization by reaching 52.9% of the capacity while non-persistent CSMA achieves 81.5% of the channel capacity.

Carrier Sense Multiple Access with Collision Detection

CSMA improves channel utilization compared to ALOHA. However, the performance can still be improved, especially in wired networks. Consider the situation of two terminals that are connected to the same cable. This cable could, for example, be a coaxial cable as in the early days of Ethernet [Metcalfe1976]. It could also be built with twisted pairs. Before extending CSMA, it is useful to understand more intuitively, how frames are transmitted in such a network and how collisions can occur. The figure below illustrates the physical transmission of a frame on such a cable. To transmit its frame, host A must send an electrical signal on the shared medium. The first step is thus to begin the transmission of the electrical signal. This is point *(1)* in the figure below. This electrical signal will travel along the cable. Although electrical signals travel fast, we know that information cannot travel faster than the speed of light (i.e. 300.000 kilometers/second). On a coaxial cable, an electrical signal is slightly slower than the speed of light and 200.000 kilometers per second is a reasonable estimation. This implies that if the cable has a length of one kilometer, the electrical signal will need 5 microseconds to travel from one end of the cable to the other. The ends of coaxial cables are equipped with termination points that ensure that the electrical signal is not reflected back to its source. This is illustrated at point *(3)* in the figure, where the electrical signal has reached the left endpoint and host B. At this point, B starts to receive the frame being transmitted by A. Notice that there is a delay between the transmission of a bit on host A and its reception by host B. If there were other hosts attached to the cable, they would receive the first bit of the frame at slightly different times. As we will see later, this timing difference is a key problem for MAC algorithms. At point *(4)*, the electrical signal has reached both ends of the cable and occupies it completely. Host A continues to transmit the electrical signal until the end of the frame. As shown at point *(5)*, when the sending host stops its transmission, the electrical signal corresponding to the end of the frame leaves the coaxial cable. The channel becomes empty again once the entire electrical signal has been removed from the cable.

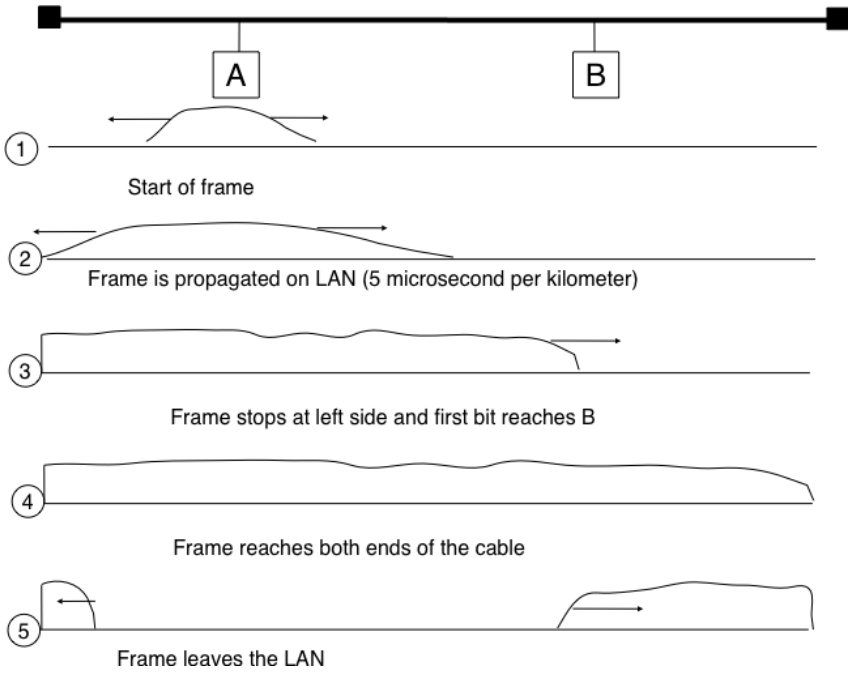


Figure 2.69: Frame transmission on a shared bus

Now that we have looked at how a frame is actually transmitted as an electrical signal on a shared bus, it is interesting to look in more detail at what happens when two hosts transmit a frame at almost the same time. This is illustrated in the figure below, where hosts A and B start their transmission at the same time (point *(1)*). At this time, if host C senses the channel, it will consider it to be free. This will not last a long time and at point *(2)* the electrical signals from both host A and host B reach host C. The combined electrical signal (shown graphically as the superposition of the two curves in the figure) cannot be decoded by host C. Host C detects a collision, as it receives a signal that it cannot decode. Since host C cannot decode the frames, it cannot determine which hosts are sending the colliding frames. Note that host A (and host B) will detect the collision after host C (point *(3)* in the figure below).

As shown above, hosts detect collisions when they receive an electrical signal that they cannot decode. In a wired

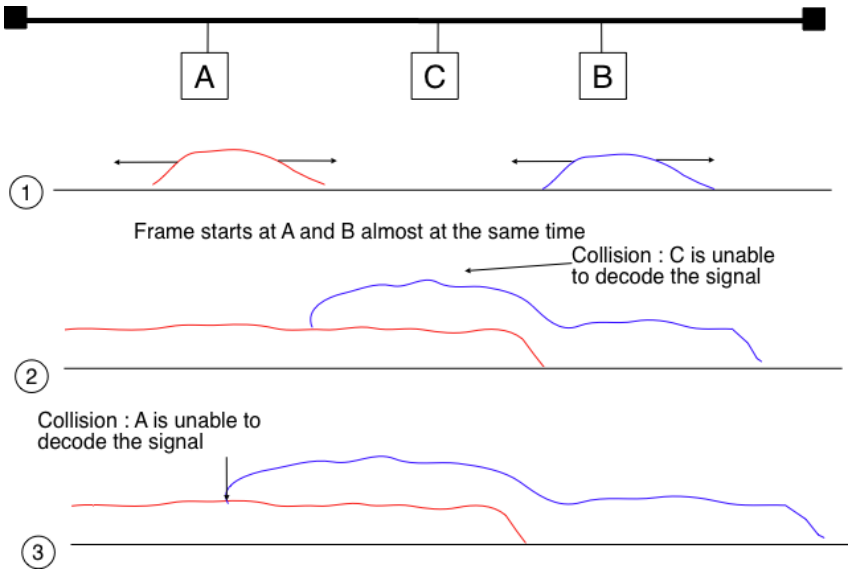


Figure 2.70: Frame collision on a shared bus

network, a host is able to detect such a collision both while it is listening (e.g. like host C in the figure above) and also while it is sending its own frame. When a host transmits a frame, it can compare the electrical signal that it transmits with the electrical signal that it senses on the wire. At points *(1)* and *(2)* in the figure above, host A senses only its own signal. At point *(3)*, it senses an electrical signal that differs from its own signal and can thus detects the collision. At this point, its frame is corrupted and it can stop its transmission. The ability to detect collisions while transmitting is the starting point for the *Carrier Sense Multiple Access with Collision Detection (CSMA/CD)* Medium Access Control algorithm, which is used in Ethernet networks [Metcalfe1976] [802.3] . When an Ethernet host detects a collision while it is transmitting, it immediately stops its transmission. Compared with pure CSMA, CSMA/CD is an important improvement since when collisions occur, they only last until colliding hosts have detected it and stopped their transmission. In practice, when a host detects a collision, it sends a special jamming signal on the cable to ensure that all hosts have detected the collision.

To better understand these collisions, it is useful to analyse what would be the worst collision on a shared bus network. Let us consider a wire with two hosts attached at both ends, as shown in the figure below. Host A starts to transmit its frame and its electrical signal is propagated on the cable. Its propagation time depends on the physical length of the cable and the speed of the electrical signal. Let us use τ to represent this propagation delay in seconds. Slightly less than τ seconds after the beginning of the transmission of A's frame, B decides to start transmitting its own frame. After ϵ seconds, B senses A's frame, detects the collision and stops transmitting. The beginning of B's frame travels on the cable until it reaches host A. Host A can thus detect the collision at time $\tau - \epsilon + \tau \approx 2 \times \tau$. An important point to note is that a collision can only occur during the first $2 \times \tau$ seconds of its transmission. If a collision did not occur during this period, it cannot occur afterwards since the transmission channel is busy after τ seconds and CSMA/CD hosts sense the transmission channel before transmitting their frame.

Furthermore, on the wired networks where CSMA/CD is used, collisions are almost the only cause of transmission errors that affect frames. Transmission errors that only affect a few bits inside a frame seldom occur in these wired networks. For this reason, the designers of CSMA/CD chose to completely remove the acknowledgement frames in the datalink layer. When a host transmits a frame, it verifies whether its transmission has been affected by a collision. If not, given the negligible Bit Error Ratio of the underlying network, it assumes that the frame was received correctly by its destination. Otherwise the frame is retransmitted after some delay.

Removing acknowledgements is an interesting optimisation as it reduces the number of frames that are exchanged on the network and the number of frames that need to be processed by the hosts. However, to use this optimisation, we must ensure that all hosts will be able to detect all the collisions that affect their frames. The problem is important for short frames. Let us consider two hosts, A and B, that are sending a small frame to host C as illustrated in the figure below. If the frames sent by A and B are very short, the situation illustrated below may occur. Hosts A and B send their frame and stop transmitting (point *(1)*). When the two short frames arrive at the location of host C, they collide and host C cannot decode them (point *(2)*). The two frames are absorbed by the

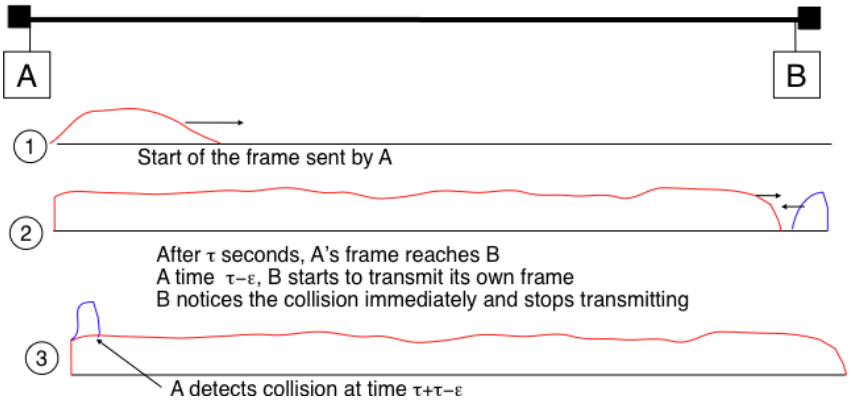


Figure 2.71: The worst collision on a shared bus

ends of the wire. Neither host A nor host B have detected the collision. They both consider their frame to have been received correctly by its destination.

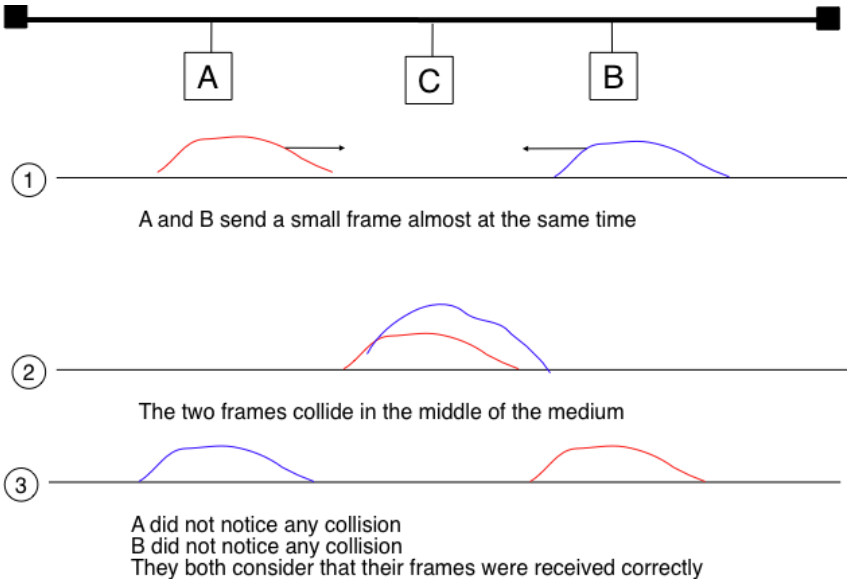


Figure 2.72: The short-frame collision problem

To solve this problem, networks using CSMA/CD require hosts to transmit for at least $2 \times \tau$ seconds. Since the network transmission speed is fixed for a given network technology, this implies that a technology that uses CSMA/CD enforces a minimum frame size. In the most popular CSMA/CD technology, Ethernet, $2 \times \tau$ is called the *slot time* [16].

The last innovation introduced by CSMA/CD is the computation of the retransmission timeout. As for ALOHA, this timeout cannot be fixed, otherwise hosts could become synchronised and always retransmit at the same time. Setting such a timeout is always a compromise between the network access delay and the amount of collisions. A short timeout would lead to a low network access delay but with a higher risk of collisions. On the other hand, a long timeout would cause a long network access delay but a lower risk of collisions. The *binary exponential back-off* algorithm was introduced in CSMA/CD networks to solve this problem.

To understand *binary exponential back-off*, let us consider a collision caused by exactly two hosts. Once it has detected the collision, a host can either retransmit its frame immediately or defer its transmission for some time. If each colliding host flips a coin to decide whether to retransmit immediately or to defer its retransmission, four cases are possible :

[16] This name should not be confused with the duration of a transmission slot in slotted ALOHA. In CSMA/CD networks, the slot time is the time during which a collision can occur at the beginning of the transmission of a frame. In slotted ALOHA, the duration of a slot is the transmission time of an entire fixed-size frame.

1. Both hosts retransmit immediately and a new collision occurs
2. The first host retransmits immediately and the second defers its retransmission
3. The second host retransmits immediately and the first defers its retransmission
4. Both hosts defer their retransmission and a new collision occurs

In the second and third cases, both hosts have flipped different coins. The delay chosen by the host that defers its retransmission should be long enough to ensure that its retransmission will not collide with the immediate retransmission of the other host. However the delay should not be longer than the time necessary to avoid the collision, because if both hosts decide to defer their transmission, the network will be idle during this delay. The *slot time* is the optimal delay since it is the shortest delay that ensures that the first host will be able to retransmit its frame completely without any collision.

If two hosts are competing, the algorithm above will avoid a second collision 50% of the time. However, if the network is heavily loaded, several hosts may be competing at the same time. In this case, the hosts should be able to automatically adapt their retransmission delay. The *binary exponential back-off* performs this adaptation based on the number of collisions that have affected a frame. After the first collision, the host flips a coin and waits 0 or 1 *slot time*. After the second collision, it generates a random number and waits 0, 1, 2 or 3 *slot times*, etc. The duration of the waiting time is doubled after each collision. The complete pseudo-code for the CSMA/CD algorithm is shown in the figure below.

```
# CSMA/CD pseudo-code
N=1
while N<= max :
   wait(channel_becomes_free)
   send(frame)
   wait_until (end_of_frame) or (collision)
   if collision detected:
        stop transmitting
        send(jamming)
        k = min (10, N)
        r = random(0, 2k - 1) * slotTime
        wait(r*slotTime)
        N=N+1
   else :
        wait(inter-frame_delay)
        break
 # end of while loop
   # Too many transmission attempts
```

The inter-frame delay used in this pseudo-code is a short delay corresponding to the time required by a network adapter to switch from transmit to receive mode. It is also used to prevent a host from sending a continuous stream of frames without leaving any transmission opportunities for other hosts on the network. This contributes to the fairness of CSMA/CD. Despite this delay, there are still conditions where CSMA/CD is not completely fair [RY1994]. Consider for example a network with two hosts : a server sending long frames and a client sending acknowledgments. Measurements reported in [RY1994] have shown that there are situations where the client could suffer from repeated collisions that lead it to wait for long periods of time due to the exponential back-off algorithm.

Carrier Sense Multiple Access with Collision Avoidance

The *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) Medium Access Control algorithm was designed for the popular WiFi wireless network technology [802.11]. CSMA/CA also senses the transmission channel before transmitting a frame. Furthermore, CSMA/CA tries to avoid collisions by carefully tuning the timers used by CSMA/CA devices.

CSMA/CA uses acknowledgements like CSMA. Each frame contains a sequence number and a CRC. The CRC is used to detect transmission errors while the sequence number is used to avoid frame duplication. When a device receives a correct frame, it returns a special acknowledgement frame to the sender. CSMA/CA introduces a small delay, named *Short Inter Frame Spacing* (SIFS), between the reception of a frame and the transmission of

the acknowledgement frame. This delay corresponds to the time that is required to switch the radio of a device between the reception and transmission modes.

Compared to CSMA, CSMA/CA defines more precisely when a device is allowed to send a frame. First, CSMA/CA defines two delays : *DIFS* and *EIFS*. To send a frame, a device must first wait until the channel has been idle for at least the *Distributed Coordination Function Inter Frame Space* (DIFS) if the previous frame was received correctly. However, if the previously received frame was corrupted, this indicates that there are collisions and the device must sense the channel idle for at least the *Extended Inter Frame Space* (EIFS), with $SIFS < DIFS < EIFS$. The exact values for SIFS, DIFS and EIFS depend on the underlying physical layer [802.11].

The figure below shows the basic operation of CSMA/CA devices. Before transmitting, host *A* verifies that the channel is empty for a long enough period. Then, its sends its data frame. After checking the validity of the received frame, the recipient sends an acknowledgement frame after a short SIFS delay. Host *C*, which does not participate in the frame exchange, senses the channel to be busy at the beginning of the data frame. Host *C* can use this information to determine how long the channel will be busy for. Note that as $SIFS < DIFS < EIFS$, even a device that would start to sense the channel immediately after the last bit of the data frame could not decide to transmit its own frame during the transmission of the acknowledgement frame.

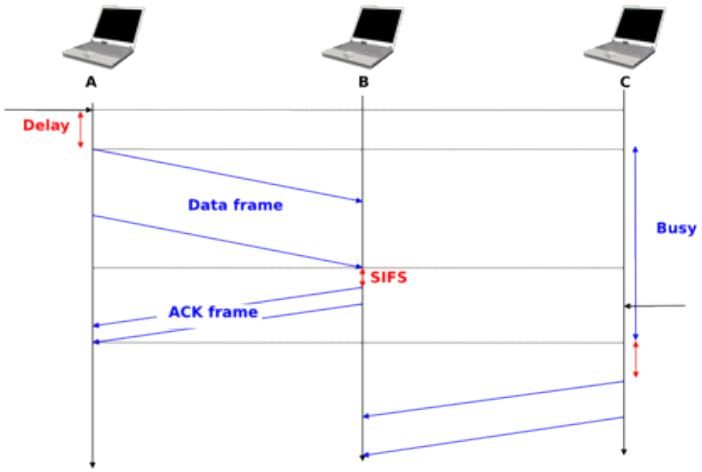


Figure 2.73: Operation of a CSMA/CA device

The main difficulty with CSMA/CA is when two or more devices transmit at the same time and cause collisions. This is illustrated in the figure below, assuming a fixed timeout after the transmission of a data frame. With CSMA/CA, the timeout after the transmission of a data frame is very small, since it corresponds to the SIFS plus the time required to transmit the acknowledgement frame.

To deal with this problem, CSMA/CA relies on a backoff timer. This backoff timer is a random delay that is chosen by each device in a range that depends on the number of retransmissions for the current frame. The range grows exponentially with the retransmissions as in CSMA/CD. The minimum range for the backoff timer is $[0, 7 * slotTime]$ where the *slotTime* is a parameter that depends on the underlying physical layer. Compared to CSMA/CD's exponential backoff, there are two important differences to notice. First, the initial range for the backoff timer is seven times larger. This is because it is impossible in CSMA/CA to detect collisions as they happen. With CSMA/CA, a collision may affect the entire frame while with CSMA/CD it can only affect the beginning of the frame. Second, a CSMA/CA device must regularly sense the transmission channel during its back off timer. If the channel becomes busy (i.e. because another device is transmitting), then the back off timer must be frozen until the channel becomes free again. Once the channel becomes free, the back off timer is restarted. This is in contrast with CSMA/CD where the back off is recomputed after each collision. This is illustrated in the figure below. Host *A* chooses a smaller backoff than host *C*. When *C* senses the channel to be busy, it freezes its backoff timer and only restarts it once the channel is free again.

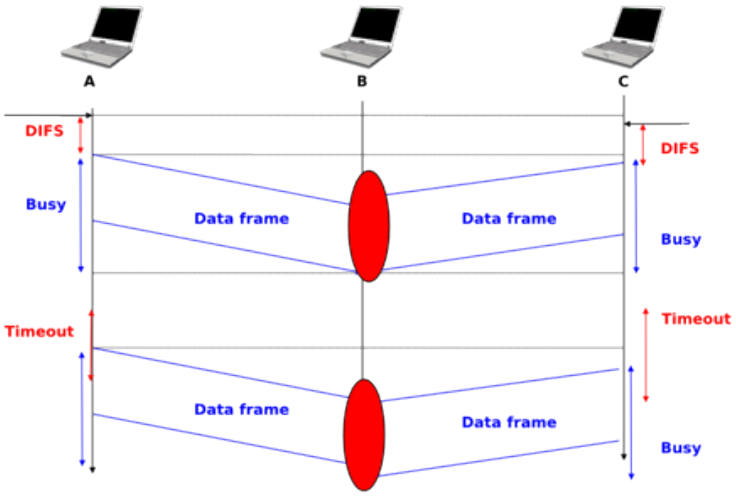


Figure 2.74: Collisions with CSMA/CA

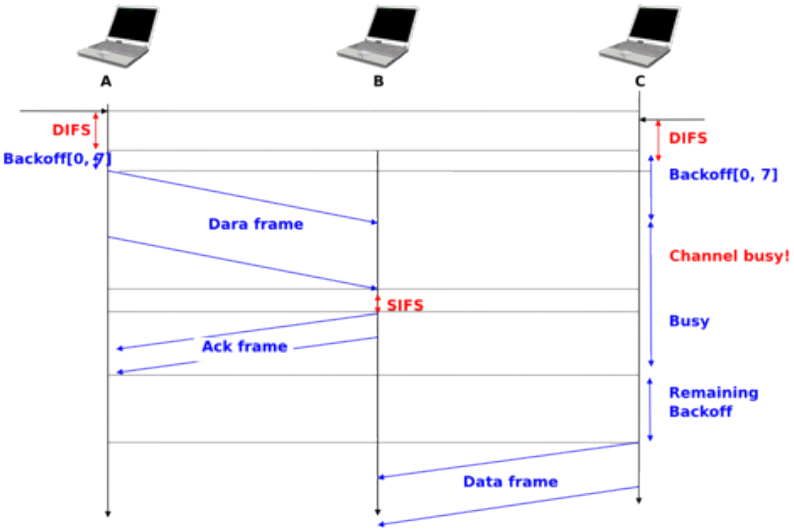


Figure 2.75: Detailed example with CSMA/CA

The pseudo-code below summarizes the operation of a CSMA/CA device. The values of the SIFS, DIFS, EIFS and slotTime depend on the underlying physical layer technology [802.11]

```
# CSMA/CA simplified pseudo-code
N=1
while N<= max :
   waitUntil(free(channel))
   if correct(last_frame) :
      wait(channel_free_during_t >=DIFS)
   else:
      wait(channel_free_during_t >=EIFS)

   back-off_time = int(random[0,min(255,7*(2^(N-1)))])*slotTime
   wait(channel free during backoff_time)
   # backoff timer is frozen while channel is sensed to be busy
   send(frame)
   wait(ack or timeout)
   if received(ack)
      # frame received correctly
      break
   else:
      # retransmission required
      N=N+1
```

Another problem faced by wireless networks is often called the *hidden station problem*. In a wireless network, radio signals are not always propagated same way in all directions. For example, two devices separated by a wall may not be able to receive each other's signal while they could both be receiving the signal produced by a third host. This is illustrated in the figure below, but it can happen in other environments. For example, two devices that are on different sides of a hill may not be able to receive each other's signal while they are both able to receive the signal sent by a station at the top of the hill. Furthermore, the radio propagation conditions may change with time. For example, a truck may temporarily block the communication between two nearby devices.

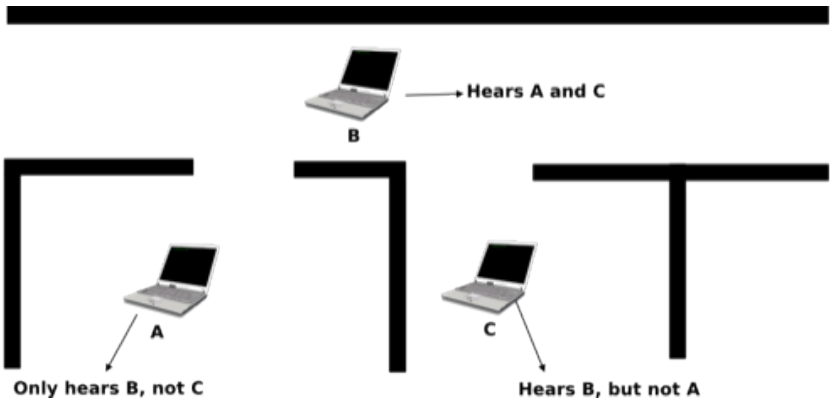


Figure 2.76: The hidden station problem

To avoid collisions in these situations, CSMA/CA allows devices to reserve the transmission channel for some time. This is done by using two control frames : *Request To Send* (RTS) and *Clear To Send* (CTS). Both are very short frames to minimize the risk of collisions. To reserve the transmission channel, a device sends a RTS frame to the intended recipient of the data frame. The RTS frame contains the duration of the requested reservation. The recipient replies, after a SIFS delay, with a CTS frame which also contains the duration of the reservation. As the duration of the reservation has been sent in both RTS and CTS, all hosts that could collide with either the sender or the reception of the data frame are informed of the reservation. They can compute the total duration of the transmission and defer their access to the transmission channel until then. This is illustrated in the figure below where host *A* reserves the transmission channel to send a data frame to host *B*. Host *C* notices the reservation and defers its transmission.

The utilization of the reservations with CSMA/CA is an optimisation that is useful when collisions are frequent. If there are few collisions, the time required to transmit the RTS and CTS frames can become significant and in

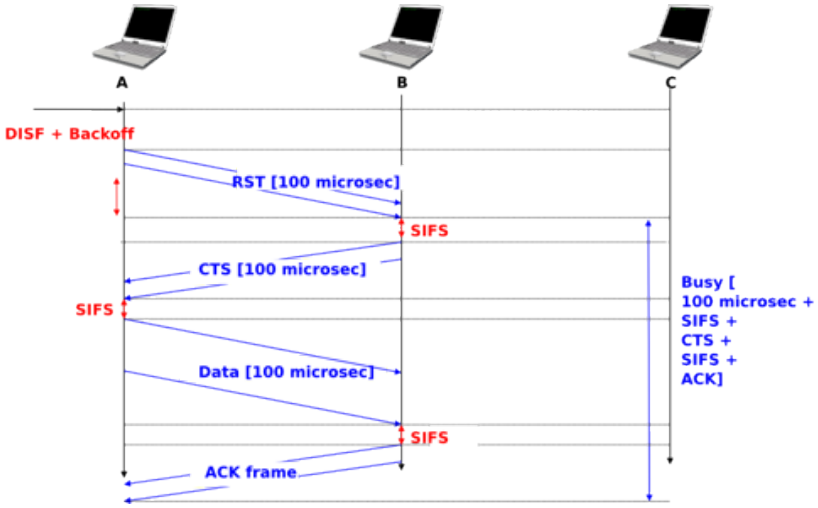


Figure 2.77: Reservations with CSMA/CA

particular when short frames are exchanged. Some devices only turn on RTS/CTS after transmission errors.

Deterministic Medium Access Control algorithms

During the 1970s and 1980s, there were huge debates in the networking community about the best suited Medium Access Control algorithms for Local Area Networks. The optimistic algorithms that we have described until now were relatively easy to implement when they were designed. From a performance perspective, mathematical models and simulations showed the ability of these optimistic techniques to sustain load. However, none of the optimistic techniques are able to guarantee that a frame will be delivered within a given delay bound and some applications require predictable transmission delays. The deterministic MAC algorithms were considered by a fraction of the networking community as the best solution to fulfill the needs of Local Area Networks.

Both the proponents of the deterministic and the opportunistic techniques lobbied to develop standards for Local Area networks that would incorporate their solution. Instead of trying to find an impossible compromise between these diverging views, the IEEE 802 committee that was chartered to develop Local Area Network standards chose to work in parallel on three different LAN technologies and created three working groups. The IEEE 802.3 working group became responsible for CSMA/CD. The proponents of deterministic MAC algorithms agreed on the basic principle of exchanging special frames called tokens between devices to regulate the access to the transmission medium. However, they did not agree on the most suitable physical layout for the network. IBM argued in favor of Ring-shaped networks while the manufacturing industry, led by General Motors, argued in favor of a bus-shaped network. This led to the creation of the *IEEE 802.4 working group* to standardise Token Bus networks and the IEEE 802.5 working group to standardise Token Ring networks. Although these techniques are not widely used anymore today, the principles behind a token-based protocol are still important.

The IEEE 802.5 Token Ring technology is defined in [802.5]. We use Token Ring as an example to explain the principles of the token-based MAC algorithms in ring-shaped networks. Other ring-shaped networks include the almost defunct FDDI [Ross1989] or the more recent Resilient Pack Ring [DYGU2004] . A good survey of the token ring networks may be found in [Bux1989] .

A Token Ring network is composed of a set of stations that are attached to a unidirectional ring. The basic principle of the Token Ring MAC algorithm is that two types of frames travel on the ring : tokens and data frames. When the Token Ring starts, one of the stations sends the token. The token is a small frame that represents the authorization to transmit data frames on the ring. To transmit a data frame on the ring, a station must first capture the token by removing it from the ring. As only one station can capture the token at a time, the station that owns the token can safely transmit a data frame on the ring without risking collisions. After having transmitted its frame, the station must remove it from the ring and resend the token so that other stations can transmit their own frames.

While the basic principles of the Token Ring are simple, there are several subtle implementation details that add complexity to Token Ring networks. To understand these details let us analyse the operation of a Token Ring interface on a station. A Token Ring interface serves three different purposes. Like other LAN interfaces, it must

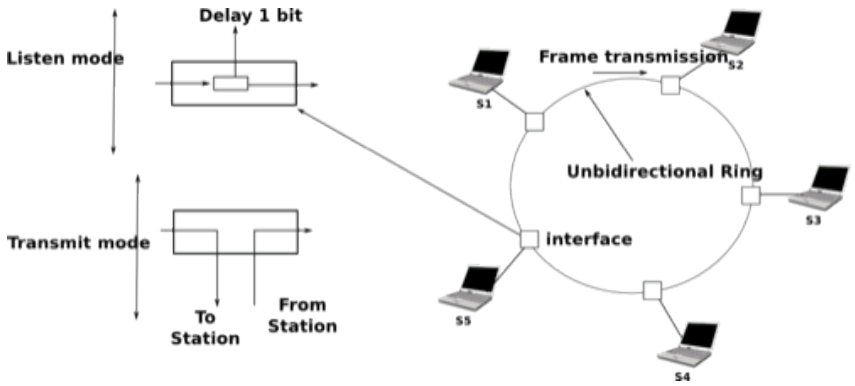


Figure 2.78: A Token Ring network

be able to send and receive frames. In addition, a Token Ring interface is part of the ring, and as such, it must be able to forward the electrical signal that passes on the ring even when its station is powered off.

When powered-on, Token Ring interfaces operate in two different modes : *listen* and *transmit*. When operating in *listen* mode, a Token Ring interface receives an electrical signal from its upstream neighbour on the ring, introduces a delay equal to the transmission time of one bit on the ring and regenerates the signal before sending it to its downstream neighbour on the ring.

The first problem faced by a Token Ring network is that as the token represents the authorization to transmit, it must continuously travel on the ring when no data frame is being transmitted. Let us assume that a token has been produced and sent on the ring by one station. In Token Ring networks, the token is a 24 bits frame whose structure is shown below.

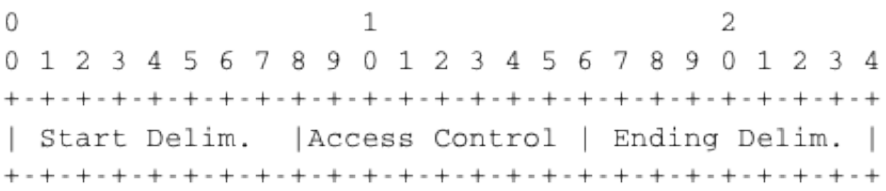


Figure 2.79: 802.5 token format

The token is composed of three fields. First, the *Starting Delimiter* is the marker that indicates the beginning of a frame. The first Token Ring networks used Manchester coding and the *Starting Delimiter* contained both symbols representing *0* and symbols that do not represent bits. The last field is the *Ending Delimiter* which marks the end of the token. The *Access Control* field is present in all frames, and contains several flags. The most important is the *Token* bit that is set in token frames and reset in other frames.

Let us consider the five station network depicted in figure *A Token Ring network* above and assume that station *S1* sends a token. If we neglect the propagation delay on the inter-station links, as each station introduces a one bit delay, the first bit of the frame would return to *S1* while it sends the fifth bit of the token. If station *S1* is powered off at that time, only the first five bits of the token will travel on the ring. To avoid this problem, there is a special station called the *Monitor* on each Token Ring. To ensure that the token can travel forever on the ring, this *Monitor* inserts a delay that is equal to at least 24 bit transmission times. If station *S3* was the *Monitor* in figure *A Token Ring network*, *S1* would have been able to transmit the entire token before receiving the first bit of the token from its upstream neighbor.

Now that we have explained how the token can be forwarded on the ring, let us analyse how a station can capture a token to transmit a data frame. For this, we need some information about the format of the data frames. An 802.5 data frame begins with the *Starting Delimiter* followed by the *Access Control* field whose *Token* bit is reset, a *Frame Control* field that allows for the definition of several types of frames, destination and source address, a

payload, a CRC, the *Ending Delimiter* and a *Frame Status* field. The format of the Token Ring data frames is illustrated below.

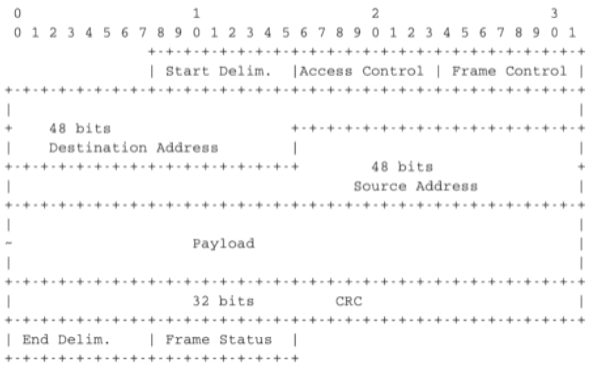


Figure 2.80: 802.5 data frame format

To capture a token, a station must operate in *Listen* mode. In this mode, the station receives bits from its upstream neighbour. If the bits correspond to a data frame, they must be forwarded to the downstream neighbour. If they correspond to a token, the station can capture it and transmit its data frame. Both the data frame and the token are encoded as a bit string beginning with the *Starting Delimiter* followed by the *Access Control* field. When the station receives the first bit of a *Starting Delimiter*, it cannot know whether this is a data frame or a token and must forward the entire delimiter to its downstream neighbour. It is only when it receives the fourth bit of the *Access Control* field (i.e. the *Token* bit) that the station knows whether the frame is a data frame or a token. If the *Token* bit is reset, it indicates a data frame and the remaining bits of the data frame must be forwarded to the downstream station. Otherwise (*Token* bit is set), this is a token and the station can capture it by resetting the bit that is currently in its buffer. Thanks to this modification, the beginning of the token is now the beginning of a data frame and the station can switch to *Transmit* mode and send its data frame starting at the fifth bit of the *Access Control* field. Thus, the one-bit delay introduced by each Token Ring station plays a key role in enabling the stations to efficiently capture the token.

After having transmitted its data frame, the station must remain in *Transmit* mode until it has received the last bit of its own data frame. This ensures that the bits sent by a station do not remain in the network forever. A data frame sent by a station in a Token Ring network passes in front of all stations attached to the network. Each station can detect the data frame and analyse the destination address to possibly capture the frame.

The text above describes the basic operation of a Token Ring network when all stations work correctly. Unfortunately, a real Token Ring network must be able to handle various types of anomalies and this increases the complexity of Token Ring stations. We briefly list the problems and outline their solutions below. A detailed description of the operation of Token Ring stations may be found in [802.5]. The first problem is when all the stations attached to the network start. One of them must bootstrap the network by sending the first token. For this, all stations implement a distributed election mechanism that is used to select the *Monitor*. Any station can become a *Monitor*. The *Monitor* manages the Token Ring network and ensures that it operates correctly. Its first role is to introduce a delay of 24 bit transmission times to ensure that the token can travel smoothly on the ring. Second, the *Monitor* sends the first token on the ring. It must also verify that the token passes regularly. According to the Token Ring standard [802.5], a station cannot retain the token to transmit data frames for a duration longer than the *Token Holding Time* (THT) (slightly less than 10 milliseconds). On a network containing *N* stations, the *Monitor* must receive the token at least every $N \times THT$ seconds. If the *Monitor* does not receive a token during such a period, it cuts the ring for some time and then reinitialises the ring and sends a token.

Several other anomalies may occur in a Token Ring network. For example, a station could capture a token and be powered off before having resent the token. Another station could have captured the token, sent its data frame and be powered off before receiving all of its data frame. In this case, the bit string corresponding to the end of a frame would remain in the ring without being removed by its sender. Several techniques are defined in [802.5] to allow the *Monitor* to handle all these problems. If unfortunately, the *Monitor* fails, another station will be elected to become the new *Monitor*.

2.6.5 Congestion control

Most networks contain links having different bandwidth. Some hosts can use low bandwidth wireless networks. Some servers are attached via 10 Gbps interfaces and inter-router links may vary from a few tens of kilobits per second up to hundred Gbps. Despite these huge differences in performance, any host should be able to efficiently exchange segments with a high-end server.

To understand this problem better, let us consider the scenario shown in the figure below, where a server (*A*) attached to a *10 Mbps* link needs to reliably transfer segments to another computer (*C*) through a path that contains a *2 Mbps* link.

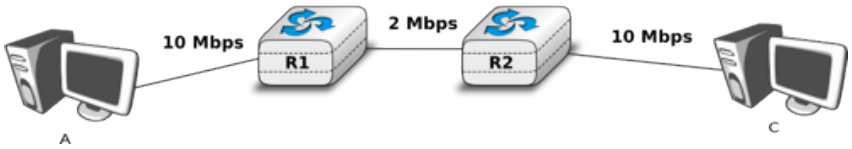


Figure 2.81: Reliable transport with heterogeneous links

In this network, the segments sent by the server reach router *R1*. *R1* forwards the segments towards router *R2*. Router *R1* can potentially receive segments at *10 Mbps*, but it can only forward them at *2 Mbps* to router *R2* and then to host *C*. Router *R1* includes buffers that allow it to store the packets that cannot immediately be forwarded to their destination. To understand the operation of a reliable transport protocol in this environment, let us consider a simplified model of this network where host *A* is attached to a *10 Mbps* link to a queue that represents the buffers of router *R1*. This queue is emptied at a rate of *2 Mbps*.

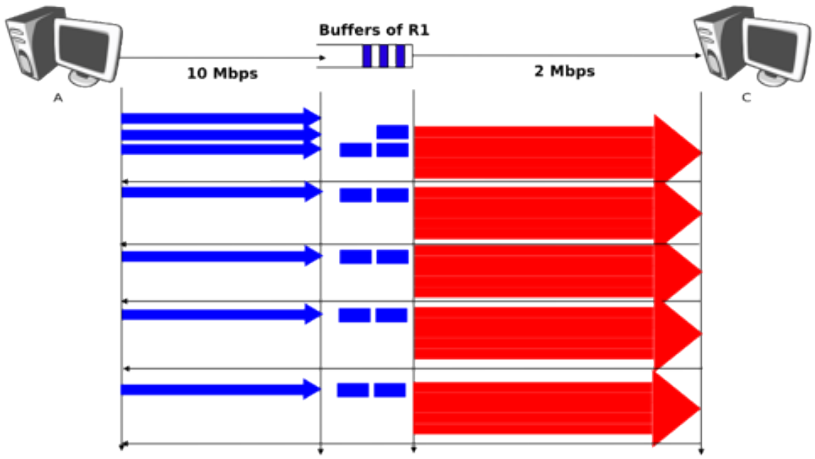


Figure 2.82: Self clocking

Let us consider that host *A* uses a window of three segments. It thus sends three back-to-back segments at *10 Mbps* and then waits for an acknowledgement. Host *A* stops sending segments when its window is full. These segments reach the buffers of router *R2*. The first segment stored in this buffer is sent by router *R2* at a rate of *2 Mbps* to the destination host. Upon reception of this segment, the destination sends an acknowledgement. This acknowledgement allows host *A* to transmit a new segment. This segment is stored in the buffers of router *R2* while it is transmitting the second segment that was sent by host *A*... Thus, after the transmission of the first window of segments, the reliable transport protocol sends one data segment after the reception of each acknowledgement returned by the destination. In practice, the acknowledgements sent by the destination serve as a kind of *clock* that allows the sending host to adapt its transmission rate to the rate at which segments are received by the

destination. This *self-clocking* is the first mechanism that allows a window-based reliable transport protocol to adapt to heterogeneous networks [Jacobson1988]. It depends on the availability of buffers to store the segments that have been sent by the sender but have not yet been transmitted to the destination.

However, transport protocols are not only used in this environment. In the global Internet, a large number of hosts send segments to a large number of receivers. For example, let us consider the network depicted below which is similar to the one discussed in [Jacobson1988] and **RFC 896**. In this network, we assume that the buffers of the router are infinite to ensure that no packet is lost.

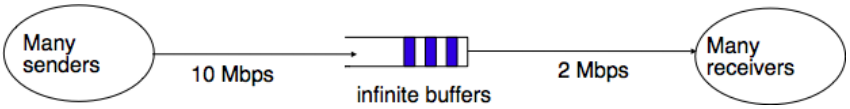


Figure 2.83: The congestion collapse problem

If many senders are attached to the left part of the network above, they all send a window full of segments. These segments are stored in the buffers of the router before being transmitted towards their destination. If there are many senders on the left part of the network, the occupancy of the buffers quickly grows. A consequence of the buffer occupancy is that the round-trip-time, measured by the transport protocol, between the sender and the receiver increases. Consider a network where 10,000 bits segments are sent. When the buffer is empty, such a segment requires 1 millisecond to be transmitted on the *10 Mbps* link and 5 milliseconds to be the transmitted on the *2 Mbps* link. Thus, the measured round-trip-time measured is roughly 6 milliseconds if we ignore the propagation delay on the links. If the buffer contains 100 segments, the round-trip-time becomes $1 + 100 \times 5 + 5$ milliseconds as new segments are only transmitted on the *2 Mbps* link once all previous segments have been transmitted. Unfortunately, if the reliable transport protocol uses a retransmission timer and performs *go-back-n* to recover from transmission errors it will retransmit a full window of segments. This increases the occupancy of the buffer and the delay through the buffer... Furthermore, the buffer may store and send on the low bandwidth links several retransmissions of the same segment. This problem is called *congestion collapse*. It occurred several times during the late 1980s on the Internet [Jacobson1988].

The *congestion collapse* is a problem that all heterogeneous networks face. Different mechanisms have been proposed in the scientific literature to avoid or control network congestion. Some of them have been implemented and deployed in real networks. To understand this problem in more detail, let us first consider a simple network with two hosts attached to a high bandwidth link that are sending segments to destination *C* attached to a low bandwidth link as depicted below.

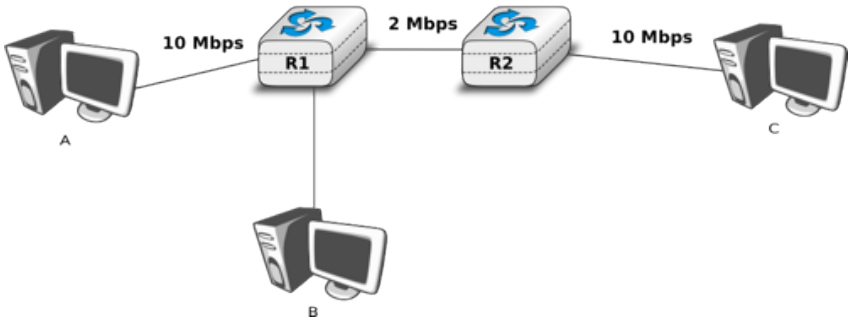


Figure 2.84: The congestion problem

To avoid *congestion collapse*, the hosts must regulate their transmission rate [17] by using a *congestion control* mechanism. Such a mechanism can be implemented in the transport layer or in the network layer. In TCP/IP networks, it is implemented in the transport layer, but other technologies such as *Asynchronous Transfer Mode (ATM)* or *Frame Relay* include congestion control mechanisms in lower layers.

[17] In this section, we focus on congestion control mechanisms that regulate the transmission rate of the hosts. Other types of mechanisms have been proposed in the literature. For example, *credit-based* flow-control has been proposed to avoid congestion in ATM networks [KR1995]. With a credit-based mechanism, hosts can only send packets once they have received credits from the routers and the credits depend on the occupancy of the router's buffers.

Let us first consider the simple problem of a set of i hosts that share a single bottleneck link as shown in the example above. In this network, the congestion control scheme must achieve the following objectives [CJ1989] :

1. The congestion control scheme must *avoid congestion*. In practice, this means that the bottleneck link cannot be overloaded. If $r_i(t)$ is the transmission rate allocated to host i at time t and R the bandwidth of the bottleneck link, then the congestion control scheme should ensure that, on average, $\forall t \sum r_i(t) \leq R$.
2. The congestion control scheme must be *efficient*. The bottleneck link is usually both a shared and an expensive resource. Usually, bottleneck links are wide area links that are much more expensive to upgrade than the local area networks. The congestion control scheme should ensure that such links are efficiently used. Mathematically, the control scheme should ensure that $\forall t \sum r_i(t) \approx R$.
3. The congestion control scheme should be *fair*. Most congestion schemes aim at achieving *max-min fairness*. An allocation of transmission rates to sources is said to be *max-min fair* if :

- no link in the network is congested
- the rate allocated to source j cannot be increased without decreasing the rate allocated to a source i whose allocation is smaller than the rate allocated to source j [Leboudec2008] .

Depending on the network, a *max-min fair allocation* may not always exist. In practice, *max-min fairness* is an ideal objective that cannot necessarily be achieved. When there is a single bottleneck link as in the example above, *max-min fairness* implies that each source should be allocated the same transmission rate.

To visualise the different rate allocations, it is useful to consider the graph shown below. In this graph, we plot on the *x-axis* (resp. *y-axis*) the rate allocated to host *B* (resp. *A*). A point in the graph (r_B, r_A) corresponds to a possible allocation of the transmission rates. Since there is a *2 Mbps* bottleneck link in this network, the graph can be divided into two regions. The lower left part of the graph contains all allocations (r_B, r_A) such that the bottleneck link is not congested ($r_A + r_B < 2$). The right border of this region is the *efficiency line*, i.e. the set of allocations that completely utilise the bottleneck link ($r_A + r_B = 2$). Finally, the *fairness line* is the set of fair allocations.

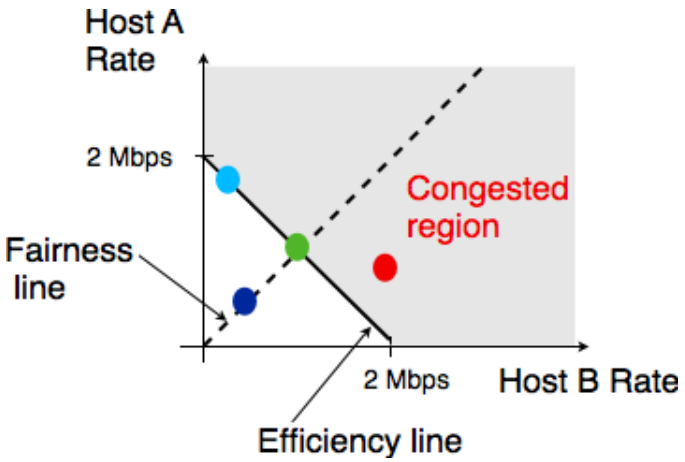


Figure 2.85: Possible allocated transmission rates

As shown in the graph above, a rate allocation may be fair but not efficient (e.g. $r_A = 0.7, r_B = 0.7$), fair and efficient (e.g. $r_A = 1, r_B = 1$) or efficient but not fair (e.g. $r_A = 1.5, r_B = 0.5$). Ideally, the allocation should be both fair and efficient. Unfortunately, maintaining such an allocation with fluctuations in the number of flows that use the network is a challenging problem. Furthermore, there might be several thousands flows that pass through the same link [18].

To deal with these fluctuations in demand, which result in fluctuations in the available bandwidth, computer networks use a congestion control scheme. This congestion control scheme should achieve the three objectives

[18] For example, the measurements performed in the Sprint network in 2004 reported more than 10k active TCP connections on a link, see https://research.sprintlabs.com/packstat/packetoverview.php. More recent information about backbone links may be obtained from caida 's realtime measurements, see e.g. http://www.caida.org/data/realtime/passive/

listed above. Some congestion control schemes rely on a close cooperation between the endhosts and the routers, while others are mainly implemented on the endhosts with limited support from the routers.

A congestion control scheme can be modelled as an algorithm that adapts the transmission rate ($r_i(t)$) of host i based on the feedback received from the network. Different types of feedbacks are possible. The simplest scheme is a binary feedback [CJ1989] [Jacobson1988] where the hosts simply learn whether the network is congested or not. Some congestion control schemes allow the network to regularly send an allocated transmission rate in Mbps to each host [BF1995].

Let us focus on the binary feedback scheme which is the most widely used today. Intuitively, the congestion control scheme should decrease the transmission rate of a host when congestion has been detected in the network, in order to avoid congestion collapse. Furthermore, the hosts should increase their transmission rate when the network is not congested. Otherwise, the hosts would not be able to efficiently utilise the network. The rate allocated to each host fluctuates with time, depending on the feedback received from the network. The figure below illustrates the evolution of the transmission rates allocated to two hosts in our simple network. Initially, two hosts have a low allocation, but this is not efficient. The allocations increase until the network becomes congested. At this point, the hosts decrease their transmission rate to avoid congestion collapse. If the congestion control scheme works well, after some time the allocations should become both fair and efficient.

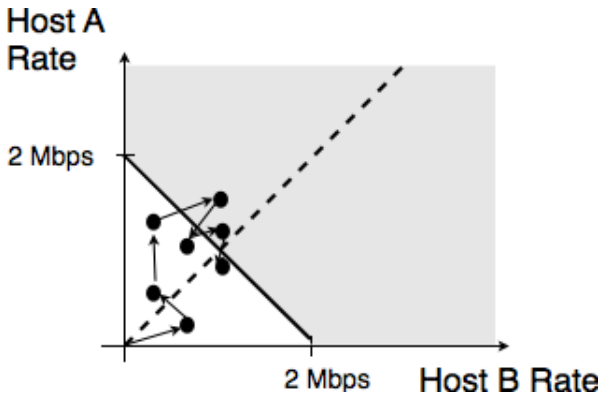


Figure 2.86: Evolution of the transmission rates

Various types of rate adaption algorithms are possible. Dah Ming Chiu and Raj Jain have analysed, in [CJ1989], different types of algorithms that can be used by a source to adapt its transmission rate to the feedback received from the network. Intuitively, such a rate adaptation algorithm increases the transmission rate when the network is not congested (ensure that the network is efficiently used) and decrease the transmission rate when the network is congested (to avoid congestion collapse).

The simplest form of feedback that the network can send to a source is a binary feedback (the network is congested or not congested). In this case, a *linear* rate adaptation algorithm can be expressed as :

- $rate(t+1) = \alpha_C + \beta_C rate(t)$ when the network is congested
- $rate(t+1) = \alpha_N + \beta_N rate(t)$ when the network is *not* congested

With a linear adaption algorithm, $\alpha_C, \alpha_N, \beta_C$ and β_N are constants. The analysis of [CJ1989] shows that to be fair and efficient, such a binary rate adaption mechanism must rely on *Additive Increase and Multiplicative Decrease*. When the network is not congested, the hosts should slowly increase their transmission rate ($\beta_N = 1 \; and \; \alpha_N > 0$). When the network is congested, the hosts must multiplicatively decrease their transmission rate ($\beta_C < 1 \; and \; \alpha_C = 0$). Such an AIMD rate adaptation algorithm can be implemented by the pseudo-code below.

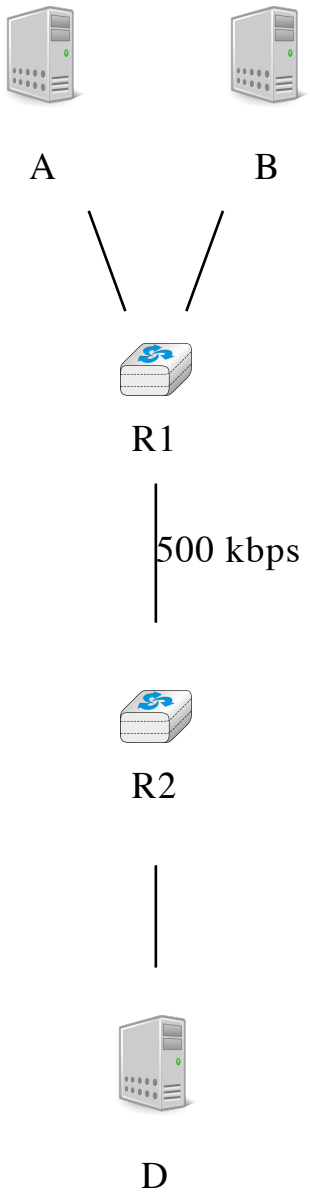
```
# Additive Increase Multiplicative Decrease
if congestion :
   rate=rate*betaC    # multiplicative decrease, betaC<1
else
   rate=rate+alphaN    # additive increase, v0>0
```

Note: Which binary feedback ?

Two types of binary feedback are possible in computer networks. A first solution is to rely on implicit feedback. This is the solution chosen for TCP. TCP's congestion control scheme [Jacobson1988] does not require any cooperation from the router. It only assumes that they use buffers and that they discard packets when there is congestion. TCP uses the segment losses as an indication of congestion. When there are no losses, the network is assumed to be not congested. This implies that congestion is the main cause of packet losses. This is true in wired networks, but unfortunately not always true in wireless networks. Another solution is to rely on explicit feedback. This is the solution proposed in the DECBit congestion control scheme [RJ1995] and used in Frame Relay and ATM networks. This explicit feedback can be implemented in two ways. A first solution would be to define a special message that could be sent by routers to hosts when they are congested. Unfortunately, generating such messages may increase the amount of congestion in the network. Such a congestion indication packet is thus discouraged **RFC 1812**. A better approach is to allow the intermediate routers to indicate, in the packets that they forward, their current congestion status. Binary feedback can be encoded by using one bit in the packet header. With such a scheme, congested routers set a special bit in the packets that they forward while non-congested routers leave this bit unmodified. The destination host returns the congestion status of the network in the acknowledgements that it sends. Details about such a solution in IP networks may be found in **RFC 3168**. Unfortunately, as of this writing, this solution is still not deployed despite its potential benefits.

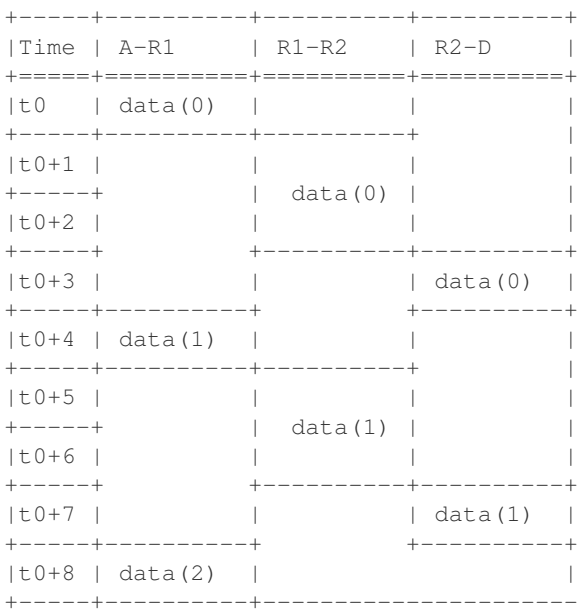
Congestion control in a window-based transport protocol

AIMD controls congestion by adjusting the transmission rate of the sources in reaction to the current congestion level. If the network is not congested, the transmission rate increases. If congestion is detected, the transmission rate is multiplicatively decreased. In practice, directly adjusting the transmission rate can be difficult since it requires the utilisation of fine grained timers. In reliable transport protocols, an alternative is to dynamically adjust the sending window. This is the solution chosen for protocols like TCP and SCTP that will be described in more details later. To understand how window-based protocols can adjust their transmission rate, let us consider the very simple scenario of a reliable transport protocol that uses *go-back-n*. Consider the very simple scenario shown in the figure below.

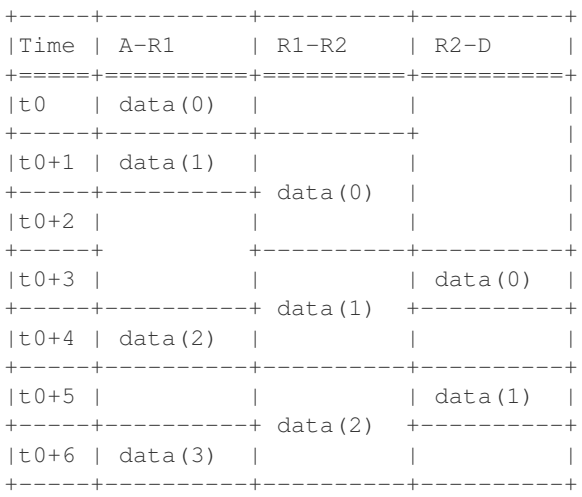


The links between the hosts and the routers have a bandwidth of 1 Mbps while the link between the two routers has a bandwidth of 500 Kbps. There is no significant propagation delay in this network. For simplicity, assume that hosts *A* and *B* send 1000 bits packets. The transmission of such a packet on a *host-router* (resp. *router-router*) link requires 1 msec (resp. 2 msec). If there is no traffic in the network, round-trip-time measured by host *A* is slightly larger than 4 msec. Let us observe the flow of packets with different window sizes to understand the relationship between sending window and transmission rate.

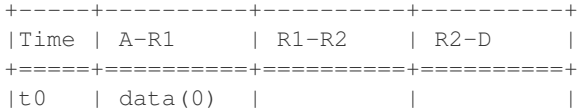
Consider first a window of one segment. This segment takes 4 msec to reach host *D*. The destination replies with an acknowledgement and the next segment can be transmitted. With such a sending window, the transmission rate is roughly 250 segments per second of 250 Kbps.

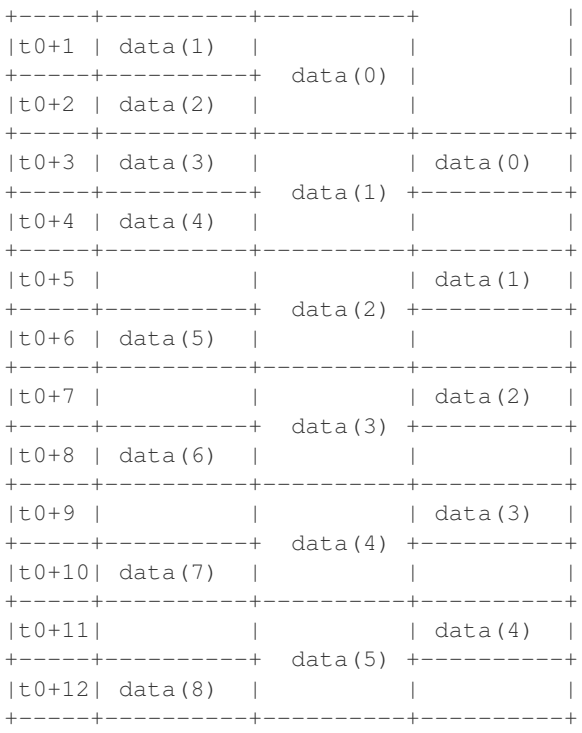


Consider now a window of two segments. Host *A* can send two segments within 2 msec on its 1 Mbps link. If the first segment is sent at time t_0, it reaches host *D* at $t_0 + 4$. Host *D* replies with an acknowledgement that opens the sending window on host *A* and enables it to transmit a new segment. In the meantime, the second segment was buffered by router *R1*. It reaches host *D* at $t_0 + 6$ and an acknowledgement is returned. With a window of two segments, host *A* transmits at roughly 500 Kbps, i.e. the transmission rate of the bottleneck link.



Our last example is a window of four segments. These segments are sent at t_0, $t_0 + 1$, $t_0 + 2$ and $t_0 + 3$. The first segment reaches host *D* at $t_0 + 4$. Host *D* replies to this segment by sending an acknowledgement that enables host *A* to transmit its fifth segment. This segment reaches router *R1* at $t_0 + 5$. At that time, router *R1* is transmitting the third segment to router *R2* and the fourth segment is still in its buffers. At time $t_0 + 6$, host *D* receives the second segment and returns the corresponding acknowledgement. This acknowledgement enables host *A* to send its sixth segment. This segment reaches router *R1* at roughly $t_0 + 7$. At that time, the router starts to transmit the fourth segment to router *R2*. Since link *R1-R2* can only sustain 500 Kbps, packets will accumulate in the buffers of *R1*. On average, there will be two packets waiting in the buffers of *R1*. The presence of these two packets will induce an increase of the round-trip-time as measured by the transport protocol. While the first segment was acknowledged within 4 msec, the fifth segment (*data(4)*) that was transmitted at time $t_0 + 4$ is only acknowledged at time $t_0 + 11$. On average, the sender transmits at 500 Kbps, but the utilisation of a large window induces a longer delay through the network.





From the above example, we can adjust the transmission rate by adjusting the sending window of a reliable transport protocol. A reliable transport protocol cannot send data faster than $\frac{window}{rtt}$ where $window$ is current sending window. To control the transmission rate, we introduce a *congestion window*. This congestion window limits the sending window. A any time, the sending window is restricted to $min(swin, cwin)$, where *swin* is the sending window and *cwin* the current *congestion window*. Of course, the window is further constrained by the receive window advertised by the remote peer. With the utilization of a congestion window, a simple reliable transport protocol that uses fixed size segments could implement *AIMD* as follows.

For the *Additive Increase* part our simple protocol would simply increase its *congestion window* by one segment every round-trip-time. The *Multiplicative Decrease* part of *AIMD* could be implemented by halving the congestion window when congestion is detected. For simplicity, we assume that congestion is detected thanks to a binary feedback and that no segments are lost. We will discuss in more details how losses affect a real transport protocol like TCP.

A congestion control scheme for our simple transport protocol could be implemented as follows.

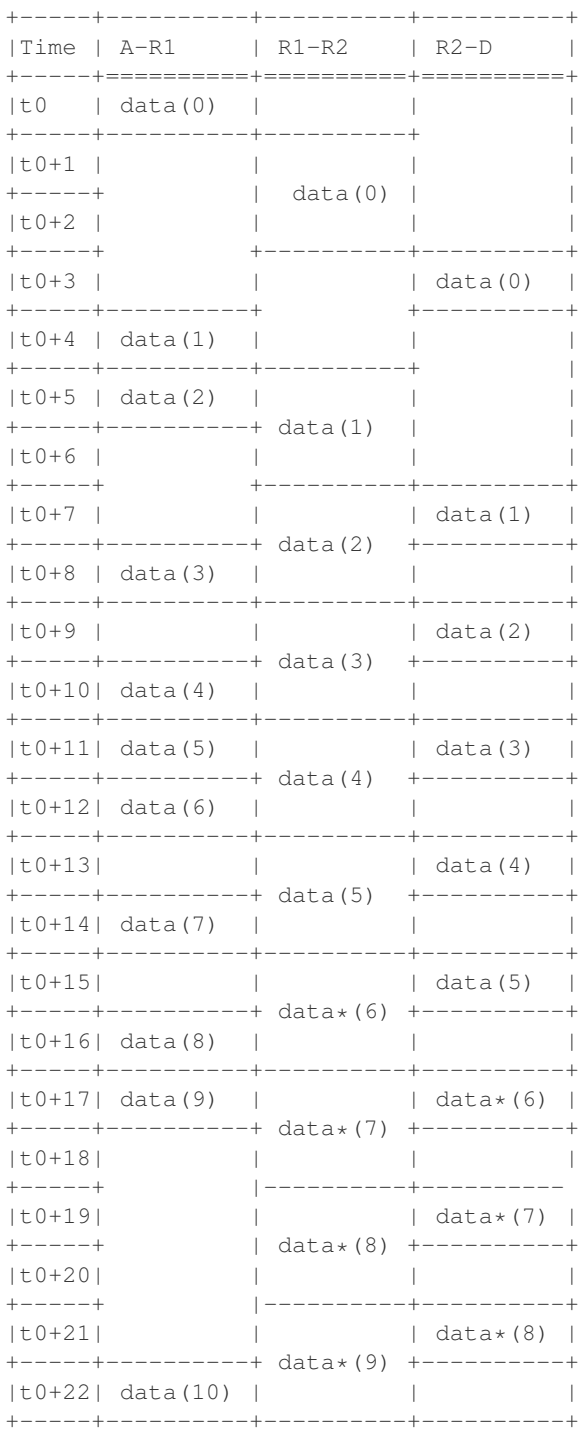
```
# Initialisation
cwin = 1  # congestion window measured in segments

# Ack arrival
if newack :  # new ack, no congestion
   # increase cwin by one every rtt
   cwin = cwin+ (1/cwin)
else:
   # no increase

Congestion detected:
   cwnd=cwin/2 # only once per rtt
```

In the above pseudocode, *cwin* contains the congestion window stored as a real in segments. This congestion window is updated upon the arrival of each acknowledgment and when congestion is detected. For simplicity, we assume that *cwin* is stored as a floating point number but only full segments can be transmitted.

As an illustration, let us consider the network scenario above and assume that the router implements the DECBit binary feedback scheme [RJ1995]. This scheme uses a form of Forward Explicit Congestion Notification and a router marks the congestion bit in arriving packets when its buffer contains one or more packets. In the figure below, we use a * to indicate a marked packet.



When the connection starts, its congestion window is set to one segment. Segment *data(0)* is sent at acknowledgment at roughly $t_0 + 4$. The congestion window is increased by one segment and *data(1)* and *data(2)* are transmitted at time $t_0 + 4$ and $t_0 + 5$. The corresponding acknowledgements are received at times $t_0 + 8$ and $t_0 + 10$. Upon reception of this last acknowledgement, the congestion window reaches *3* and segments can be sent (*data(4)* and *data(5)*). When segment *data(6)* reaches router *R1*, its buffers already contain *data(5)*. The packet containing *data(6)* is thus marked to inform the sender of the congestion. Note that the sender will only notice the congestion once it receives the corresponding acknowledgement at $t_0 + 18$. In the meantime, the congestion window continues to increase. At $t_0 + 16$, upon reception of the acknowledgement for *data(5)*, it reaches *4*. When congestion is detected, the congestion window is decreased down to *2*. This explains the idle time between the reception of the acknowledgement for *data*(6)* and the transmission of *data(10)*.

2.7 The reference models

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=5

Given the growing complexity of computer networks, during the 1970s network researchers proposed various reference models to facilitate the description of network protocols and services. Of these, the Open Systems Interconnection (OSI) model [Zimmermann80] was probably the most influential. It served as the basis for the standardisation work performed within the *ISO* to develop global computer network standards. The reference model that we use in this book can be considered as a simplified version of the OSI reference model [19].

2.7.1 The five layers reference model

Our reference model is divided into five layers, as shown in the figure below.

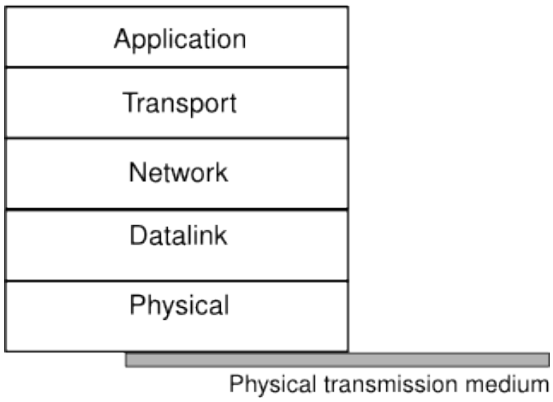


Figure 2.87: The five layers of the reference model

2.7.2 The Physical layer

Starting from the bottom, the first layer is the Physical layer. Two communicating devices are linked through a physical medium. This physical medium is used to transfer an electrical or optical signal between two directly connected devices.

An important point to note about the Physical layer is the service that it provides. This service is usually an unreliable connection-oriented service that allows the users of the Physical layer to exchange bits. The unit of information transfer in the Physical layer is the bit. The Physical layer service is unreliable because :

- the Physical layer may change, e.g. due to electromagnetic interferences, the value of a bit being transmitted
- the Physical layer may deliver *more* bits to the receiver than the bits sent by the sender
- the Physical layer may deliver *fewer* bits to the receiver than the bits sent by the sender

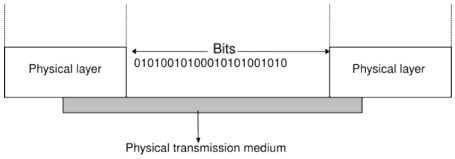


Figure 2.88: The Physical layer

[19] An interesting historical discussion of the OSI-TCP/IP debate may be found in [Russel06]

2.7.3 The Datalink layer

The *Datalink layer* builds on the service provided by the underlying physical layer. The *Datalink layer* allows two hosts that are directly connected through the physical layer to exchange information. The unit of information exchanged between two entities in the *Datalink layer* is a frame. A frame is a finite sequence of bits. Some *Datalink layers* use variable-length frames while others only use fixed-length frames. Some *Datalink layers* provide a connection-oriented service while others provide a connectionless service. Some *Datalink layers* provide reliable delivery while others do not guarantee the correct delivery of the information.

An important point to note about the *Datalink layer* is that although the figure below indicates that two entities of the *Datalink layer* exchange frames directly, in reality this is slightly different. When the *Datalink layer* entity on the left needs to transmit a frame, it issues as many *Data.request* primitives to the underlying *physical layer* as there are bits in the frame. The physical layer will then convert the sequence of bits in an electromagnetic or optical signal that will be sent over the physical medium. The *physical layer* on the right hand side of the figure will decode the received signal, recover the bits and issue the corresponding *Data.indication* primitives to its *Datalink layer* entity. If there are no transmission errors, this entity will receive the frame sent earlier.

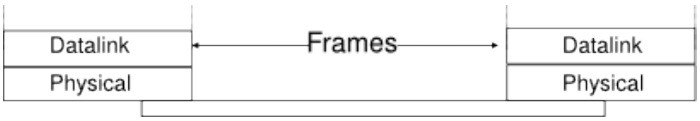


Figure 2.89: The Datalink layer

2.7.4 The Network layer

The *Datalink layer* allows directly connected hosts to exchange information, but it is often necessary to exchange information between hosts that are not attached to the same physical medium. This is the task of the *network layer*. The *network layer* is built above the *datalink layer*. Network layer entities exchange *packets*. A *packet* is a finite sequence of bytes that is transported by the datalink layer inside one or more frames. A packet usually contains information about its origin and its destination, and usually passes through several intermediate devices called routers on its way from its origin to its destination.

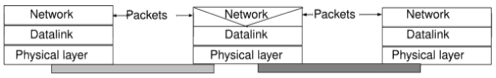


Figure 2.90: The network layer

2.7.5 The Transport layer

Most realisations of the network layer, including the internet, do not provide a reliable service. However, many applications need to exchange information reliably and so using the network layer service directly would be very difficult for them. Ensuring the reliable delivery of the data produced by applications is the task of the *transport layer*. *Transport layer* entities exchange *segments*. A segment is a finite sequence of bytes that are transported inside one or more packets. A transport layer entity issues segments (or sometimes part of segments) as *Data.request* to the underlying network layer entity.

There are different types of transport layers. The most widely used transport layers on the Internet are *TCP* ,that provides a reliable connection-oriented bytestream transport service, and *UDP* ,that provides an unreliable connection-less transport service.

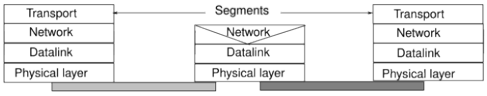


Figure 2.91: The transport layer

2.7.6 The Application layer

The upper layer of our architecture is the *Application layer*. This layer includes all the mechanisms and data structures that are necessary for the applications. We will use Application Data Unit (ADU) or the generic Service Data Unit (SDU) term to indicate the data exchanged between two entities of the Application layer.

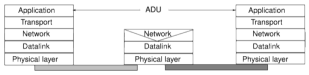


Figure 2.92: The Application layer

In the remaining chapters of this text, we will often refer to the information exchanged between entities located in different layers. To avoid any confusion, we will stick to the terminology defined earlier, i.e. :

- physical layer entities exchange bits
- datalink layer entities exchange *frames*
- network layer entities exchange *packets*
- transport layer entities exchange *segments*
- application layer entities exchange *SDUs*

2.7.7 Reference models

Two reference models have been successful in the networking community : the OSI reference model and the TCP/IP reference model. We discuss them briefly in this section.

The TCP/IP reference model

In contrast with OSI, the TCP/IP community did not spend a lot of effort defining a detailed reference model; in fact, the goals of the Internet architecture were only documented after TCP/IP had been deployed [Clark88]. **RFC 1122** , which defines the requirements for Internet hosts, mentions four different layers. Starting from the top, these are :

- the Application layer
- the Transport layer
- the Internet layer which is equivalent to the network layer of our reference model
- the Link layer which combines the functionalities of the physical and datalink layers of our five-layer reference model

Besides this difference in the lower layers, the TCP/IP reference model is very close to the five layers that we use throughout this document.

The OSI reference model

Compared to the five layers reference model explained above, the *OSI* reference model defined in [X200] is divided in seven layers. The four lower layers are similar to the four lower layers described above. The OSI reference model refined the application layer by dividing it in three layers :

- the *Session layer*. The Session layer contains the protocols and mechanisms that are necessary to organize and to synchronize the dialogue and to manage the data exchange of presentation layer entities. While one of the main functions of the transport layer is to cope with the unreliability of the network layer, the session's layer objective is to hide the possible failures of transport-level connections to the upper layer higher. For this, the Session Layer provides services that allow to establish a session-connection, to support orderly data exchange (including mechanisms that allow to recover from the abrupt release of an underlying transport connection), and to release the connection in an orderly manner.

- the *Presentation layer* was designed to cope with the different ways of representing information on computers. There are many differences in the way computer store information. Some computers store integers as 32 bits field, others use 64 bits field and the same problem arises with floating point number. For textual information, this is even more complex with the many different character codes that have been used [20]. The situation is even more complex when considering the exchange of structured information such as database records. To solve this problem, the Presentation layer contains provides for a common representation of the data transferred. The *ASN.1* notation was designed for the Presentation layer and is still used today by some protocols.
- the *Application layer* that contains the mechanisms that do not fit in neither the Presentation nor the Session layer. The OSI Application layer was itself further divided in several generic service elements.

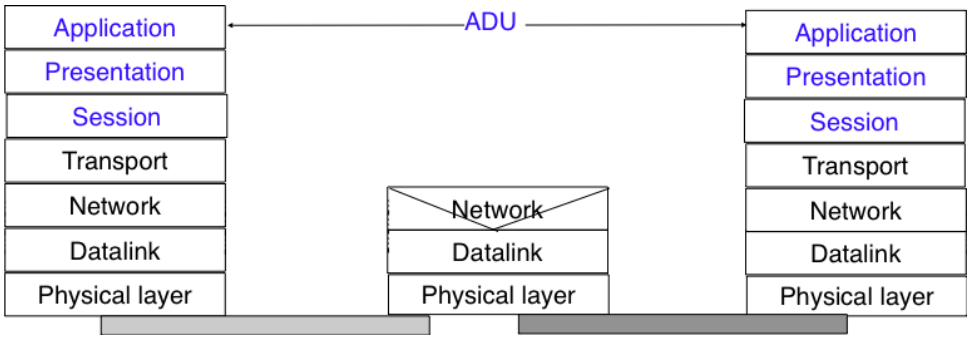


Figure 2.93: The seven layers of the OSI reference model

[20] There is now a rough consensus for the greater use of the Unicode character format. Unicode can represent more than 100,000 different characters from the known written languages on Earth. Maybe one day, all computers will only use Unicode to represent all their stored characters and Unicode could become the standard format to exchange characters, but we are not yet at this stage today.

CHAPTER 3

Part 2: Protocols

3.1 The application layer

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=5

Networked applications rely on the transport service. As explained earlier, there are two main types of transport services :

- the *connectionless* service
- the *connection-oriented* or *byte-stream* service

The connectionless service allows applications to easily exchange messages or Service Data Units. On the Internet, this service is provided by the UDP protocol that will be explained in the next chapter. The connectionless transport service on the Internet is unreliable, but is able to detect transmission errors. This implies that an application will not receive data that has been corrupted due to transmission errors.

The connectionless transport service allows networked application to exchange messages. Several networked applications may be running at the same time on a single host. Each of these applications must be able to exchange SDUs with remote applications. To enable these exchanges of SDUs, each networked application running on a host is identified by the following information :

- the *host* on which the application is running
- the *port number* on which the application *listens* for SDUs

On the Internet, the *port number* is an integer and the *host* is identified by its network address. There are two types of Internet Addresses :

- *IP version 4* addresses that are 32 bits wide
- *IP version 6* addresses that are 128 bits wide

IPv4 addresses are usually represented by using a dotted decimal representation where each decimal number corresponds to one byte of the address, e.g. *203.0.113.56*. IPv6 addresses are usually represented as a set of hexadecimal numbers separated by semicolons, e.g. *2001:db8:3080:2:217:f2ff:fed6:65c0*. Today, most Internet hosts have one IPv4 address. A small fraction of them also have an IPv6 address. In the future, we can expect that more and more hosts will have IPv6 addresses and that some of them will not have an IPv4 address anymore. A host that only has an IPv4 address cannot communicate with a host having only an IPv6 address. The figure below illustrates two that are using the datagram service provided by UDP on hosts that are using IPv4 addresses.

Note: Textual representation of IPv6 addresses

It is sometimes necessary to write IPv6 addresses in text format, e.g. when manually configuring addresses or for documentation purposes. The preferred format for writing IPv6 addresses is *x:x:x:x:x:x:x:x*, where the *x* 's are hexadecimal digits representing the eight 16-bit parts of the address. Here are a few examples of IPv6 addresses :

- abcd:Eef01:2345:6789:abcd:ef01:2345:6789

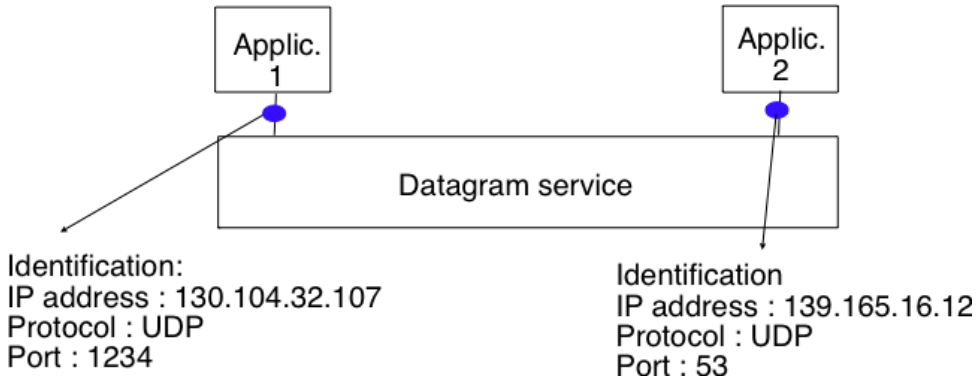


Figure 3.1: The connectionless or datagram service

- 2001:db8:0:0:8:800:200c:417a
- fe80:0:0:0:219:e3ff:fed7:1204

IPv6 addresses often contain a long sequence of bits set to *0*. In this case, a compact notation has been defined. With this notation, *::* is used to indicate one or more groups of 16 bits blocks containing only bits set to *0*. For example,

- 2001:db8:0:0:8:800:200c:417a is represented as *2001:db8::8:800:200c:417a*
- ff01:0:0:0:0:0:0:101 is represented as *ff01::101*
- 0:0:0:0:0:0:0:1 is represented as *::1*
- 0:0:0:0:0:0:0:0 is represented as *::*

The second transport service is the connection-oriented service. On the Internet, this service is often called the *byte-stream service* as it creates a reliable byte stream between the two applications that are linked by a transport connection. Like the datagram service, the networked applications that use the byte-stream service are identified by the host on which they run and a port number. These hosts can be identified by an address or a name. The figure below illustrates two applications that are using the byte-stream service provided by the TCP protocol on IPv6 hosts. The byte stream service provided by TCP is reliable and bidirectional.

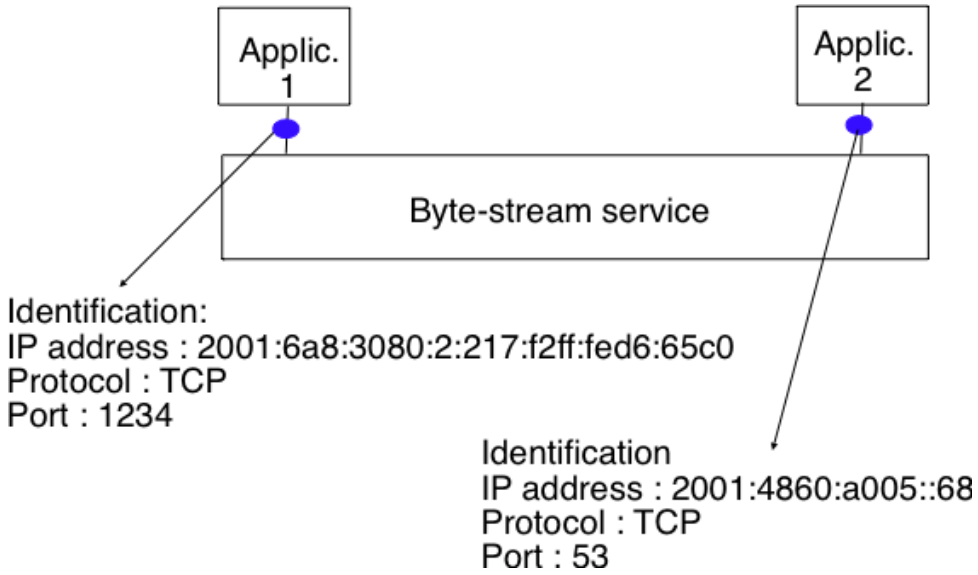


Figure 3.2: The connection-oriented or byte-stream service

3.2 The Domain Name System

We have already explained the main principles that underlie the utilisation of names on the Internet and their mapping to addresses. The last component of the Domain Name System is the DNS protocol. The DNS protocol runs above both the datagram service and the bytestream services. In practice, the datagram service is used when short queries and responses are exchanged, and the bytestream service is used when longer responses are expected. In this section, we will only discuss the utilisation of the DNS protocol above the datagram service. This is the most frequent utilisation of the DNS.

DNS messages are composed of five parts that are named sections in **RFC 1035**. The first three sections are mandatory and the last two sections are optional. The first section of a DNS message is its *Header*. It contains information about the type of message and the content of the other sections. The second section contains the *Question* sent to the name server or resolver. The third section contains the *Answer* to the *Question*. When a client sends a DNS query, the *Answer* section is empty. The fourth section, named *Authority*, contains information about the servers that can provide an authoritative answer if required. The last section contains additional information that is supplied by the resolver or server but was not requested in the question.

The header of DNS messages is composed of 12 bytes and its structure is shown in the figure below.

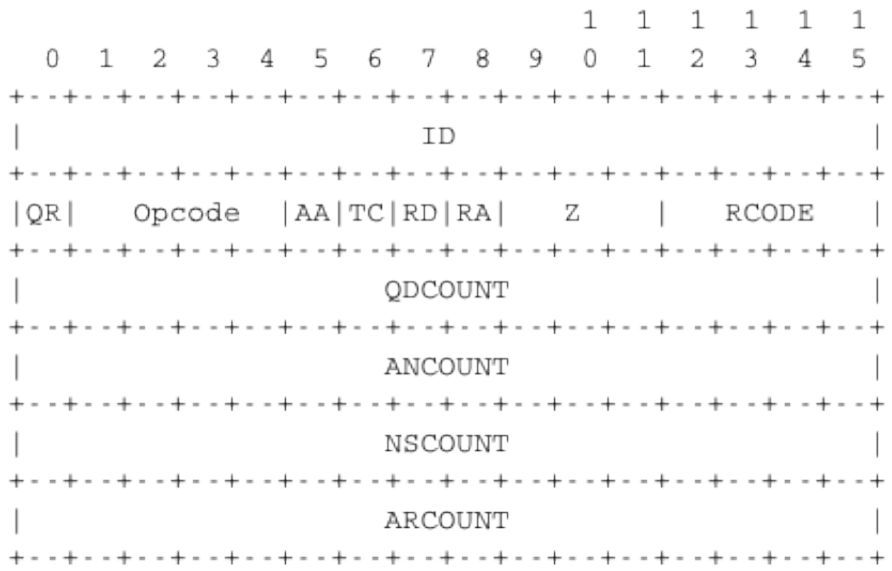


Figure 3.3: DNS header

The *ID* (identifier) is a 16-bits random value chosen by the client. When a client sends a question to a DNS server, it remembers the question and its identifier. When a server returns an answer, it returns in the *ID* field the identifier chosen by the client. Thanks to this identifier, the client can match the received answer with the question that it sent.

The *QR* flag is set to *0* in DNS queries and *1* in DNS answers. The *Opcode* is used to specify the type of query. For instance, a *standard query* is when a client sends a *name* and the server returns the corresponding *data* and an update request is when the client sends a *name* and new *data* and the server then updates its database.

The *AA* bit is set when the server that sent the response has *authority* for the domain name found in the question section. In the original DNS deployments, two types of servers were considered : *authoritative* servers and *non-authoritative* servers. The *authoritative* servers are managed by the system administrators responsible for a given domain. They always store the most recent information about a domain. *Non-authoritative* servers are servers or resolvers that store DNS information about external domains without being managed by the owners of a domain. They may thus provide answers that are out of date. From a security point of view, the *authoritative* bit is not an absolute indication about the validity of an answer. Securing the Domain Name System is a complex problem that was only addressed satisfactorily recently by the utilisation of cryptographic signatures in the DNSSEC extensions to DNS described in **RFC 4033**. However, these extensions are outside the scope of this chapter.

The *RD* (recursion desired) bit is set by a client when it sends a query to a resolver. Such a query is said to be *recursive* because the resolver will recurse through the DNS hierarchy to retrieve the answer on behalf of the client. In the past, all resolvers were configured to perform recursive queries on behalf of any Internet host. However, this exposes the resolvers to several security risks. The simplest one is that the resolver could become overloaded by having too many recursive queries to process. As of this writing, most resolvers [1] only allow recursive queries from clients belonging to their company or network and discard all other recursive queries. The *RA* bit indicates whether the server supports recursion. The *RCODE* is used to distinguish between different types of errors. See **RFC 1035** for additional details. The last four fields indicate the size of the *Question*, *Answer*, *Authority* and *Additional* sections of the DNS message.

The last four sections of the DNS message contain *Resource Records* (RR). All RRs have the same top level format shown in the figure below.

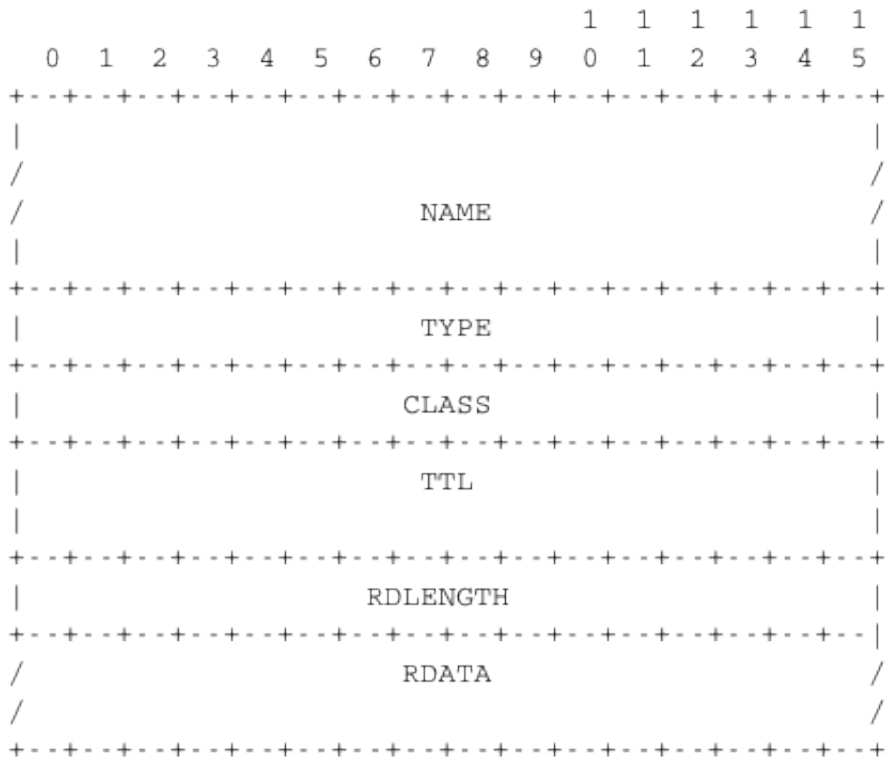


Figure 3.4: DNS Resource Records

In a *Resource Record* (*RR*), the *Name* indicates the name of the node to which this resource record pertains. The two bytes *Type* field indicate the type of resource record. The *Class* field was used to support the utilisation of the DNS in other environments than the Internet.

The *TTL* field indicates the lifetime of the *Resource Record* in seconds. This field is set by the server that returns an answer and indicates for how long a client or a resolver can store the *Resource Record* inside its cache. A long *TTL* indicates a stable *RR*. Some companies use short *TTL* values for mobile hosts and also for popular servers. For example, a web hosting company that wants to spread the load over a pool of hundred servers can configure its nameservers to return different answers to different clients. If each answer has a small *TTL*, the clients will be forced to send DNS queries regularly. The nameserver will reply to these queries by supplying the address of the less loaded server.

The *RDLength* field is the length of the *RData* field that contains the information of the type specified in the *Type* field.

Several types of DNS RR are used in practice. The *A* type is used to encode the IPv4 address that corresponds to the specified name. The *AAAA* type is used to encode the IPv6 address that corresponds to the specified name. A

[1] Some DNS resolvers allow any host to send queries. Google operates a public DNS resolver at addresses *2001:4860:4860::8888* and *2001:4860:4860::8844*

NS record contains the name of the DNS server that is responsible for a given domain. For example, a query for the *AAAA* record associated to the *www.ietf.org* name returns the following answer.

```
; <<>> DiG 9.6-ESV-R4-P3 <<>> -t AAAA www.ietf.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 55279
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 6, ADDITIONAL: 11

;; QUESTION SECTION:
;www.ietf.org.                  IN      AAAA

;; ANSWER SECTION:
www.ietf.org.           1661    IN      AAAA    2001:1890:123a::1:1e

;; AUTHORITY SECTION:
ietf.org.               1661    IN      NS      ns1.hkg1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns1.sea1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns0.ams1.com.
ietf.org.               1661    IN      NS      ns1.yyz1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns1.ams1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns1.mia1.afilias-nst.info.

;; ADDITIONAL SECTION:
ns1.sea1.afilias-nst.info. 70552 IN     A       65.22.8.1
ns1.sea1.afilias-nst.info. 74301 IN     AAAA    2a01:8840:8::1
ns1.yyz1.afilias-nst.info. 70552 IN     A       65.22.9.1
ns1.yyz1.afilias-nst.info. 74301 IN     AAAA    2a01:8840:9::1
ns1.ams1.afilias-nst.info. 2597 IN      A       65.22.6.79
ns0.ams1.com.           148523  IN      A       64.170.98.2
ns0.ams1.com.           148523  IN      AAAA    2001:1890:126c::1:2
ns1.hkg1.afilias-nst.info. 2073 IN      A       65.22.6.1
ns1.hkg1.afilias-nst.info. 2073 IN      AAAA    2a01:8840:6::1
ns1.mia1.afilias-nst.info. 70552 IN     A       65.22.7.1
ns1.mia1.afilias-nst.info. 74301 IN     AAAA    2a01:8840:7::1

;; Query time: 16 msec
;; SERVER: 109.88.203.3#53(109.88.203.3)
;; WHEN: Tue Oct  8 20:59:02 2013
;; MSG SIZE  rcvd: 451
```

Figure 3.5: Query for the *AAAA* record of *www.ietf.org*

This answer contains several pieces of information. First, the name *www.ietf.org* is associated to IP address *2001:1890:123a::1:1e*. Second, the *ietf.org* domain is managed by six different nameservers. Five of these nameservers are reachable via IPv4 and IPv6.

CNAME (or canonical names) are used to define aliases. For example *www.example.com* could be a *CNAME* for *pc12.example.com* that is the actual name of the server on which the web server for *www.example.com* runs.

Note: Reverse DNS

The DNS is mainly used to find the address that corresponds to a given name. However, it is sometimes useful to obtain the name that corresponds to an IP address. This done by using the *PTR* (*pointer*) *RR*. The *RData* part of a *PTR RR* contains the name while the *Name* part of the *RR* contains the IP address encoded in the *in-addr.arpa* domain. IPv4 addresses are encoded in the *in-addr.arpa* by reversing the four digits that compose the dotted decimal representation of the address. For example, consider IPv4 address *192.0.2.11*. The hostname associated to this address can be found by requesting the *PTR RR* that corresponds to *11.2.0.192.in-addr.arpa*. A similar solution is used to support IPv6 addresses **RFC 3596**, but slightly more complex given the length of the IPv6 addresses. For example, consider IPv6 address *2001:1890:123a::1:1e*. To obtain the name that corresponds to this address, we need first to convert it in a reverse dotted decimal notation : *e.1.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.a.3.2.1.0.9.8.1.1.0.0.2*. In this notation, each character between dots corresponds to one nibble, i.e. four bits. The low-order byte (*e*) appears first and the high order (*2*) last. To obtain the name that corresponds to this address, one needs to append the *ip6.arpa* domain name and query for *e.1.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.a.3.2.1.0.9.8.1.1.0.0.2.ip6.arpa*. In practice, tools and libraries do the conversion automatically and the user does not need to worry about it.

An important point to note regarding the Domain Name System is its extensibility. Thanks to the *Type* and *RDLength* fields, the format of the Resource Records can easily be extended. Furthermore, a DNS implementation that receives a new Resource Record that it does not understand can ignore the record while still being able to process the other parts of the message. This allows, for example, a DNS server that only supports IPv6 to ignore the IPv4 addresses listed in the DNS reply for *www.ietf.org* while still being able to correctly parse the Resource

Records that it understands. This extensibility allowed the Domain Name System to evolve over the years while still preserving the backward compatibility with already deployed DNS implementations.

3.3 Electronic mail

Electronic mail, or email, is a very popular application in computer networks such as the Internet. Email appeared in the early 1970s and allows users to exchange text based messages. Initially, it was mainly used to exchange short messages, but over the years its usage has grown. It is now not only used to exchange small, but also long messages that can be composed of several parts as we will see later.

Before looking at the details of Internet email, let us consider a simple scenario illustrated in the figure below, where Alice sends an email to Bob. Alice prepares her email by using an email clients and sends it to her email server. Alice's email server extracts Bob's address from the email and delivers the message to Bob's server. Bob retrieves Alice's message on his server and reads it by using his favourite email client or through his webmail interface.

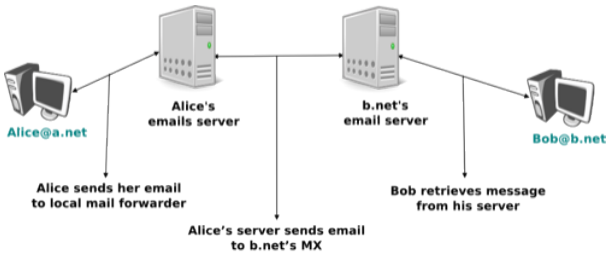


Figure 3.6: Simplified architecture of the Internet email

The email system that we consider in this book is composed of four components :

- a message format, that defines how valid email messages are encoded
- protocols, that allow hosts and servers to exchange email messages
- client software, that allows users to easily create and read email messages
- software, that allows servers to efficiently exchange email messages

We will first discuss the format of email messages followed by the protocols that are used on today's Internet to exchange and retrieve emails. Other email systems have been developed in the past [Bush1993] [Genilloud1990] [GC2000], but today most email solutions have migrated to the Internet email. Information about the software that is used to compose and deliver emails may be found on wikipedia among others, for both email clients and email servers. More detailed information about the full Internet Mail Architecture may be found in **RFC 5598**.

Email messages, like postal mail, are composed of two parts :

- a *header* that plays the same role as the letterhead in regular mail. It contains metadata about the message.
- the *body* that contains the message itself.

Email messages are entirely composed of lines of ASCII characters. Each line can contain up to 998 characters and is terminated by the *CR* and *LF* control characters **RFC 5322**. The lines that compose the *header* appear before the message *body*. An empty line, containing only the *CR* and *LF* characters, marks the end of the *header*. This is illustrated in the figure below.

The email header contains several lines that all begin with a keyword followed by a colon and additional information. The format of email messages and the different types of header lines are defined in **RFC 5322**. Two of these header lines are mandatory and must appear in all email messages :

- The sender address. This header line starts with *From:*. This contains the (optional) name of the sender followed by its email address between < and >. Email addresses are always composed of a username followed by the @ sign and a domain name.
- The date. This header line starts with *Date:*. **RFC 5322** precisely defines the format used to encode a date.

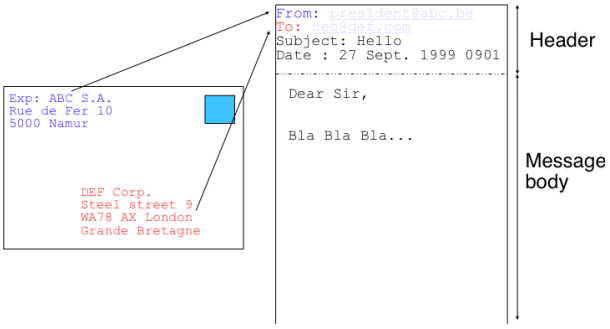


Figure 3.7: The structure of email messages

Other header lines appear in most email messages. The *Subject:* header line allows the sender to indicate the topic discussed in the email. Three types of header lines can be used to specify the recipients of a message :

- the *To:* header line contains the email addresses of the primary recipients of the message [2] . Several addresses can be separated by using commas.
- the *cc:* header line is used by the sender to provide a list of email addresses that must receive a carbon copy of the message. Several addresses can be listed in this header line, separated by commas. All recipients of the email message receive the *To:* and *cc:* header lines.
- the *bcc:* header line is used by the sender to provide a list of comma separated email addresses that must receive a blind carbon copy of the message. The *bcc:* header line is not delivered to the recipients of the email message.

A simple email message containing the *From:*, *To:*, *Subject:* and *Date:* header lines and two lines of body is shown below.

```
From: Bob Smith <Bob@machine.example>
To: Alice Doe <alice@example.net>, Alice Smith <Alice@machine.example>
Subject: Hello
Date: Mon, 8 Mar 2010 19:55:06 -0600

This is the "Hello world" of email messages.
This is the second line of the body
```

Note the empty line after the *Date:* header line; this empty line contains only the *CR* and *LF* characters, and marks the boundary between the header and the body of the message.

Several other optional header lines are defined in **RFC 5322** and elsewhere [3]. Furthermore, many email clients and servers define their own header lines starting from *X-*. Several of the optional header lines defined in **RFC 5322** are worth being discussed here :

- the *Message-Id:* header line is used to associate a "unique" identifier to each email. Email identifiers are usually structured like *string@domain* where *string* is a unique character string or sequence number chosen by the sender of the email and *domain* the domain name of the sender. Since domain names are unique, a host can generate globally unique message identifiers concatenating a locally unique identifier with its domain name.
- the *In-reply-to:* is used when a message was created in reply to a previous message. In this case, the end of the *In-reply-to:* line contains the identifier of the original message.
- the *Received:* header line is used when an email message is processed by several servers before reaching its destination. Each intermediate email server adds a *Received:* header line. These header lines are useful to debug problems in delivering email messages.

The figure below shows the header lines of one email message. The message originated at a host named

[2] It could be surprising that the *To:* is not mandatory inside an email message. While most email messages will contain this header line an email that does not contain a *To:* header line and that relies on the *bcc:* to specify the recipient is valid as well.

[3] The list of all standard email header lines may be found at http://www.iana.org/assignments/message-headers/message-header-index.html

wira.firstpr.com.au and was received by *smtp3.sgsi.ucl.ac.be*. The *Received:* lines have been wrapped for readability.

```
Received: from smtp3.sgsi.ucl.ac.be (Unknown [10.1.5.3])
    by mmp.sipr-dc.ucl.ac.be
    (Sun Java(tm) System Messaging Server 7u3-15.01 64bit (built Feb 12 2010))
    with ESMTP id <0KYY00L85LI5JLE0@mmp.sipr-dc.ucl.ac.be>; Mon,
    08 Mar 2010 11:37:17 +0100 (CET)
Received: from mail.ietf.org (mail.ietf.org [64.170.98.32])
    by smtp3.sgsi.ucl.ac.be (Postfix) with ESMTP id B92351C60D7; Mon,
    08 Mar 2010 11:36:51 +0100 (CET)
Received: from [127.0.0.1] (localhost [127.0.0.1])     by core3.amsl.com (Postfix)
    with ESMTP id F066A3A68B9; Mon, 08 Mar 2010 02:36:38 -0800 (PST)
Received: from localhost (localhost [127.0.0.1])       by core3.amsl.com (Postfix)
    with ESMTP id A1E6C3A681B  for <rrg@core3.amsl.com>; Mon,
    08 Mar 2010 02:36:37 -0800 (PST)
Received: from mail.ietf.org ([64.170.98.32])
    by localhost (core3.amsl.com [127.0.0.1]) (amavisd-new, port 10024)
    with ESMTP id erw8ih2v8VQa for <rrg@core3.amsl.com>; Mon,
    08 Mar 2010 02:36:36 -0800 (PST)
Received: from gair.firstpr.com.au (gair.firstpr.com.au [150.101.162.123])
    by core3.amsl.com (Postfix) with ESMTP id 03E893A67ED      for <rrg@irtf.org>; Mon,
    08 Mar 2010 02:36:35 -0800 (PST)
Received: from [10.0.0.6] (wira.firstpr.com.au [10.0.0.6])
    by gair.firstpr.com.au (Postfix) with ESMTP id D0A49175B63; Mon,
    08 Mar 2010 21:36:37 +1100 (EST)
Date: Mon, 08 Mar 2010 21:36:38 +1100
From: Robin Whittle <rw@firstpr.com.au>
Subject: Re: [rrg] Recommendation and what happens next
In-reply-to: <C7B9C21A.4FAB%tony.li@tony.li>
To: RRG <rrg@irtf.org>
Message-id: <4B94D336.7030504@firstpr.com.au>

Message content removed
```

Initially, email was used to exchange small messages of ASCII text between computer scientists. However, with the growth of the Internet, supporting only ASCII text became a severe limitation for two reasons. First of all, non-English speakers wanted to write emails in their native language that often required more characters than those of the ASCII character table. Second, many users wanted to send other content than just ASCII text by email such as binary files, images or sound.

To solve this problem, the IETF developed the Multipurpose Internet Mail Extensions (*MIME*). These extensions were carefully designed to allow Internet email to carry non-ASCII characters and binary files without breaking the email servers that were deployed at that time. This requirement for backward compatibility forced the MIME designers to develop extensions to the existing email message format **RFC 822** instead of defining a completely new format that would have been better suited to support the new types of emails.

RFC 2045 defines three new types of header lines to support MIME :

- The *MIME-Version:* header indicates the version of the MIME specification that was used to encode the email message. The current version of MIME is 1.0. Other versions of MIME may be defined in the future. Thanks to this header line, the software that processes email messages will be able to adapt to the MIME version used to encode the message. Messages that do not contain this header are supposed to be formatted according to the original **RFC 822** specification.
- The *Content-Type:* header line indicates the type of data that is carried inside the message (see below)
- The *Content-Transfer-Encoding:* header line is used to specify how the message has been encoded. When MIME was designed, some email servers were only able to process messages containing characters encoded using the 7 bits ASCII character set. MIME allows the utilisation of other character encodings.

Inside the email header, the *Content-Type:* header line indicates how the MIME email message is structured. **RFC 2046** defines the utilisation of this header line. The two most common structures for MIME messages are :

- *Content-Type: multipart/mixed*. This header line indicates that the MIME message contains several independent parts. For example, such a message may contain a part in plain text and a binary file.
- *Content-Type: multipart/alternative*. This header line indicates that the MIME message contains several representations of the same information. For example, a *multipart/alternative* message may contain both a plain text and an HTML version of the same text.

To support these two types of MIME messages, the recipient of a message must be able to extract the different parts from the message. In **RFC 822**, an empty line was used to separate the header lines from the body. Using an empty line to separate the different parts of an email body would be difficult as the body of email messages often contains one or more empty lines. Another possible option would be to define a special line, e.g. **-LAST_LINE-** to mark the boundary between two parts of a MIME message. Unfortunately, this is not possible as some emails may contain this string in their body (e.g. emails sent to students to explain the format of MIME messages). To solve this problem, the *Content-Type:* header line contains a second parameter that specifies the string that has been used by the sender of the MIME message to delineate the different parts. In practice, this string is often chosen randomly by the mail client.

The email message below, copied from **RFC 2046** shows a MIME message containing two parts that are both in plain text and encoded using the ASCII character set. The string *simple boundary* is defined in the *Content-Type:* header as the marker for the boundary between two successive parts. Another example of MIME messages may be found in **RFC 2046**.

```
Date: Mon, 20 Sep 1999 16:33:16 +0200
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Test
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="simple boundary"

preamble, to be ignored

--simple boundary
Content-Type: text/plain; charset=us-ascii

First part

--simple boundary
Content-Type: text/plain; charset=us-ascii

Second part
--simple boundary
```

The *Content-Type:* header can also be used inside a MIME part. In this case, it indicates the type of data placed in this part. Each data type is specified as a type followed by a subtype. A detailed description may be found in **RFC 2046**. Some of the most popular *Content-Type:* header lines are :

- *text*. The message part contains information in textual format. There are several subtypes : *text/plain* for regular ASCII text, *text/html* defined in **RFC 2854** for documents in *HTML* format or the *text/enriched* format defined in **RFC 1896**. The *Content-Type:* header line may contain a second parameter that specifies the character set used to encode the text. *charset=us-ascii* is the standard ASCII character table. Other frequent character sets include *charset=UTF8* or *charset=iso-8859-1*. The list of standard character sets is maintained by *IANA*
- *image*. The message part contains a binary representation of an image. The subtype indicates the format of the image such as gif, jpg or png.
- *audio*. The message part contains an audio clip. The subtype indicates the format of the audio clip like wav or mp3
- *video*. The message part contains a video clip. The subtype indicates the format of the video clip like avi or mp4
- *application*. The message part contains binary information that was produced by the particular application listed as the subtype. Email clients use the subtype to launch the application that is able to decode the

received binary information.

Note: From ASCII to Unicode

The first computers used different techniques to represent characters in memory and on disk. During the 1960s, computers began to exchange information via tape or telephone lines. Unfortunately, each vendor had its own proprietary character set and exchanging data between computers from different vendors was often difficult. The 7 bits ASCII character table **RFC 20** set was adopted by several vendors and by many Internet protocols. However, ASCII became a problem with the internationalisation of the Internet and the desire of more and more users to use character sets that support their own written language. A first attempt at solving this problem was the definition of the ISO-8859 character sets by *ISO*. This family of standards specified various character sets that allowed the representation of many European written languages by using 8 bits characters. Unfortunately, an 8-bits character set is not sufficient to support some widely used languages, such as those used in Asian countries. Fortunately, at the end of the 1980s, several computer scientists proposed to develop a standard that supports all written languages used on Earth today. The Unicode standard [Unicode] has now been adopted by most computer and software vendors. For example, Java uses Unicode natively to manipulate characters, Python can handle both ASCII and Unicode characters. Internet applications are slowly moving towards complete support for the Unicode character sets, but moving from ASCII to Unicode is an important change that can have a huge impact on current deployed implementations. See for example, the work to completely internationalise email **RFC 4952** and domain names **RFC 5890**.

The last MIME header line is *Content-Transfer-Encoding:*. This header line is used after the *Content-Type:* header line, within a message part, and specifies how the message part has been encoded. The default encoding is to use 7 bits ASCII. The most frequent encodings are *quoted-printable* and *Base64*. Both support encoding a sequence of bytes into a set of ASCII lines that can be safely transmitted by email servers. *quoted-printable* is defined in **RFC 2045**. We briefly describe *base64* which is defined in **RFC 2045** and **RFC 4648**.

Base64 divides the sequence of bytes to be encoded into groups of three bytes (with the last group possibly being partially filled). Each group of three bytes is then divided into four six-bit fields and each six bit field is encoded as a character from the table below.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

The example below, from **RFC 4648**, illustrates the *Base64* encoding.

Input data	0x14fb9c03d97e
8-bit	00010100 11111011 10011100 00000011 11011001 01111110
6-bit	000101 001111 101110 011100 000000 111101 100101 111110
Decimal	5 15 46 28 0 61 37 62
Encoding	F P u c A 9 l +

The last point to be discussed about *base64* is what happens when the length of the sequence of bytes to be encoded is not a multiple of three. In this case, the last group of bytes may contain one or two bytes instead of three. *Base64* reserves the = character as a padding character. This character is used once when the last group

contains two bytes and twice when it contains one byte as illustrated by the two examples below.

Input data	0x14
8-bit	00010100
6-bit	000101 000000
Decimal	5 0
Encoding	F A = =

Input data	0x14b9
8-bit	00010100 11111011
6-bit	000101 001111 101100
Decimal	5 15 44
Encoding	F P s =

Now that we have explained the format of the email messages, we can discuss how these messages can be exchanged through the Internet. The figure below illustrates the protocols that are used when *Alice* sends an email message to *Bob*. *Alice* prepares her email with an email client or on a webmail interface. To send her email to *Bob*, *Alice*'s client will use the Simple Mail Transfer Protocol (*SMTP*) to deliver her message to her SMTP server. *Alice*'s email client is configured with the name of the default SMTP server for her domain. There is usually at least one SMTP server per domain. To deliver the message, *Alice*'s SMTP server must find the SMTP server that contains *Bob*'s mailbox. This can be done by using the Mail eXchange (MX) records of the DNS. A set of MX records can be associated to each domain. Each MX record contains a numerical preference and the fully qualified domain name of a SMTP server that is able to deliver email messages destined to all valid email addresses of this domain. The DNS can return several MX records for a given domain. In this case, the server with the lowest preference is used first. If this server is not reachable, the second most preferred server is used etc. *Bob*'s SMTP server will store the message sent by *Alice* until *Bob* retrieves it using a webmail interface or protocols such as the Post Office Protocol (*POP*) or the Internet Message Access Protocol (*IMAP*).

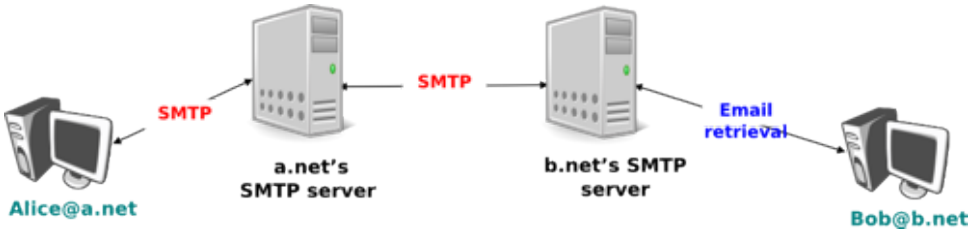


Figure 3.8: Email delivery protocols

3.3.1 The Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (*SMTP*) defined in **RFC 5321** is a client-server protocol. The SMTP specification distinguishes between five types of processes involved in the delivery of email messages. Email messages are composed on a Mail User Agent (MUA). The MUA is usually either an email client or a webmail. The MUA sends the email message to a Mail Submission Agent (MSA). The MSA processes the received email and forwards it to the Mail Transmission Agent (MTA). The MTA is responsible for the transmission of the email, directly or via intermediate MTAs to the MTA of the destination domain. This destination MTA will then forward the message to the Mail Delivery Agent (MDA) where it will be accessed by the recipient's MUA. SMTP is used for the interactions between MUA and MSA [4], MSA-MTA and MTA-MTA.

SMTP is a text-based protocol like many other application-layer protocols on the Internet. It relies on the byte-stream service. Servers listen on port *25*. Clients send commands that are each composed of one line of ASCII text terminated by *CR+LF*. Servers reply by sending ASCII lines that contain a three digit numerical error/success code and optional comments.

The SMTP protocol, like most text-based protocols, is specified as a *BNF*. The full BNF is defined in **RFC 5321**. The main SMTP commands are defined by the BNF rules shown in the figure below.

[4] During the last years, many Internet Service Providers, campus and enterprise networks have deployed SMTP extensions **RFC 4954** on their MSAs. These extensions force the MUAs to be authenticated before the MSA accepts an email message from the MUA.

```
helo = "HELO" SP Domain CRLF
mail = "MAIL FROM:" Path CRLF
rcpt = "RCPT TO:" ( "<Postmaster@" Domain ">" / "<Postmaster>" / Path ) CRLF
data = "DATA" CRLF
quit = "QUIT" CRLF
Path           = "<" Mailbox ">"
Domain         = sub-domain *("." sub-domain)
sub-domain     = Let-dig [Ldh-str]
Let-dig        = ALPHA / DIGIT
Ldh-str        = *( ALPHA / DIGIT / "-" ) Let-dig
Mailbox        = Local-part "@" Domain
Local-part     = Dot-string
Dot-string     = Atom *("."  Atom)
Atom           = 1*atext
```

Figure 3.9: BNF specification of the SMTP commands

In this BNF, *atext* corresponds to printable ASCII characters. This BNF rule is defined in **RFC 5322**. The five main commands are *EHLO*, *MAIL FROM:*, *RCPT TO:*, *DATA* and *QUIT* [5]. *Postmaster* is the alias of the system administrator who is responsible for a given domain or SMTP server. All domains must have a *Postmaster* alias.

The SMTP responses are defined by the BNF shown in the figure below.

```
Greeting       = "220 " Domain [ SP textstring ] CRLF
textstring     = 1*atext
Reply-line     = *( Reply-code "-" [ textstring ] CRLF )
                 Reply-code [ SP textstring ] CRLF
Reply-code     = %x32-35 %x30-35 %x30-39
```

Figure 3.10: BNF specification of the SMTP responses

SMTP servers use structured reply codes containing three digits and an optional comment. The first digit of the reply code indicates whether the command was successful or not. A reply code of *2xy* indicates that the command has been accepted. A reply code of *3xy* indicates that the command has been accepted, but additional information from the client is expected. A reply code of *4xy* indicates a transient negative reply. This means that for some reason, which is indicated by either the other digits or the comment, the command cannot be processed immediately, but there is some hope that the problem will only be transient. This is basically telling the client to try the same command again later. In contrast, a reply code of *5xy* indicates a permanent failure or error. In this case, it is useless for the client to retry the same command later. Other application layer protocols such as FTP **RFC 959** or HTTP **RFC 2616** use a similar structure for their reply codes. Additional details about the other reply codes may be found in **RFC 5321**.

Examples of SMTP reply codes include the following :

```
500  Syntax error, command unrecognized
501  Syntax error in parameters or arguments
502  Command not implemented
503  Bad sequence of commands
220  <domain> Service ready
221  <domain> Service closing transmission channel
421  <domain> Service not available, closing transmission channel
250  Requested mail action okay, completed
450  Requested mail action not taken: mailbox unavailable
452  Requested action not taken: insufficient system storage
550  Requested action not taken: mailbox unavailable
354  Start mail input; end with <CRLF>.<CRLF>
```

The first four reply codes correspond to errors in the commands sent by the client. The fourth reply code would be sent by the server when the client sends commands in an incorrect order (e.g. the client tries to send an email before providing the destination address of the message). Reply code *220* is used by the server as the first message when it agrees to interact with the client. Reply code *221* is sent by the server before closing the underlying transport connection. Reply code *421* is returned when there is a problem (e.g. lack of memory/disk resources) that prevents the server from accepting the transport connection. Reply code *250* is the standard positive reply that indicates the success of the previous command. Reply codes *450* and *452* indicate that the destination mailbox

[5] The first versions of SMTP used *HELO* as the first command sent by a client to a SMTP server. When SMTP was extended to support newer features such as 8 bits characters, it was necessary to allow a server to recognise whether it was interacting with a client that supported the extensions or not. *EHLO* became mandatory with the publication of **RFC 2821**.

is temporarily unavailable, for various reasons, while reply code *550* indicates that the mailbox does not exist or cannot be used for policy reasons. Reply code *354* indicates that the client can start transmitting its email message.

The transfer of an email message is performed in three phases. During the first phase, the client opens a transport connection with the server. Once the connection has been established, the client and the server exchange greetings messages (*EHLO* command). Most servers insist on receiving valid greeting messages and some of them drop the underlying transport connection if they do not receive a valid greeting. Once the greetings have been exchanged, the email transfer phase can start. During this phase, the client transfers one or more email messages by indicating the email address of the sender (*MAIL FROM:* command), the email address of the recipient (*RCPT TO:* command) followed by the headers and the body of the email message (*DATA* command). Once the client has finished sending all its queued email messages to the SMTP server, it terminates the SMTP association (*QUIT* command).

A successful transfer of an email message is shown below

```
S: 220 smtp.example.com ESMTP MTA information
C: EHLO mta.example.org
S: 250 Hello mta.example.org, glad to meet you
C: MAIL FROM:<alice@example.org>
S: 250 Ok
C: RCPT TO:<bob@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Alice Doe" <alice@example.org>
C: To: Bob Smith <bob@example.com>
C: Date: Mon, 9 Mar 2010 18:22:32 +0100
C: Subject: Hello
C:
C: Hello Bob
C: This is a small message containing 4 lines of text.
C: Best regards,
C: Alice
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

In the example above, the MTA running on *mta.example.org* opens a TCP connection to the SMTP server on host *smtp.example.com*. The lines prefixed with *S:* (resp. *C:*) are the responses sent by the server (resp. the commands sent by the client). The server sends its greetings as soon as the TCP connection has been established. The client then sends the *EHLO* command with its fully qualified domain name. The server replies with reply-code *250* and sends its greetings. The SMTP association can now be used to exchange an email.

To send an email, the client must first provide the address of the recipient with *RCPT TO:*. Then it uses the *MAIL FROM:* with the address of the sender. Both the recipient and the sender are accepted by the server. The client can now issue the *DATA* command to start the transfer of the email message. After having received the *354* reply code, the client sends the headers and the body of its email message. The client indicates the end of the message by sending a line containing only the . (dot) character [6]. The server confirms that the email message has been queued for delivery or transmission with a reply code of *250*. The client issues the *QUIT* command to close the session and the server confirms with reply-code *221*, before closing the TCP connection.

Note: Open SMTP relays and spam

Since its creation in 1971, email has been a very useful tool that is used by many users to exchange lots of information. In the early days, all SMTP servers were open and anyone could use them to forward emails towards their final destination. Unfortunately, over the years, some unscrupulous users have found ways to use email for marketing purposes or to send malware. The first documented abuse of email for marketing purposes occurred in 1978 when a marketer who worked for a computer vendor sent a marketing email to many ARPANET users. At that time, the ARPANET could only be used for research purposes and this was an abuse of the acceptable use policy. Unfortunately, given the extremely low cost of sending emails, the problem of unsolicited emails has not

[6] This implies that a valid email message cannot contain a line with one dot followed by *CR* and *LF*. If a user types such a line in an email, his email client will automatically add a space character before or after the dot when sending the message over SMTP.

stopped. Unsolicited emails are now called spam and a study carried out by ENISA in 2009 reveals that 95% of email was spam and this number seems to continue to grow. This places a burden on the email infrastructure of Internet Service Providers and large companies that need to process many useless messages.

Given the amount of spam messages, SMTP servers are no longer open **RFC 5068**. Several extensions to SMTP have been developed in recent years to deal with this problem. For example, the SMTP authentication scheme defined in **RFC 4954** can be used by an SMTP server to authenticate a client. Several techniques have also been proposed to allow SMTP servers to *authenticate* the messages sent by their users **RFC 4870** **RFC 4871** .

3.3.2 The Post Office Protocol

When the first versions of SMTP were designed, the Internet was composed of minicomputers that were used by an entire university department or research lab. These minicomputers were used by many users at the same time. Email was mainly used to send messages from a user on a given host to another user on a remote host. At that time, SMTP was the only protocol involved in the delivery of the emails as all hosts attached to the network were running an SMTP server. On such hosts, an email destined to local users was delivered by placing the email in a special directory or file owned by the user. However, the introduction of personal computers in the 1980s, changed this environment. Initially, users of these personal computers used applications such as *telnet* to open a remote session on the local *minicomputer* to read their email. This was not user-friendly. A better solution appeared with the development of user friendly email client applications on personal computers. Several protocols were designed to allow these client applications to retrieve the email messages destined to a user from his/her server. Two of these protocols became popular and are still used today. The Post Office Protocol (POP), defined in **RFC 1939**, is the simplest one. It allows a client to download all the messages destined to a given user from his/her email server. We describe POP briefly in this section. The second protocol is the Internet Message Access Protocol (IMAP), defined in **RFC 3501**. IMAP is more powerful, but also more complex than POP. IMAP was designed to allow client applications to efficiently access in real-time to messages stored in various folders on servers. IMAP assumes that all the messages of a given user are stored on a server and provides the functions that are necessary to search, download, delete or filter messages.

POP is another example of a simple line-based protocol. POP runs above the bytestream service. A POP server usually listens to port 110. A POP session is composed of three parts : an *authorisation* phase during which the server verifies the client's credential, a *transaction* phase during which the client downloads messages and an *update* phase that concludes the session. The client sends commands and the server replies are prefixed by *+OK* to indicate a successful command or by *-ERR* to indicate errors.

When a client opens a transport connection with the POP server, the latter sends as banner an ASCII-line starting with *+OK*. The POP session is at that time in the *authorisation* phase. In this phase, the client can send its username (resp. password) with the *USER* (resp. *PASS*) command. The server replies with *+OK* if the username (resp. password) is valid and *-ERR* otherwise.

Once the username and password have been validated, the POP session enters in the *transaction* phase. In this phase, the client can issue several commands. The *STAT* command is used to retrieve the status of the server. Upon reception of this command, the server replies with a line that contains *+OK* followed by the number of messages in the mailbox and the total size of the mailbox in bytes. The *RETR* command, followed by a space and an integer, is used to retrieve the nth message of the mailbox. The *DELE* command is used to mark for deletion the nth message of the mailbox.

Once the client has retrieved and possibly deleted the emails contained in the mailbox, it must issue the *QUIT* command. This command terminates the POP session and allows the server to delete all the messages that have been marked for deletion by using the *DELE* command.

The figure below provides a simple POP session. All lines prefixed with *C:* (resp. *S:*) are sent by the client (resp. server).

```
S:    +OK POP3 server ready
C:    USER alice
S:    +OK
C     PASS 12345pass
S:    +OK alice's maildrop has 2 messages (620 octets)
C:    STAT
```

```
S:    +OK 2 620
C:    LIST
S:    +OK 2 messages (620 octets)
S:    1 120
S:    2 500
S:    .
C:    RETR 1
S:    +OK 120 octets
S:    <the POP3 server sends message 1>
S:    .
C:    DELE 1
S:    +OK message 1 deleted
C:    QUIT
S:    +OK POP3 server signing off (1 message left)
```

In this example, a POP client contacts a POP server on behalf of the user named *alice*. Note that in this example, Alice's password is sent in clear by the client. This implies that if someone is able to capture the packets sent by Alice, he will know Alice's password [7]. Then Alice's client issues the *STAT* command to know the number of messages that are stored in her mailbox. It then retrieves and deletes the first message of the mailbox.

3.4 The HyperText Transfer Protocol

In the early days of the Internet was mainly used for remote terminal access with telnet, email and file transfer. The default file transfer protocol, *FTP*, defined in **RFC 959** was widely used and *FTP* clients and servers are still included in most operating systems.

Many *FTP* clients offer a user interface similar to a Unix shell and allow the client to browse the file system on the server and to send and retrieve files. *FTP* servers can be configured in two modes :

- authenticated : in this mode, the ftp server only accepts users with a valid user name and password. Once authenticated, they can access the files and directories according to their permissions
- anonymous : in this mode, clients supply the *anonymous* userid and their email address as password. These clients are granted access to a special zone of the file system that only contains public files.

ftp was very popular in the 1990s and early 2000s, but today it has mostly been superseded by more recent protocols. Authenticated access to files is mainly done by using the Secure Shell (ssh) protocol defined in **RFC 4251** and supported by clients such as scp or sftp. Nowadays, anonymous access is mainly provided by web protocols.

In the late 1980s, high energy physicists working at CERN had to efficiently exchange documents about their ongoing and planned experiments. Tim Berners-Lee evaluated several of the documents sharing techniques that were available at that time [B1989]. As none of the existing solutions met CERN's requirements, they chose to develop a completely new document sharing system. This system was initially called the *mesh*, but was quickly renamed the *world wide web*. The starting point for the *world wide web* are hypertext documents. An hypertext document is a document that contains references (hyperlinks) to other documents that the reader can immediately access. Hypertext was not invented for the world wide web. The idea of hypertext documents was proposed in 1945 [Bush1945] and the first experiments were done during the 1960s [Nelson1965] [Myers1998] . Compared to the hypertext documents that were used in the late 1980s, the main innovation introduced by the *world wide web* was to allow hyperlinks to reference documents stored on remote machines.

A document sharing system such as the *world wide web* is composed of three important parts.

1. A standardised addressing scheme that allows unambiguous identification of documents
2. A standard document format : the HyperText Markup Language
3. A standardised protocol that facilitates efficient retrieval of documents stored on a server

---

Note: Open standards and open implementations

[7] **RFC 1939** defines the APOP authentication scheme that is not vulnerable to such attacks.

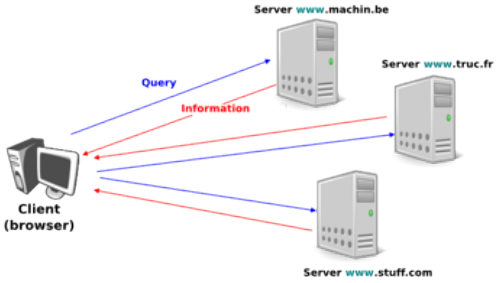


Figure 3.11: World-wide web clients and servers

Open standards have, and are still playing a key role in the success of the *world wide web* as we know it today. Without open standards, the world wide web would never have reached its current size. In addition to open standards, another important factor for the success of the web was the availability of open and efficient implementations of these standards. When CERN started to work on the *web*, their objective was to build a running system that could be used by physicists. They developed open-source implementations of the first web servers and web clients. These open-source implementations were powerful and could be used as is, by institutions willing to share information on the web. They were also extended by other developers who contributed to new features. For example, NCSA added support for images in their Mosaic browser that was eventually used to create Netscape Communications.

The first components of the *world wide web* are the Uniform Resource Identifiers (URI), defined in **RFC 3986**. A URI is a character string that unambiguously identifies a resource on the world wide web. Here is a subset of the BNF for URIs

```
URI          = scheme ":" "//" authority path [ "?" query ] [ "#" fragment ]
scheme       = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
authority    = [ userinfo "@" ] host [ ":" port ]
query        = *( pchar / "/" / "?" )
fragment     = *( pchar / "/" / "?" )
pchar          = unreserved / pct-encoded / sub-delims / ":" / "@"
query          = *( pchar / "/" / "?" )
fragment       = *( pchar / "/" / "?" )
pct-encoded    = "%" HEXDIG HEXDIG
unreserved     = ALPHA / DIGIT / "-" / "." / "_" / "~"
reserved       = gen-delims / sub-delims
gen-delims     = ":" / "/" / "?" / "#" / "[" / "]" / "@"
sub-delims     = "!" / "$" / "&" / "'" / "(" / ")" / "*" / "+" / "," / ";" / "="
```

The first component of a URI is its *scheme*. A *scheme* can be seen as a selector, indicating the meaning of the fields after it. In practice, the scheme often identifies the application-layer protocol that must be used by the client to retrieve the document, but it is not always the case. Some schemes do not imply a protocol at all and some do not indicate a retrievable document [8]. The most frequent scheme is *http that will be described later. A URI scheme can be defined for almost any application layer protocol [#furilist]_. The characters ':* and *//* follow the *scheme* of any URI.

[8] An example of a non-retrievable URI is *urn:isbn:0-380-81593-1* which is an unique identifier for a book, through the urn scheme (see **RFC 3187**). Of course, any URI can be made retrievable via a dedicated server or a new protocol but this one has no explicit protocol. Same thing for the scheme tag (see **RFC 4151**), often used in Web syndication (see **RFC 4287** about the Atom syndication format). Even when the scheme is retrievable (for instance with http'), it is often used only as an identifier, not as a way to get a resource. See http://norman.walsh.name/2006/07/25/namesAndAddresses for a good explanation.

The second part of the URI is the *authority*. With retrievable URI, this includes the DNS name or the IP address of the server where the document can be retrieved using the protocol specified via the *scheme*. This name can be preceded by some information about the user (e.g. a user name) who is requesting the information. Earlier definitions of the URI allowed the specification of a user name and a password before the @ character (**RFC 1738**), but this is now deprecated as placing a password inside a URI is insecure. The host name can be followed by the semicolon character and a port number. A default port number is defined for some protocols and the port number should only be included in the URI if a non-default port number is used (for other protocols, techniques like service DNS records are used).

The third part of the URI is the path to the document. This path is structured as filenames on a Unix host (but it does not imply that the files are indeed stored this way on the server). If the path is not specified, the server will return a default document. The last two optional parts of the URI are used to provide a query and indicate a specific part (e.g. a section in an article) of the requested document. Sample URIs are shown below.

```
http://tools.ietf.org/html/rfc3986.html
mailto:infobot@example.com?subject=current-issue
http://docs.python.org/library/basehttpserver.html?highlight=http#BaseHTTPServer.BaseHTTPRequestHa
telnet://[2001:db8:3080:3::2]:80/
ftp://cnn.example.com&story=breaking_news@10.0.0.1/top_story.htm
```

The first URI corresponds to a document named *rfc3986.html* that is stored on the server named *tools.ietf.org* and can be accessed by using the *http* protocol on its default port. The second URI corresponds to an email message, with subject *current-issue*, that will be sent to user *infobot* in domain *example.com*. The *mailto:* URI scheme is defined in **RFC 6068**. The third URI references the portion *BaseHTTPServer.BaseHTTPRequestHandler* of the document *basehttpserver.html* that is stored in the *library* directory on server *docs.python.org*. This document can be retrieved by using the *http* protocol. The query *highlight=http* is associated to this URI. The fourth example is a server that operates the telnet protocol, uses IPv6 address *2001:db8:3080:3::2* and is reachable on port 80. The last URI is somewhat special. Most users will assume that it corresponds to a document stored on the *cnn.example.com* server. However, to parse this URI, it is important to remember that the @ character is used to separate the user name from the host name in the authorisation part of a URI. This implies that the URI points to a document named *top_story.htm* on host having IPv4 address *10.0.0.1*. The document will be retrieved by using the *ftp* protocol with the user name set to *cnn.example.com&story=breaking_news*.

The second component of the *word wide web* is the HyperText Markup Language (HTML). HTML defines the format of the documents that are exchanged on the *web*. The first version of HTML was derived from the Standard Generalized Markup Language (SGML) that was standardised in 1986 by *ISO*. SGML was designed to allow large project documents in industries such as government, law or aerospace to be shared efficiently in a machine-readable manner. These industries require documents to remain readable and editable for tens of years and insisted on a standardised format supported by multiple vendors. Today, SGML is no longer widely used beyond specific applications, but its descendants including *HTML* and *XML* are now widespread.

A markup language is a structured way of adding annotations about the formatting of the document within the document itself. Example markup languages include troff, which is used to write the Unix man pages or Latex. HTML uses markers to annotate text and a document is composed of *HTML elements*. Each element is usually composed of three items: a start tag that potentially includes some specific attributes, some text (often including other elements), and an end tag. A HTML tag is a keyword enclosed in angle brackets. The generic form of a HTML element is

```
<tag>Some text to be displayed</tag>
```

More complex HTML elements can also include optional attributes in the start tag

```
<tag attribute1="value1" attribute2="value2">some text to be displayed</tag>
```

The HTML document shown below is composed of two parts : a header, delineated by the <*head*> and <*/head*> markers, and a body (between the <*body*> and <*/body*> markers). In the example below, the header only contains a title, but other types of information can be included in the header. The body contains an image, some text and a list with three hyperlinks. The image is included in the web page by indicating its URI between brackets inside the <*img src="..."*> marker. The image can, of course, reside on any server and the client will automatically download it when rendering the web page. The <*h1*>...<*/h1*> marker is used to specify the first level of headings. The <*ul*> marker indicates an unnumbered list while the <*li*> marker indicates a list item. The <*a href="URI"*>*text*<*/a*>

indicates a hyperlink. The *text* will be underlined in the rendered web page and the client will fetch the specified URI when the user clicks on the link.

Figure 3.12: A simple HTML page

Additional details about the various extensions to HTML may be found in the official specifications maintained by W3C.

The third component of the *world wide web* is the HyperText Transport Protocol (HTTP). HTTP is a text-based protocol, in which the client sends a request and the server returns a response. HTTP runs above the bytestream service and HTTP servers listen by default on port *80*. The design of HTTP has largely been inspired by the Internet email protocols. Each HTTP request contains three parts :

- a *method* , that indicates the type of request, a URI, and the version of the HTTP protocol used by the client
- a *header* , that is used by the client to specify optional parameters for the request. An empty line is used to mark the end of the header
- an optional MIME document attached to the request

The response sent by the server also contains three parts :

- a *status line* , that indicates whether the request was successful or not
- a *header* , that contains additional information about the response. The response header ends with an empty line.
- a MIME document

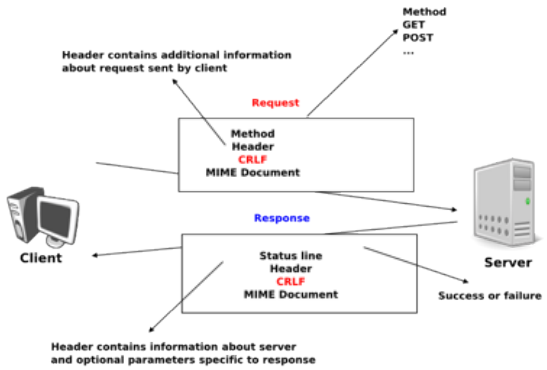


Figure 3.13: HTTP requests and responses

Several types of method can be used in HTTP requests. The three most important ones are :

- the *GET* method is the most popular one. It is used to retrieve a document from a server. The *GET* method is encoded as *GET* followed by the path of the URI of the requested document and the version of HTTP used by the client. For example, to retrieve the http://www.w3.org/MarkUp/ URI, a client must open a TCP on port *80* with host *www.w3.org* and send a HTTP request containing the following line :

```
GET /MarkUp/ HTTP/1.0
```

- the *HEAD* method is a variant of the *GET* method that allows the retrieval of the header lines for a given URI without retrieving the entire document. It can be used by a client to verify if a document exists, for instance.
- the *POST* method can be used by a client to send a document to a server. The sent document is attached to the HTTP request as a MIME document.

HTTP clients and servers can include many different HTTP headers in HTTP requests and responses. Each HTTP header is encoded as a single ASCII-line terminated by *CR* and *LF*. Several of these headers are briefly described below. A detailed discussion of all standard headers may be found in **RFC 1945**. The MIME headers can appear in both HTTP requests and HTTP responses.

- the *Content-Length:* header is the *MIME* header that indicates the length of the MIME document in bytes.
- the *Content-Type:* header is the *MIME* header that indicates the type of the attached MIME document. HTML pages use the *text/html* type.
- the *Content-Encoding:* header indicates how the *MIME document* has been encoded. For example, this header would be set to *x-gzip* for a document compressed using the gzip software.

RFC 1945 and **RFC 2616** define headers that are specific to HTTP responses. These server headers include :

- the *Server:* header indicates the version of the web server that has generated the HTTP response. Some servers provide information about their software release and optional modules that they use. For security reasons, some system administrators disable these headers to avoid revealing too much information about their server to potential attackers.
- the *Date:* header indicates when the HTTP response has been produced by the server.
- the *Last-Modified:* header indicates the date and time of the last modification of the document attached to the HTTP response.

Similarly, the following header lines can only appear inside HTTP requests sent by a client :

- the *User-Agent:* header provides information about the client that has generated the HTTP request. Some servers analyse this header line and return different headers and sometimes different documents for different user agents.
- the *If-Modified-Since:* header is followed by a date. It enables clients to cache in memory or on disk the recent or most frequently used documents. When a client needs to request a URI from a server, it first checks whether the document is already in its cache. If it is, the client sends a HTTP request with the *If-Modified-Since:* header indicating the date of the cached document. The server will only return the document attached to the HTTP response if it is newer than the version stored in the client's cache.
- the *Referrer:* header is followed by a URI. It indicates the URI of the document that the client visited before sending this HTTP request. Thanks to this header, the server can know the URI of the document containing the hyperlink followed by the client, if any. This information is very useful to measure the impact of advertisements containing hyperlinks placed on websites.
- the *Host:* header contains the fully qualified domain name of the URI being requested.

Note: The importance of the *Host:* header line

The first version of HTTP did not include the *Host:* header line. This was a severe limitation for web hosting companies. For example consider a web hosting company that wants to serve both *web.example.com* and *www.example.net* on the same physical server. Both web sites contain a */index.html* document. When a client sends a request for either *http://web.example.com/index.html* or *http://www.example.net/index.html*, the HTTP 1.0 request contains the following line :

```
GET /index.html HTTP/1.0
```

By parsing this line, a server cannot determine which *index.html* file is requested. Thanks to the *Host:* header line, the server knows whether the request is for *http://web.example.com/index.html* or *http://www.dummy.net/index.html*. Without the *Host:* header, this is impossible. The *Host:* header line allowed web hosting companies to develop their business by supporting a large number of independent web servers on the same physical server.

The status line of the HTTP response begins with the version of HTTP used by the server (usually *HTTP/1.0* defined in **RFC 1945** or *HTTP/1.1* defined in **RFC 2616**) followed by a three digit status code and additional information in English. HTTP status codes have a similar structure as the reply codes used by SMTP.

- All status codes starting with digit *2* indicate a valid response. *200 Ok* indicates that the HTTP request was successfully processed by the server and that the response is valid.
- All status codes starting with digit *3* indicate that the requested document is no longer available on the server. *301 Moved Permanently* indicates that the requested document is no longer available on this server. A *Location:* header containing the new URI of the requested document is inserted in the HTTP response. *304 Not Modified* is used in response to an HTTP request containing the *If-Modified-Since:* header. This status line is used by the server if the document stored on the server is not more recent than the date indicated in the *If-Modified-Since:* header.
- All status codes starting with digit *4* indicate that the server has detected an error in the HTTP request sent by the client. *400 Bad Request* indicates a syntax error in the HTTP request. *404 Not Found* indicates that the requested document does not exist on the server.
- All status codes starting with digit *5* indicate an error on the server. *500 Internal Server Error* indicates that the server could not process the request due to an error on the server itself.

In both the HTTP request and the HTTP response, the MIME document refers to a representation of the document with the MIME headers indicating the type of document and its size.

As an illustration of HTTP/1.0, the transcript below shows a HTTP request for http://www.ietf.org and the corresponding HTTP response. The HTTP request was sent using the curl command line tool. The *User-Agent:* header line contains more information about this client software. There is no MIME document attached to this HTTP request, and it ends with a blank line.

```
GET / HTTP/1.0
User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
Host: www.ietf.org
```

The HTTP response indicates the version of the server software used with the modules included. The *Last-Modified:* header indicates that the requested document was modified about one week before the request. A HTML document (not shown) is attached to the response. Note the blank line between the header of the HTTP response and the attached MIME document. The *Server:* header line has been truncated in this output.

```
HTTP/1.1 200 OK
Date: Mon, 15 Mar 2010 13:40:38 GMT
Server: Apache/2.2.4 (Linux/SUSE) mod_ssl/2.2.4 OpenSSL/0.9.8e (truncated)
Last-Modified: Tue, 09 Mar 2010 21:26:53 GMT
Content-Length: 17019
Content-Type: text/html

<!DOCTYPE HTML PUBLIC .../HTML>
```

HTTP was initially designed to share self-contained text documents. For this reason, and to ease the implementation of clients and servers, the designers of HTTP chose to open a TCP connection for each HTTP request. This implies that a client must open one TCP connection for each URI that it wants to retrieve from a server as illustrated on the figure below. For a web page containing only text documents this was a reasonable design choice as the client usually remains idle while the (human) user is reading the retrieved document.

However, as the web evolved to support richer documents containing images, opening a TCP connection for each URI became a performance problem [Mogul1995]. Indeed, besides its HTML part, a web page may include

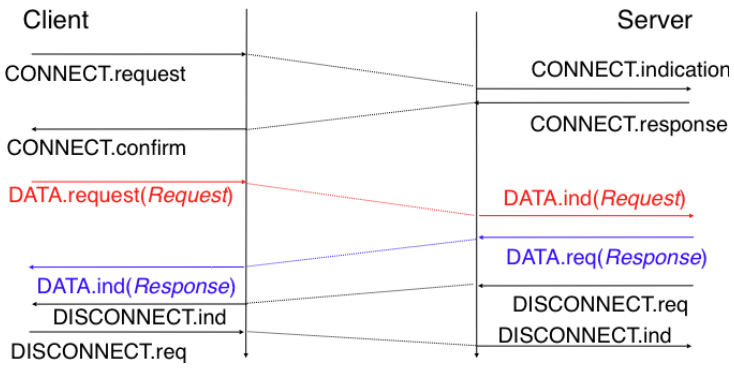


Figure 3.14: HTTP 1.0 and the underlying TCP connection

dozens of images or more. Forcing the client to open a TCP connection for each component of a web page has two important drawbacks. First, the client and the server must exchange packets to open and close a TCP connection as we will see later. This increases the network overhead and the total delay of completely retrieving all the components of a web page. Second, a large number of established TCP connections may be a performance bottleneck on servers.

This problem was solved by extending HTTP to support persistent TCP connections **RFC 2616**. A persistent connection is a TCP connection over which a client may send several HTTP requests. This is illustrated in the figure below.

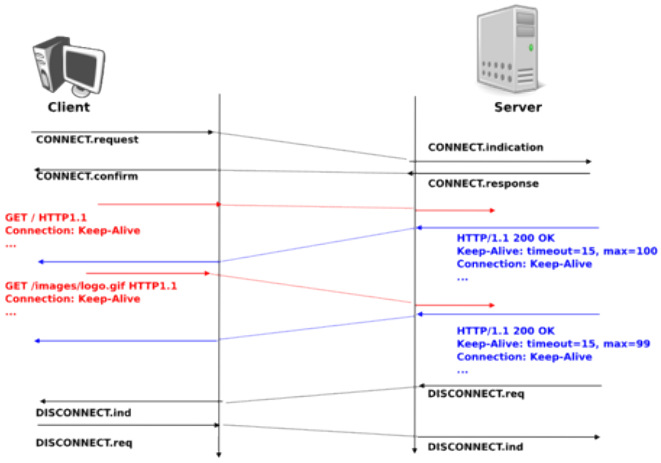


Figure 3.15: HTTP 1.1 persistent connections

To allow the clients and servers to control the utilisation of these persistent TCP connections, HTTP 1.1 **RFC 2616** defines several new HTTP headers :

- The *Connection:* header is used with the *Keep-Alive* argument by the client to indicate that it expects the underlying TCP connection to be persistent. When this header is used with the *Close* argument, it indicates that the entity that sent it will close the underlying TCP connection at the end of the HTTP response.
- The *Keep-Alive:* header is used by the server to inform the client about how it agrees to use the persistent connection. A typical *Keep-Alive:* contains two parameters : the maximum number of requests that the server agrees to serve on the underlying TCP connection and the timeout (in seconds) after which the server will close an idle connection

The example below shows the operation of HTTP/1.1 over a persistent TCP connection to retrieve three URIs stored on the same server. Once the connection has been established, the client sends its first request with the *Connection: keep-alive* header to request a persistent connection.

```
GET / HTTP/1.1
Host: www.kame.net
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive
```

The server replies with the *Connection: Keep-Alive* header and indicates that it accepts a maximum of 100 HTTP requests over this connection and that it will close the connection if it remains idle for 15 seconds.

```
HTTP/1.1 200 OK
Date: Fri, 19 Mar 2010 09:23:37 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Length: 3462
Content-Type: text/html

<html>...   </html>
```

The client sends a second request for the style sheet of the retrieved web page.

```
GET /style.css HTTP/1.1
Host: www.kame.net
Referer: http://www.kame.net/
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive
```

The server replies with the requested style sheet and maintains the persistent connection. Note that the server only accepts 99 remaining HTTP requests over this persistent connection.

```
HTTP/1.1 200 OK
Date: Fri, 19 Mar 2010 09:23:37 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Last-Modified: Mon, 10 Apr 2006 05:06:39 GMT
Content-Length: 2235
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/css

...
```

Then the client automatically requests the web server's icon [9] , that could be displayed by the browser. This server does not contain such URI and thus replies with a *404* HTTP status. However, the underlying TCP connection is not closed immediately.

```
GET /favicon.ico HTTP/1.1
Host: www.kame.net
Referer: http://www.kame.net/
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive

HTTP/1.1 404 Not Found
Date: Fri, 19 Mar 2010 09:23:40 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Content-Length: 318
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"> ...
```

[9] Favorite icons are small icons that are used to represent web servers in the toolbar of Internet browsers. Microsoft added this feature in their browsers without taking into account the W3C standards. See http://www.w3.org/2005/10/howto-favicon for a discussion on how to cleanly support such favorite icons.

As illustrated above, a client can send several HTTP requests over the same persistent TCP connection. However, it is important to note that all of these HTTP requests are considered to be independent by the server. Each HTTP request must be self-contained. This implies that each request must include all the header lines that are required by the server to understand the request. The independence of these requests is one of the important design choices of HTTP. As a consequence of this design choice, when a server processes a HTTP request, it doesn't' use any other information than what is contained in the request itself. This explains why the client adds its *User-Agent:* header in all of the HTTP requests it sends over the persistent TCP connection.

However, in practice, some servers want to provide content tuned for each user. For example, some servers can provide information in several languages or other servers want to provide advertisements that are targeted to different types of users. To do this, servers need to maintain some information about the preferences of each user and use this information to produce content matching the user's preferences. HTTP contains several mechanisms that enable to solve this problem. We discuss three of them below.

A first solution is to force the users to be authenticated. This was the solution used by *FTP* to control the files that each user could access. Initially, user names and passwords could be included inside URIs **RFC 1738**. However, placing passwords in the clear in a potentially publicly visible URI is completely insecure and this usage has now been deprecated **RFC 3986**. HTTP supports several extension headers **RFC 2617** that can be used by a server to request the authentication of the client by providing his/her credentials. However, user names and passwords have not been popular on web servers as they force human users to remember one user name and one password per server. Remembering a password is acceptable when a user needs to access protected content, but users will not accept the need for a user name and password only to receive targeted advertisements from the web sites that they visit.

A second solution to allow servers to tune that content to the needs and capabilities of the user is to rely on the different types of *Accept-** HTTP headers. For example, the *Accept-Language:* can be used by the client to indicate its preferred languages. Unfortunately, in practice this header is usually set based on the default language of the browser and it is not possible for a user to indicate the language it prefers to use by selecting options on each visited web server.

The third, and widely adopted, solution are HTTP cookies. HTTP cookies were initially developed as a private extension by Netscape. They are now part of the standard **RFC 6265**. In a nutshell, a cookie is a short string that is chosen by a server to represent a given client. Two HTTP headers are used : *Cookie:* and *Set-Cookie:*. When a server receives an HTTP request from a new client (i.e. an HTTP request that does not contain the *Cookie:* header), it generates a cookie for the client and includes it in the *Set-Cookie:* header of the returned HTTP response. The *Set-Cookie:* header contains several additional parameters including the domain names for which the cookie is valid. The client stores all received cookies on disk and every time it sends a HTTP request, it verifies whether it already knows a cookie for this domain. If so, it attaches the *Cookie:* header to the HTTP request. This is illustrated in the figure below with HTTP 1.1, but cookies also work with HTTP 1.0.

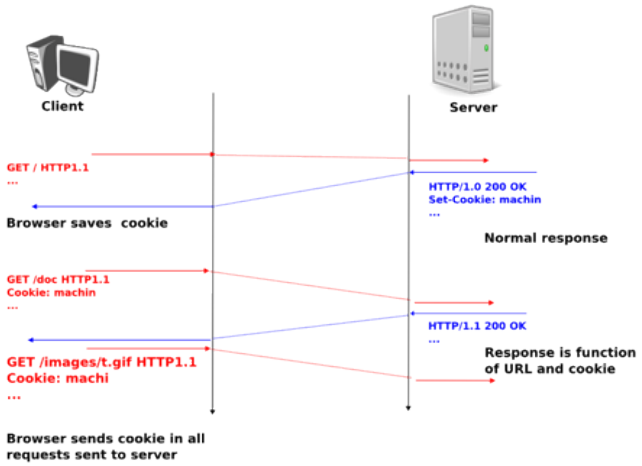


Figure 3.16: HTTP cookies

Note: Privacy issues with HTTP cookies

The HTTP cookies introduced by Netscape are key for large e-commerce websites. However, they have also raised many discussions concerning their potential misuses. Consider *ad.com*, a company that delivers lots of advertisements on web sites. A web site that wishes to include *ad.com*'s advertisements next to its content will add links to *ad.com* inside its HTML pages. If *ad.com* is used by many web sites, *ad.com* could be able to track the interests of all the users that visit its client websites and use this information to provide targeted advertisements. Privacy advocates have even sued online advertisement companies to force them to comply with the privacy regulations. More recent related technologies also raise privacy concerns

3.5 Remote Procedure Calls

In the previous sections, we have described several protocols that enable humans to exchange messages and access to remote documents. This is not the only usage of computer networks and in many situations applications use the network to exchange information with other applications. When an application needs to perform a large computation on a host, it can sometimes be useful to request computations from other hosts. Many distributed systems have been built by distributing applications on different hosts and using *Remote Procedure Calls* as a basic building block.

In traditional programming languages, *procedure calls* allow programmers to better structure their code. Each procedure is identified by a name, a return type and a set of parameters. When a procedure is called, the current flow of program execution is diverted to execute the procedure. This procedure uses the provided parameters to perform its computation and returns one or more values. This programming model was designed with a single host in mind. In a nutshell, most programming languages support it as follows :

1. The caller places the values of the parameters at a location (register, stack, ...) where the callee can access them
2. The caller transfers the control of execution to the callee's procedure
3. The callee accesses the parameters and performs the requested computation
4. The callee places the return value(s) at a location (register, stack, ...) where the caller can access them
5. The callee returns the control of execution to the caller's

This model was developed with a single host in mind. How should it be modified if the caller and the callee are different hosts connected through a network ? Since the two hosts can be different, the two main problems are the fact they do not share the same memory and that they do not necessarily use the same representation for numbers, characters, ... Let us examine how the five steps identified above can be supported through a network.

The first problem to be solved is how to transfer the information from the caller to the callee. This problem is not simple and includes two sub-problems. The first subproblem is the encoding of the information. How to encode the values of the parameters so that they can be transferred correctly through the network ? The second problem is how to reach the callee through the network ? The callee is identified by a procedure name, but to use the transport service, we need to convert this name into an address and a port number.

3.5.1 Encoding data

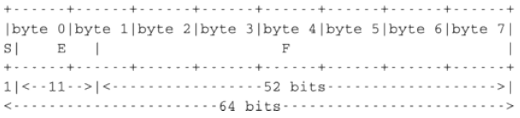
The encoding problem exists in a wide range of applications. In the previous sections, we have described how character-based encodings are used by email and http. Although standard encoding techniques such as ASN.1 [Dubuisson2000] have been defined to cover most application needs, many applications have defined their specific encoding. *Remote Procedure Call* are no exception to this rule. The three most popular encoding methods are probably XDR **RFC 1832** used by ONC-RPC **RFC 1831**, XML, used by XML-RPC and JSON **RFC 4627**.

The eXternal Data Representation (XDR) Standard, defined in **RFC 1832** is an early specification that describes how information exchanged during Remote Procedure Calls should be encoded before being transmitted through a network. Since the transport service allows to transfer a block of bytes (with the connectionless service) or a stream of bytes (by using the connection-oriented service), XDR maps each datatype onto a sequence of bytes.

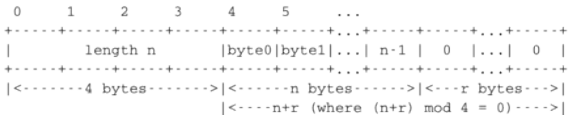
The caller encodes each data in the appropriate sequence and the callee decodes the received information. Here are a few examples extracted from **RFC 1832** to illustrate how this encoding/decoding can be performed.

For basic data types, **RFC 1832** simply maps their representation into a sequence of bytes. For example a 32 bits integer is transmitted as follows (with the most significant byte first, which corresponds to big-endian encoding).

XDR also supports 64 bits integers and booleans. The booleans are mapped onto integers (*0* for *false* and *1* for *true*). For the floating point numbers, the encoding defined in the IEEE standard is used.



In this representation, the first bit (*S*) is the sign (*0* represents positive). The next 11 bits represent the exponent of the number (*E*), in base 2, and the remaining 52 bits are the fractional part of the number (*F*). The floating point number that corresponds to this representation is $(-1)^S \times 2^{E-1023} \times 1.F$. XDR also allows to encode complex data types. A first example is the string of bytes. A string of bytes is composed of two parts : a length (encoded as an integer) and a sequence of bytes. For performance reasons, the encoding of a string is aligned to 32 bits boundaries. This implies that some padding bytes may be inserted during the encoding operation is the length of the string is not a multiple of 4. The structure of the string is shown below (source **RFC 1832**).

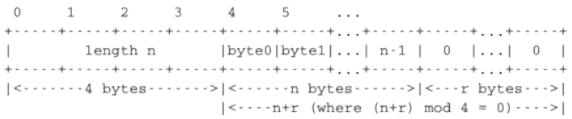


In some situations, it is necessary to encode fixed or variable length arrays. XDR **RFC 1832** supports such arrays. For example, the encoding below corresponds to a variable length array containing n elements. The encoded representation starts with an integer that contains the number of elements and follows with all elements in sequence. It is also possible to encode a fixed-length array. In this case, the first integer is missing.

XDR also supports the definition of unions, structures, ... Additional details are provided in **RFC 1832**.

A second popular method to encode data is the JavaScript Object Notation (JSON). This syntax was initially defined to allow applications written in JavaScript to exchange data, but it has now wider usages. JSON **RFC 4627** is a text-based representation. The simplest data type is the integer. It is represented as a sequence of digits in ASCII. Strings can also be encoding by using JSON. A JSON string always starts and ends with a quote character (“) as in the C language. As in the C language, some characters (like “ or \) must be escaped if they appear in a string. **RFC 4627** describes this in details. Booleans are also supported by using the strings *false* and *true*. Like XDR, JSON supports more complex data types. A structure or object is defined as a comma separated list of elements enclosed in curly brackets. **RFC 4627** provides the following example as an illustration.

```
{
   "Image": {
       "Width":  800,
       "Height": 600,
       "Title":  "View from 15th Floor",
       "Thumbnail": {
           "Url":    "http://www.example.com/image/481989943",
           "Height": 125,
```




```
            "Width":  100
        },
        "ID": 1234
      }
}
```

This object has one field named *Image*. It has five attributes. The first one, *Width*, is an integer set to 800. The third one is a string. The fourth attribute, *Thumbnail* is also an object composed of three different attributes, one string and two integers. JSON can also be used to encode arrays or lists. In this case, square brackets are used as delimiters. The snippet below shows an array which contains the prime integers that are smaller than ten.

```
{
  "Primes" : [ 2, 3, 5, 7 ]
}
```

Compared with XDR, the main advantage of JSON is that the transfer syntax is easily readable by a human. However, this comes at the expense of a less compact encoding. Some data encoded in JSON will usually take more space than when it is encoded with XDR. More compact encoding schemes have been defined, see e.g. [BH2013] and the references therein.

3.5.2 Reaching the callee

The second subproblem is how to reach the callee. A simple solution to this problem is to make sure that the callee listens on a specific port on the remote machine and then exchange information with this server process. This is the solution chosen for JSON-RPC [JSON-RPC2]. JSON-RPC can be used over the connectionless or the connection-oriented transport. A JSON-RPC request contains the following information :

- *jsonrpc*: a string indicating the version of the protocol used. This is important to allow the protocol to evolve in the future.
- *method*: a string that contains the name of the procedure which is invoked
- *params*: a structure that contains the values of the parameters that are passed to the method
- *id*: an identifier chosen by the caller

The JSON-RPC is encoded as a JSON object. For example, the example below shows an invokation of a method called *sum* with *1* and *3* as parameters.

```
{"jsonrpc": "2.0", "method": "sum", "params": [1, 3], "id": 1}
```

Upon reception of this JSON structure, the callee parses the object, locates the corresponding method and passes the parameters. This method returns a response which is also encoded as a JSON structure. This response contains the following information :

- *jsonrpc*: a string indicating the version of the protocol used to encode the response
- *id*: the same identifier as the identifier chosen by the caller
- *result*: if the request succeeded, this member contains the result of the request (in our example, value *4*).
- *error*: if the method called does not exist or its execution causes an error, the *result* element will be replaced by an *error* element which contains the following members :
  - *code*: a number that indicates the type of error. Several error codes are defined in [JSON-RPC2]. For example, *-32700* indicates an error in parsing the request, *-32602* indicates invalid parameters and *-32601* indicates that the method could not be found on the server. Other error codes are listed in [JSON-RPC2].

- *message*: a string (limited to one sentence) that provides a short description of the error.
- *data*: an optional field that provides additional information about the error.

Coming back to our example with the call for the *sum* procedure, it would return the following JSON structure.

```
{ "jsonrpc": "2.0", "result": 4, "id": 1}
```

If the *sum* method is not implemented on the server, it would reply with the following response.

```
{ "jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "1"}
```

The *id* field, which is present in the request and the response plays the same role as the identifier field in the DNS message. It allows the caller to match the response with the request that it sent. This *id* is very important when JSON-RPC is used over the connectionless service which is unreliable. If a request is sent, it may need to be retransmitted and it is possible that a callee will receive twice the same request (e.g. if the response for the first request was lost). In the DNS, when a request is lost, it can be retransmitted without causing any difficulty. However with remote procedure calls in general, losses can cause some problems. Consider a method which is used to deposit money on a bank account. If the request is lost, it will be retransmitted and the deposit will be eventually performed. However, if the response is lost, the caller will also retransmit its request. This request will be received by the callee that will deposit the money again. To prevent this problem from affecting the application, either the programmer must ensure that the remote procedures that it calls can be safely called multiple times or the application must verify whether the request has been transmitted earlier. In most deployments, the programmers use remote methods that can be safely called multiple times without breaking the application logic.

ONC-RPC uses a more complex method to allow a caller to reach the callee. On a host, server processes can run on different ports and given the limited number of port values (2^{16} per host on the Internet), it is impossible to reserve one port number for each method. The solution used in ONC-RPC **RFC 1831** is to use a special method which is called the *portmapper* **RFC 1833**. The *portmapper* is a kind of directory that runs on a server that hosts methods. The *portmapper* runs on a standard port (*111* for ONC-RPC **RFC 1833**). A server process that implements a method registers its method on the local *portmapper*. When a caller needs to call a method on a remote server, it first contacts the *portmapper* to obtain the port number of the server which implements the method. The response from the portmapper allows it to directly contact the server process which implements the method.

3.6 Internet transport protocols

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=6

Transport protocols rely on the service provided by the network layer. On the Internet, the network layer provides a connectionless service. The network layer identifies each (interface of a) host by using an IP address. It enables hosts to transmit packets that contain up to 64 KBytes of payload to any destination reachable through the network. The network layer does not guarantee the delivery of information, cannot detect transmission errors and does not preserve sequence integrity.

Several transport protocols have been designed to provide a richer service to the applications. The two most widely deployed transport protocols on the Internet are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). A third important transport protocol, the Stream Control Transmission Protocol (SCTP) **RFC 4960** appeared in the early 2000s. It is currently used by some particular applications such as signaling in Voice over IP networks. We also describe SCTP in this section to present a different design than TCP. The Real Time Transport Protocol (RTP), defined in **RFC 3550** is another important protocol that is used by many multimedia applications. It includes functions that belong to the transport layer, but also functions that are related to the encoding of the information. Due to space limitations, we do not discuss it in details in this section.

3.7 The User Datagram Protocol

The User Datagram Protocol (UDP) is defined in **RFC 768**. It provides an unreliable connectionless transport service on top of the unreliable network layer connectionless service. The main characteristics of the UDP service are :

- the UDP service cannot deliver SDUs that are larger than 65467 bytes [10]
- the UDP service does not guarantee the delivery of SDUs (losses and desquencing can occur)
- the UDP service will not deliver a corrupted SDU to the destination

Compared to the connectionless network layer service, the main advantage of the UDP service is that it allows several applications running on a host to exchange SDUs with several other applications running on remote hosts. Let us consider two hosts, e.g. a client and a server. The network layer service allows the client to send information to the server, but if an application running on the client wants to contact a particular application running on the server, then an additional addressing mechanism is required other than the IP address that identifies a host, in order to differentiate the application running on a host. This additional addressing is provided by *port numbers*. When a server using UDP is enabled on a host, this server registers a *port number*. This *port number* will be used by the clients to contact the server process via UDP.

The figure below shows a typical usage of the UDP port numbers. The client process uses port number *1234* while the server process uses port number *5678*. When the client sends a request, it is identified as originating from port number *1234* on the client host and destined to port number *5678* on the server host. When the server process replies to this request, the server's UDP implementation will send the reply as originating from port *5678* on the server host and destined to port *1234* on the client host.

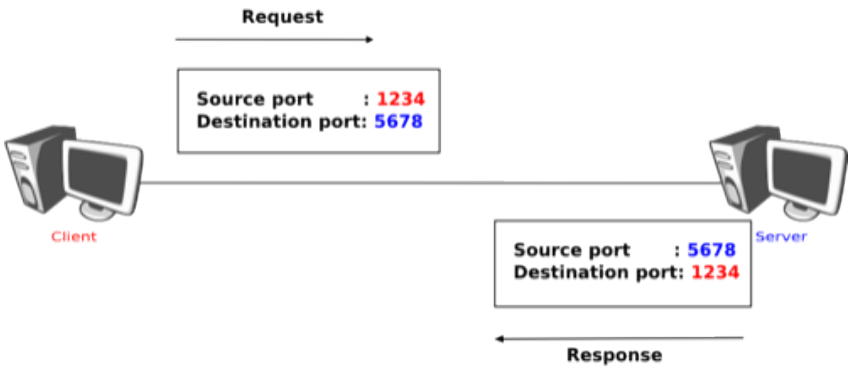


Figure 3.17: Usage of the UDP port numbers

UDP uses a single segment format shown in the figure below.

Figure 3.18: UDP Header Format

The UDP header contains four fields :

- a 16 bits source port
- a 16 bits destination port

[10] This limitation is due to the fact that the network layer cannot transport packets that are larger than 64 KBytes. As UDP does not include any segmentation/reassembly mechanism, it cannot split a SDU before sending it. The UDP header consumes 8 bytes and the IPv6 header 60. With IPv4, the IPv4 header only consumes 20 bytes and thus the maximum UDP payload size is 65507 bytes.

- a 16 bits length field
- a 16 bits checksum

As the port numbers are encoded as a 16 bits field, there can be up to only 65535 different server processes that are bound to a different UDP port at the same time on a given server. In practice, this limit is never reached. However, it is worth noticing that most implementations divide the range of allowed UDP port numbers into three different ranges :

- the privileged port numbers (1 < port < 1024)
- the ephemeral port numbers (officially [11] 49152 <= port <= 65535)
- the registered port numbers (officially 1024 <= port < 49152)

In most Unix variants, only processes having system administrator privileges can be bound to port numbers smaller than *1024*. Well-known servers such as *DNS*, *NTP* or *RPC* use privileged port numbers. When a client needs to use UDP, it usually does not require a specific port number. In this case, the UDP implementation will allocate the first available port number in the ephemeral range. The range of registered port numbers should be used by servers. In theory, developers of network servers should register their port number officially through IANA, but few developers do this.

Note: Computation of the UDP checksum

The checksum of the UDP segment is computed over :

- a pseudo header **RFC 2460** containing the source address, the destination address, the packet length encoded as a 32 bits number and a 32 bits bit field containing the three most significant bytes set to 0 and the low order byte set to 17
- the entire UDP segment, including its header

This pseudo-header allows the receiver to detect errors affecting the source or destination addresses placed in the IP layer below. This is a violation of the layering principle that dates from the time when UDP and IP were elements of a single protocol. It should be noted that if the checksum algorithm computes value '0x0000', then value '0xffff' is transmitted. A UDP segment whose checksum is set to '0x0000' is a segment for which the transmitter did not compute a checksum upon transmission. Some *NFS* servers chose to disable UDP checksums for performance reasons when running over IPv4, but this caused problems that were difficult to diagnose. Over IPv6, the UDP checksum cannot be disabled. A detailed discussion of the implementation of the Internet checksum may be found in **RFC 1071**

Several types of applications rely on UDP. As a rule of thumb, UDP is used for applications where delay must be minimised or losses can be recovered by the application itself. A first class of the UDP-based applications are applications where the client sends a short request and expects a quick and short answer. The *DNS* is an example of a UDP application that is often used in the wide area. However, in local area networks, many distributed systems rely on Remote Procedure Call (*RPC*) that is often used on top of UDP. In Unix environments, the Network File System (*NFS*) is built on top of RPC and runs frequently on top of UDP. A second class of UDP-based applications are the interactive computer games that need to frequently exchange small messages, such as the player's location or their recent actions. Many of these games use UDP to minimise the delay and can recover from losses. A third class of applications are multimedia applications such as interactive Voice over IP or interactive Video over IP. These interactive applications expect a delay shorter than about 200 milliseconds between the sender and the receiver and can recover from losses directly inside the application.

3.8 The Transmission Control Protocol

The Transmission Control Protocol (TCP) was initially defined in **RFC 793**. Several parts of the protocol have been improved since the publication of the original protocol specification [12]. However, the basics of the protocol remain and an implementation that only supports **RFC 793** should inter-operate with today's implementation.

[11] A discussion of the ephemeral port ranges used by different TCP/UDP implementations may be found in http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html

[12] A detailed presentation of all standardisation documents concerning TCP may be found in **RFC 4614**

TCP provides a reliable bytestream, connection-oriented transport service on top of the unreliable connectionless network service provided by *IP*. TCP is used by a large number of applications, including :

- Email (*SMTP*, *POP*, *IMAP*)
- World wide web (*HTTP*, ...)
- Most file transfer protocols (*ftp*, peer-to-peer file sharing applications , ...)
- remote computer access : *telnet*, *ssh*, *X11*, *VNC*, ...
- non-interactive multimedia applications : flash

On the global Internet, most of the applications used in the wide area rely on TCP. Many studies [13] have reported that TCP was responsible for more than 90% of the data exchanged in the global Internet.

To provide this service, TCP relies on a simple segment format that is shown in the figure below. Each TCP segment contains a header described below and, optionally, a payload. The default length of the TCP header is twenty bytes, but some TCP headers contain options.

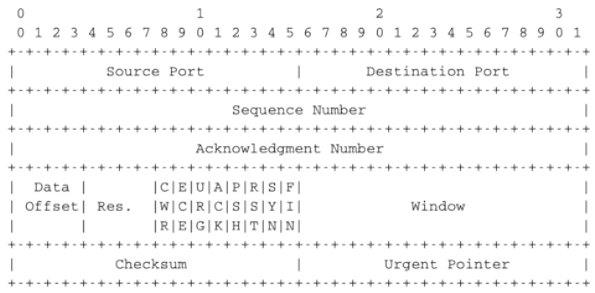


Figure 3.19: TCP header format

A TCP header contains the following fields :

- Source and destination ports. The source and destination ports play an important role in TCP, as they allow the identification of the connection to which a TCP segment belongs. When a client opens a TCP connection, it typically selects an ephemeral TCP port number as its source port and contacts the server by using the server's port number. All the segments that are sent by the client on this connection have the same source and destination ports. The server sends segments that contain as source (resp. destination port, the destination (resp. source) port of the segments sent by the client (see figure *Utilization of the TCP source and destination ports*). A TCP connection is always identified by five pieces of information :
  - the address of the client
  - the address of the server
  - the port chosen by the client
  - the port chosen by the server
  - TCP
- the *sequence number* (32 bits), *acknowledgement number* (32 bits) and *window* (16 bits) fields are used to provide a reliable data transfer, using a window-based protocol. In a TCP bytestream, each byte of the stream consumes one sequence number. Their utilisation will be described in more detail in section *TCP reliable data transfer*
- the *Urgent pointer* is used to indicate that some data should be considered as urgent in a TCP bytestream. However, it is rarely used in practice and will not be described here. Additional details about the utilisation of this pointer may be found in **RFC 793**, **RFC 1122** or [Stevens1994]

[13] Several researchers have analysed the utilisation of TCP and UDP in the global Internet. Most of these studies have been performed by collecting all the packets transmitted over a given link during a period of a few hours or days and then analysing their headers to infer the transport protocol used, the type of application, ... Recent studies include http://www.caida.org/research/traffic-analysis/tcpudpratio/, https://research.sprintlabs.com/packstat/packetoverview.php or http://www.nanog.org/meetings/nanog43/presentations/Labovitz_internetstats_N43.pdf

- the flags field contains a set of bit flags that indicate how a segment should be interpreted by the TCP entity receiving it :
  - the *SYN* flag is used during connection establishment
  - the *FIN* flag is used during connection release
  - the *RST* is used in case of problems or when an invalid segment has been received
  - when the *ACK* flag is set, it indicates that the *acknowledgment* field contains a valid number. Otherwise, the content of the *acknowledgment* field must be ignored by the receiver
  - the *URG* flag is used together with the *Urgent pointer*
  - the *PSH* flag is used as a notification from the sender to indicate to the receiver that it should pass all the data it has received to the receiving process. However, in practice TCP implementations do not allow TCP users to indicate when the *PSH* flag should be set and thus there are few real utilizations of this flag.
- the *checksum* field contains the value of the Internet checksum computed over the entire TCP segment and a pseudo-header as with UDP
- the *Reserved* field was initially reserved for future utilization. It is now used by **RFC 3168**.
- the *TCP Header Length* (THL) or *Data Offset* field is a four bits field that indicates the size of the TCP header in 32 bit words. The maximum size of the TCP header is thus 64 bytes.
- the *Optional header extension* is used to add optional information to the TCP header. Thanks to this header extension, it is possible to add new fields to the TCP header that were not planned in the original specification. This allowed TCP to evolve since the early eighties. The details of the TCP header extension are explained in sections *TCP connection establishment* and *TCP reliable data transfer*.

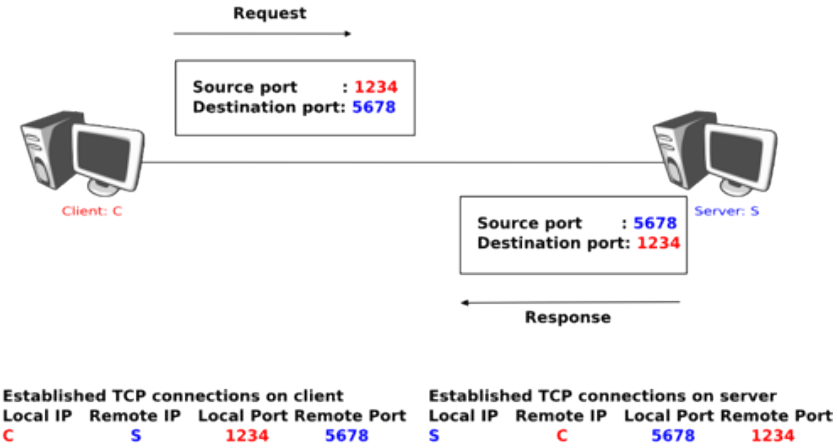


Figure 3.20: Utilization of the TCP source and destination ports

The rest of this section is organised as follows. We first explain the establishment and the release of a TCP connection, then we discuss the mechanisms that are used by TCP to provide a reliable bytestream service. We end the section with a discussion of network congestion and explain the mechanisms that TCP uses to avoid congestion collapse.

3.8.1 TCP connection establishment

A TCP connection is established by using a three-way handshake. The connection establishment phase uses the *sequence number*, the *acknowledgment number* and the *SYN* flag. When a TCP connection is established, the two communicating hosts negotiate the initial sequence number to be used in both directions of the connection. For this, each TCP entity maintains a 32 bits counter, which is supposed to be incremented by one at least every 4

microseconds and after each connection establishment [14]. When a client host wants to open a TCP connection with a server host, it creates a TCP segment with :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the client host's TCP entity

Upon reception of this segment (which is often called a *SYN segment*), the server host replies with a segment containing :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the server host's TCP entity
- the *ACK* flag set
- the *acknowledgment number* set to the *sequence number* of the received *SYN* segment incremented by 1 ($mod\ 2^{32}$). When a TCP entity sends a segment having *x+1* as acknowledgment number, this indicates that it has received all data up to and including sequence number *x* and that it is expecting data having sequence number *x+1*. As the *SYN* flag was set in a segment having sequence number *x*, this implies that setting the *SYN* flag in a segment consumes one sequence number.

This segment is often called a *SYN+ACK* segment. The acknowledgment confirms to the client that the server has correctly received the *SYN* segment. The *sequence number* of the *SYN+ACK* segment is used by the server host to verify that the *client* has received the segment. Upon reception of the *SYN+ACK* segment, the client host replies with a segment containing :

- the *ACK* flag set
- the *acknowledgment number* set to the *sequence number* of the received *SYN+ACK* segment incremented by 1 ($mod\ 2^{32}$)

At this point, the TCP connection is open and both the client and the server are allowed to send TCP segments containing data. This is illustrated in the figure below.

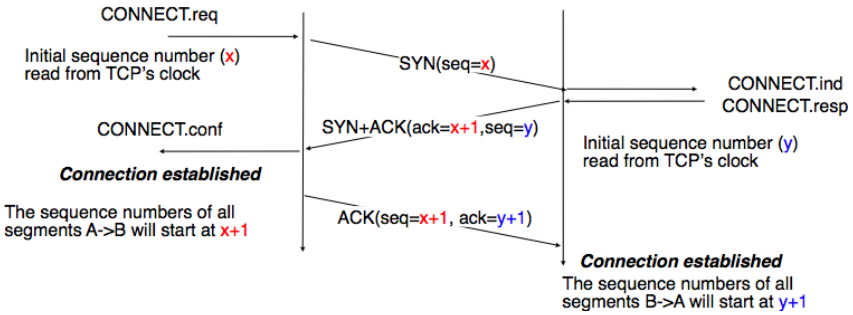


Figure 3.21: Establishment of a TCP connection

In the figure above, the connection is considered to be established by the client once it has received the *SYN+ACK* segment, while the server considers the connection to be established upon reception of the *ACK* segment. The first data segment sent by the client (server) has its *sequence number* set to *x+1* (resp. *y+1*).

Note: Computing TCP's initial sequence number

In the original TCP specification **RFC 793**, each TCP entity maintained a clock to compute the initial sequence number (*ISN*) placed in the *SYN* and *SYN+ACK* segments. This made the ISN predictable and caused a security issue. The typical security problem was the following. Consider a server that trusts a host based on its IP address and allows the system administrator to login from this host without giving a password [15]. Consider now an attacker who knows this particular configuration and is able to send IP packets having the client's address as source. He can send fake TCP segments to the server, but does not receive the server's answers. If he can predict the *ISN* that

[14] This 32 bits counter was specified in **RFC 793**. A 32 bits counter that is incremented every 4 microseconds wraps in about 4.5 hours. This period is much larger than the Maximum Segment Lifetime that is fixed at 2 minutes in the Internet (**RFC 791**, **RFC 1122**).

[15] On many departmental networks containing Unix workstations, it was common to allow users on one of the hosts to use rlogin **RFC 1258** to run commands on any of the workstations of the network without giving any password. In this case, the remote workstation "authenticated" the client host based on its IP address. This was a bad practice from a security viewpoint.

is chosen by the server, he can send a fake *SYN* segment and shortly after the fake *ACK* segment confirming the reception of the *SYN+ACK* segment sent by the server. Once the TCP connection is open, he can use it to send any command to the server. To counter this attack, current TCP implementations add randomness to the *ISN*. One of the solutions, proposed in **RFC 1948** is to compute the *ISN* as

```
ISN = M + H(localhost, localport, remotehost, remoteport, secret).
```

where *M* is the current value of the TCP clock and *H'is a cryptographic hash function. 'localhost* and *remotehost* (resp. *localport* and *remoteport*) are the IP addresses (port numbers) of the local and remote host and *secret* is a random number only known by the server. This method allows the server to use different ISNs for different clients at the same time. Measurements performed with the first implementations of this technique showed that it was difficult to implement it correctly, but today's TCP implementation now generate good ISNs.

A server could, of course, refuse to open a TCP connection upon reception of a *SYN* segment. This refusal may be due to various reasons. There may be no server process that is listening on the destination port of the *SYN* segment. The server could always refuse connection establishments from this particular client (e.g. due to security reasons) or the server may not have enough resources to accept a new TCP connection at that time. In this case, the server would reply with a TCP segment having its *RST* flag set and containing the *sequence number* of the received *SYN* segment as its *acknowledgment number*. This is illustrated in the figure below. We discuss the other utilizations of the TCP *RST* flag later (see *TCP connection release*).

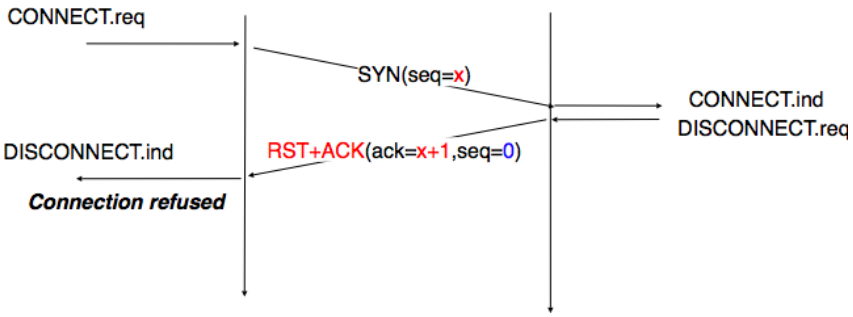


Figure 3.22: TCP connection establishment rejected by peer

TCP connection establishment can be described as the four state Finite State Machine shown below. In this FSM, *!X* (resp. *?Y*) indicates the transmission of segment *X* (resp. reception of segment *Y*) during the corresponding transition. *Init* is the initial state.

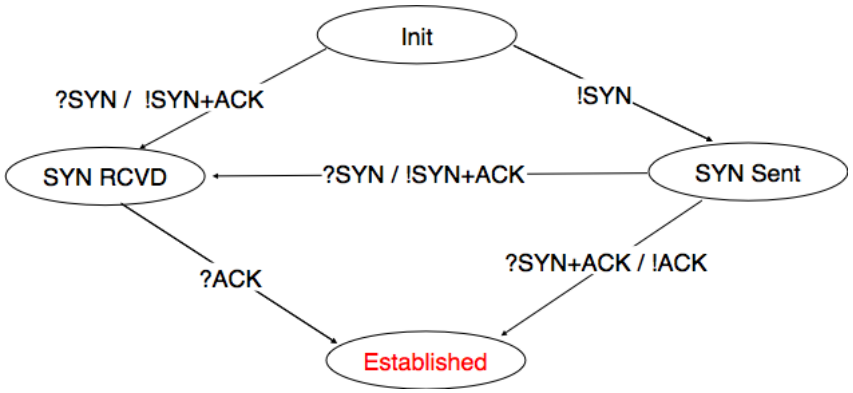


Figure 3.23: TCP FSM for connection establishment

A client host starts in the *Init* state. It then sends a *SYN* segment and enters the *SYN Sent* state where it waits for a *SYN+ACK* segment. Then, it replies with an *ACK* segment and enters the *Established* state where data can be exchanged. On the other hand, a server host starts in the *Init* state. When a server process starts to listen to a destination port, the underlying TCP entity creates a TCP control block and a queue to process incoming *SYN* segments. Upon reception of a *SYN* segment, the server's TCP entity replies with a *SYN+ACK* and enters the *SYN*

RCVD state. It remains in this state until it receives an *ACK* segment that acknowledges its *SYN+ACK* segment, with this it then enters the Established state.

Apart from these two paths in the TCP connection establishment FSM, there is a third path that corresponds to the case when both the client and the server send a *SYN* segment to open a TCP connection [16]. In this case, the client and the server send a *SYN* segment and enter the *SYN Sent* state. Upon reception of the *SYN* segment sent by the other host, they reply by sending a *SYN+ACK* segment and enter the *SYN RCVD* state. The *SYN+ACK* that arrives from the other host allows it to transition to the *Established* state. The figure below illustrates such a simultaneous establishment of a TCP connection.

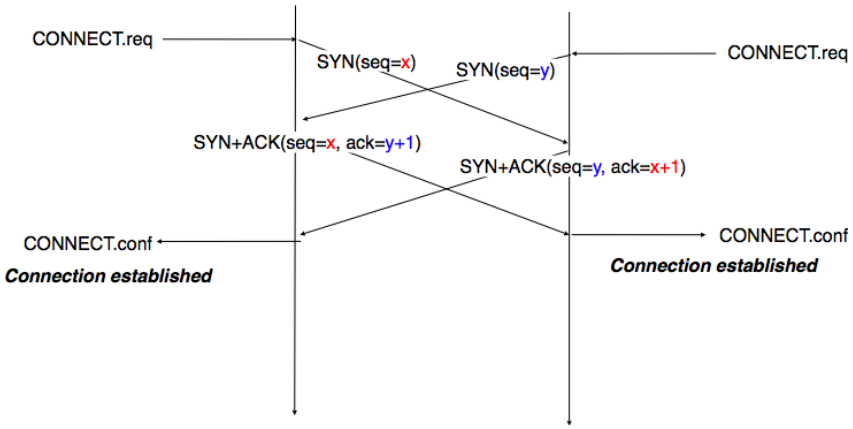


Figure 3.24: Simultaneous establishment of a TCP connection

[16] Of course, such a simultaneous TCP establishment can only occur if the source port chosen by the client is equal to the destination port chosen by the server. This may happen when a host can serve both as a client as a server or in peer-to-peer applications when the communicating hosts do not use ephemeral port numbers.

Denial of Service attacks

When a TCP entity opens a TCP connection, it creates a Transmission Control Block (*TCB*). The TCB contains the entire state that is maintained by the TCP entity for each TCP connection. During connection establishment, the TCB contains the local IP address, the remote IP address, the local port number, the remote port number, the current local sequence number, the last sequence number received from the remote entity. Until the mid 1990s, TCP implementations had a limit on the number of TCP connections that could be in the *SYN RCVD* state at a given time. Many implementations set this limit to about 100 TCBs. This limit was considered sufficient even for heavily load http servers given the small delay between the reception of a *SYN* segment and the reception of the *ACK* segment that terminates the establishment of the TCP connection. When the limit of 100 TCBs in the *SYN Rcvd* state is reached, the TCP entity discards all received TCP *SYN* segments that do not correspond to an existing TCB.
This limit of 100 TCBs in the *SYN Rcvd* state was chosen to protect the TCP entity from the risk of overloading its memory with too many TCBs in the *SYN Rcvd* state. However, it was also the reason for a new type of Denial of Service (DoS) attack **RFC 4987**. A DoS attack is defined as an attack where an attacker can render a resource unavailable in the network. For example, an attacker may cause a DoS attack on a 2 Mbps link used by a company by sending more than 2 Mbps of packets through this link. In this case, the DoS attack was more subtle. As a TCP entity discards all received *SYN* segments as soon as it has 100 TCBs in the *SYN Rcvd* state, an attacker simply had to send a few 100 *SYN* segments every second to a server and never reply to the received *SYN+ACK* segments. To avoid being caught, attackers were of course sending these *SYN* segments with a different address than their own IP address [a]. On most TCP implementations, once a TCB entered the *SYN Rcvd* state, it remained in this state for several seconds, waiting for a retransmission of the initial *SYN* segment. This attack was later called a *SYN flood* attack and the servers of the ISP named panix were among the first to be affected by this attack.
To avoid the *SYN flood* attacks, recent TCP implementations no longer enter the *SYN Rcvd* state upon reception of a *SYN segment*. Instead, they reply directly with a *SYN+ACK* segment and wait until the reception of a valid *ACK*. This implementation trick is only possible if the TCP implementation is able to verify that the received *ACK* segment acknowledges the *SYN+ACK* segment sent earlier without storing the initial sequence number of this *SYN+ACK* segment in a TCB. The solution to solve this problem, which is known as SYN cookies is to compute the 32 bits of the *ISN* as follows :

- the high order bits contain the low order bits of a counter that is incremented slowly
- the low order bits contain a hash value computed over the local and remote IP addresses and ports and a random secret only known to the server

The advantage of the SYN cookies is that by using them, the server does not need to create a *TCB* upon reception of the *SYN* segment and can still check the returned *ACK* segment by recomputing the *SYN cookie*. The main disadvantage is that they are not fully compatible with the TCP options. This is why they are not enabled by default on a typical system.

[a] Sending a packet with a different source IP address than the address allocated to the host is called sending a *spoofed packet*.

Retransmitting the first *SYN* segment

As IP provides an unreliable connectionless service, the *SYN* and *SYN+ACK* segments sent to open a TCP connection could be lost. Current TCP implementations start a retransmission timer when they send the first *SYN* segment. This timer is often set to three seconds for the first retransmission and then doubles after each retransmission **RFC 2988**. TCP implementations also enforce a maximum number of retransmissions for the initial *SYN* segment.

As explained earlier, TCP segments may contain an optional header extension. In the *SYN* and *SYN+ACK* segments, these options are used to negotiate some parameters and the utilisation of extensions to the basic TCP specification.

The first parameter which is negotiated during the establishment of a TCP connection is the Maximum Segment Size (*MSS*). The MSS is the size of the largest segment that a TCP entity is able to process. According to **RFC 879**, all TCP implementations must be able to receive TCP segments containing 536 bytes of payload. However, most TCP implementations are able to process larger segments. Such TCP implementations use the TCP MSS Option in the *SYN*/*SYN+ACK* segment to indicate the largest segment they are able to process. The MSS value indicates the maximum size of the payload of the TCP segments. The client (resp. server) stores in its *TCB* the

MSS value announced by the server (resp. the client).

Another utilisation of TCP options during connection establishment is to enable TCP extensions. For example, consider **RFC 1323** (which is discussed in *TCP reliable data transfer*). **RFC 1323** defines TCP extensions to support timestamps and larger windows. If the client supports **RFC 1323**, it adds a **RFC 1323** option to its *SYN* segment. If the server understands this **RFC 1323** option and wishes to use it, it replies with an **RFC 1323** option in the *SYN+ACK* segment and the extension defined in **RFC 1323** is used throughout the TCP connection. Otherwise, if the server's *SYN+ACK* does not contain the **RFC 1323** option, the client is not allowed to use this extension and the corresponding TCP header options throughout the TCP connection. TCP's option mechanism is flexible and it allows the extension of TCP while maintaining compatibility with older implementations.

The TCP options are encoded by using a Type Length Value format where :

- the first byte indicates the *type* of the option.
- the second byte indicates the total length of the option (including the first two bytes) in bytes
- the last bytes are specific for each type of option

RFC 793 defines the Maximum Segment Size (MSS) TCP option that must be understood by all TCP implementations. This option (type 2) has a length of 4 bytes and contains a 16 bits word that indicates the MSS supported by the sender of the *SYN* segment. The MSS option can only be used in TCP segments having the *SYN* flag set.

RFC 793 also defines two special options that must be supported by all TCP implementations. The first option is *End of option*. It is encoded as a single byte having value *0x00* and can be used to ensure that the TCP header extension ends on a 32 bits boundary. The *No-Operation* option, encoded as a single byte having value *0x01*, can be used when the TCP header extension contains several TCP options that should be aligned on 32 bit boundaries. All other options [17] are encoded by using the TLV format.

Note: The robustness principle

The handling of the TCP options by TCP implementations is one of the many applications of the *robustness principle* which is usually attributed to Jon Postel and is often quoted as *"Be liberal in what you accept, and conservative in what you send"* **RFC 1122**

Concerning the TCP options, the robustness principle implies that a TCP implementation should be able to accept TCP options that it does not understand, in particular in received *SYN* segments, and that it should be able to parse any received segment without crashing, even if the segment contains an unknown TCP option. Furthermore, a server should not send in the *SYN+ACK* segment or later, options that have not been proposed by the client in the *SYN* segment.

3.8.2 TCP reliable data transfer

The original TCP data transfer mechanisms were defined in **RFC 793**. Based on the experience of using TCP on the growing global Internet, this part of the TCP specification has been updated and improved several times, always while preserving the backward compatibility with older TCP implementations. In this section, we review the main data transfer mechanisms used by TCP.

TCP is a window-based transport protocol that provides a bi-directional byte stream service. This has several implications on the fields of the TCP header and the mechanisms used by TCP. The three fields of the TCP header are :

- *sequence number*. TCP uses a 32 bits sequence number. The *sequence number* placed in the header of a TCP segment containing data is the sequence number of the first byte of the payload of the TCP segment.
- *acknowledgement number*. TCP uses cumulative positive acknowledgements. Each TCP segment contains the *sequence number* of the next byte that the sender of the acknowledgement expects to receive from the remote host. In theory, the *acknowledgement number* is only valid if the *ACK* flag of the TCP header is set. In practice almost all [18] TCP segments have their *ACK* flag set.

[17] The full list of all TCP options may be found at http://www.iana.org/assignments/tcp-parameters/
[18] In practice, only the *SYN* segment do not have their *ACK* flag set.

- *window*. a TCP receiver uses this 16 bits field to indicate the current size of its receive window expressed in bytes.

Note: The Transmission Control Block

For each established TCP connection, a TCP implementation must maintain a Transmission Control Block (*TCB*). A TCB contains all the information required to send and receive segments on this connection **RFC 793**. This includes [19] :

- the local IP address
- the remote IP address
- the local TCP port number
- the remote TCP port number
- the current state of the TCP FSM
- the *maximum segment size* (MSS)
- *snd.nxt* : the sequence number of the next byte in the byte stream (the first byte of a new data segment that you send uses this sequence number)
- *snd.una* : the earliest sequence number that has been sent but has not yet been acknowledged
- *snd.wnd* : the current size of the sending window (in bytes)
- *rcv.nxt* : the sequence number of the next byte that is expected to be received from the remote host
- *rcv.wnd* : the current size of the receive window advertised by the remote host
- *sending buffer* : a buffer used to store all unacknowledged data
- *receiving buffer* : a buffer to store all data received from the remote host that has not yet been delivered to the user. Data may be stored in the *receiving buffer* because either it was not received in sequence or because the user is too slow to process it

---

The original TCP specification can be categorised as a transport protocol that provides a byte stream service and uses *go-back-n*.

To send new data on an established connection, a TCP entity performs the following operations on the corresponding TCB. It first checks that the *sending buffer* does not contain more data than the receive window advertised by the remote host (*rcv.wnd*). If the window is not full, up to *MSS* bytes of data are placed in the payload of a TCP segment. The *sequence number* of this segment is the sequence number of the first byte of the payload. It is set to the first available sequence number : *snd.nxt* and *snd.nxt* is incremented by the length of the payload of the TCP segment. The *acknowledgement number* of this segment is set to the current value of *rcv.nxt* and the *window* field of the TCP segment is computed based on the current occupancy of the *receiving buffer*. The data is kept in the *sending buffer* in case it needs to be retransmitted later.

When a TCP segment with the *ACK* flag set is received, the following operations are performed. *rcv.wnd* is set to the value of the *window* field of the received segment. The *acknowledgement number* is compared to *snd.una*. The newly acknowledged data is removed from the *sending buffer* and *snd.una* is updated. If the TCP segment contained data, the *sequence number* is compared to *rcv.nxt*. If they are equal, the segment was received in sequence and the data can be delivered to the user and *rcv.nxt* is updated. The contents of the *receiving buffer* is checked to see whether other data already present in this buffer can be delivered in sequence to the user. If so, *rcv.nxt* is updated again. Otherwise, the segment's payload is placed in the *receiving buffer*.

Segment transmission strategies

In a transport protocol such as TCP that offers a bytestream, a practical issue that was left as an implementation choice in **RFC 793** is to decide when a new TCP segment containing data must be sent. There are two simple and extreme implementation choices. The first implementation choice is to send a TCP segment as soon as the user

[19] A complete TCP implementation contains additional information in its TCB, notably to support the *urgent* pointer. However, this part of TCP is not discussed in this book. Refer to **RFC 793** and **RFC 2140** for more details about the TCB.

has requested the transmission of some data. This allows TCP to provide a low delay service. However, if the user is sending data one byte at a time, TCP would place each user byte in a segment containing 20 bytes of TCP header [20]. This is a huge overhead that is not acceptable in wide area networks. A second simple solution would be to only transmit a new TCP segment once the user has produced MSS bytes of data. This solution reduces the overhead, but at the cost of a potentially very high delay.

An elegant solution to this problem was proposed by John Nagle in **RFC 896**. John Nagle observed that the overhead caused by the TCP header was a problem in wide area connections, but less in local area connections where the available bandwidth is usually higher. He proposed the following rules to decide to send a new data segment when a new data has been produced by the user or a new ack segment has been received

```
if rcv.wnd>= MSS and len(data) >= MSS :
  send one MSS-sized segment
else
  if there are unacknowledged data:
    place data in buffer until acknowledgement has been received
  else
    send one TCP segment containing all buffered data
```

The first rule ensures that a TCP connection used for bulk data transfer always sends full TCP segments. The second rule sends one partially filled TCP segment every round-trip-time.

This algorithm, called the Nagle algorithm, takes a few lines of code in all TCP implementations. These lines of code have a huge impact on the packets that are exchanged in TCP/IP networks. Researchers have analysed the distribution of the packet sizes by capturing and analysing all the packets passing through a given link. These studies have shown several important results :

- in TCP/IP networks, a large fraction of the packets are TCP segments that contain only an acknowledgement. These packets usually account for 40-50% of the packets passing through the studied link
- in TCP/IP networks, most of the bytes are exchanged in long packets, usually packets containing about 1440 bytes of payload which is the default MSS for hosts attached to an Ethernet network, the most popular type of LAN

Recent measurements indicate that these packet size distributions are still valid in today's Internet, although the packet distribution tends to become bimodal with small packets corresponding to TCP pure acks and large 1440-bytes packets carrying most of the user data [SMASU2012].

3.8.3 TCP windows

From a performance point of view, one of the main limitations of the original TCP specification is the 16 bits *window* field in the TCP header. As this field indicates the current size of the receive window in bytes, it limits the TCP receive window at 65535 bytes. This limitation was not a severe problem when TCP was designed since at that time high-speed wide area networks offered a maximum bandwidth of 56 kbps. However, in today's network, this limitation is not acceptable anymore. The table below provides the rough [21] maximum throughput that can be achieved by a TCP connection with a 64 KBytes window in function of the connection's round-trip-time

RTT	Maximum Throughput
1 msec	524 Mbps
10 msec	52.4 Mbps
100 msec	5.24 Mbps
500 msec	1.05 Mbps

To solve this problem, a backward compatible extension that allows TCP to use larger receive windows was proposed in **RFC 1323**. Today, most TCP implementations support this option. The basic idea is that instead of storing *snd.wnd* and *rcv.wnd* as 16 bits integers in the *TCB*, they should be stored as 32 bits integers. As the TCP segment header only contains 16 bits to place the window field, it is impossible to copy the value of *snd.wnd* in each sent TCP segment. Instead the header contains *snd.wnd* >> S where S is the scaling factor ($0 \leq S \leq 14$)

[20] This TCP segment is then placed in an IP header. We describe IPv6 in the next chapter. The minimum size of the IPv6 (resp. IPv4) header is 40 bytes (resp. 20 bytes).

[21] A precise estimation of the maximum bandwidth that can be achieved by a TCP connection should take into account the overhead of the TCP and IP headers as well.

negotiated during connection establishment. The client adds its proposed scaling factor as a TCP option in the *SYN* segment. If the server supports **RFC 1323**, it places in the *SYN+ACK* segment the scaling factor that it uses when advertising its own receive window. The local and remote scaling factors are included in the *TCB*. If the server does not support **RFC 1323**, it ignores the received option and no scaling is applied.

By using the window scaling extensions defined in **RFC 1323**, TCP implementations can use a receive buffer of up to 1 GByte. With such a receive buffer, the maximum throughput that can be achieved by a single TCP connection becomes :

RTT	Maximum Throughput
1 msec	8590 Gbps
10 msec	859 Gbps
100 msec	86 Gbps
500 msec	17 Gbps

These throughputs are acceptable in today's networks. However, there are already servers having 10 Gbps interfaces... Early TCP implementations had fixed receiving and sending buffers [22]. Today's high performance implementations are able to automatically adjust the size of the sending and receiving buffer to better support high bandwidth flows [SMM1998]

3.8.4 TCP's retransmission timeout

In a go-back-n transport protocol such as TCP, the retransmission timeout must be correctly set in order to achieve good performance. If the retransmission timeout expires too early, then bandwidth is wasted by retransmitting segments that have already been correctly received; whereas if the retransmission timeout expires too late, then bandwidth is wasted because the sender is idle waiting for the expiration of its retransmission timeout.

A good setting of the retransmission timeout clearly depends on an accurate estimation of the round-trip-time of each TCP connection. The round-trip-time differs between TCP connections, but may also change during the lifetime of a single connection. For example, the figure below shows the evolution of the round-trip-time between two hosts during a period of 45 seconds.

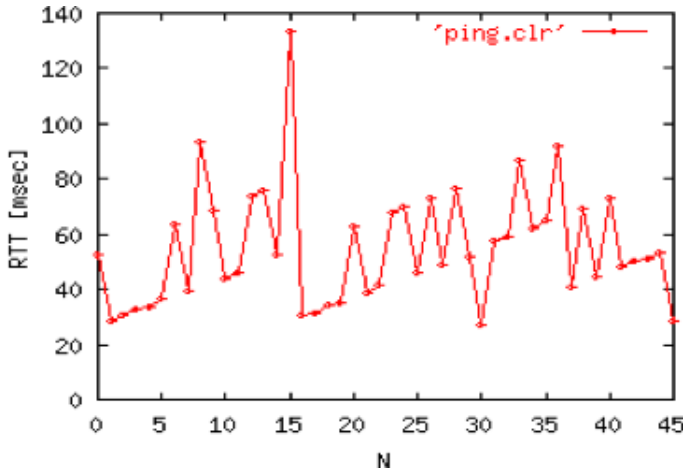


Figure 3.25: Evolution of the round-trip-time between two hosts

The easiest solution to measure the round-trip-time on a TCP connection is to measure the delay between the transmission of a data segment and the reception of a corresponding acknowledgement [23]. As illustrated in the figure below, this measurement works well when there are no segment losses.

However, when a data segment is lost, as illustrated in the bottom part of the figure, the measurement is ambiguous as the sender cannot determine whether the received acknowledgement was triggered by the first transmission of

[22] See http://fasterdata.es.net/tuning.html for more information on how to tune a TCP implementation

[23] In theory, a TCP implementation could store the timestamp of each data segment transmitted and compute a new estimate for the round-trip-time upon reception of the corresponding acknowledgement. However, using such frequent measurements introduces a lot of noise in practice and many implementations still measure the round-trip-time once per round-trip-time by recording the transmission time of one segment at a time **RFC 2988**

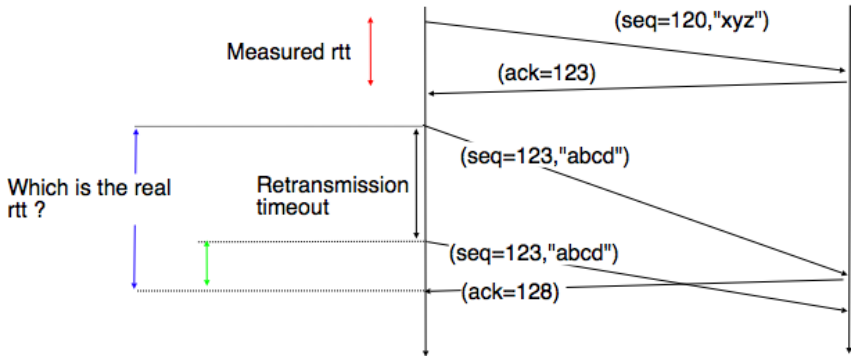


Figure 3.26: How to measure the round-trip-time ?

segment *123* or its retransmission. Using incorrect round-trip-time estimations could lead to incorrect values of the retransmission timeout. For this reason, Phil Karn and Craig Partridge proposed, in [KP91], to ignore the round-trip-time measurements performed during retransmissions.

To avoid this ambiguity in the estimation of the round-trip-time when segments are retransmitted, recent TCP implementations rely on the *timestamp option* defined in **RFC 1323**. This option allows a TCP sender to place two 32 bit timestamps in each TCP segment that it sends. The first timestamp, TS Value (*TSval*) is chosen by the sender of the segment. It could for example be the current value of its real-time clock [24]. The second value, TS Echo Reply (*TSecr*), is the last *TSval* that was received from the remote host and stored in the *TCB*. The figure below shows how the utilization of this timestamp option allows for the disambiguation of the round-trip-time measurement when there are retransmissions.

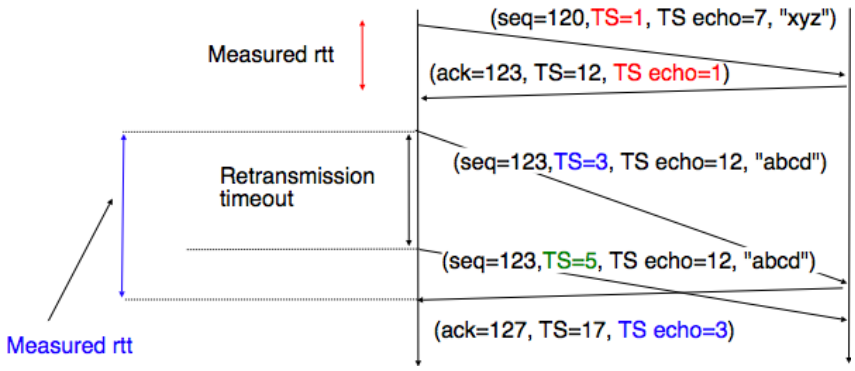


Figure 3.27: Disambiguating round-trip-time measurements with the **RFC 1323** timestamp option

Once the round-trip-time measurements have been collected for a given TCP connection, the TCP entity must compute the retransmission timeout. As the round-trip-time measurements may change during the lifetime of a connection, the retransmission timeout may also change. At the beginning of a connection [25] , the TCP entity that sends a *SYN* segment does not know the round-trip-time to reach the remote host and the initial retransmission timeout is usually set to 3 seconds **RFC 2988**.

The original TCP specification proposed in **RFC 793** to include two additional variables in the TCB :

- *srtt* : the smoothed round-trip-time computed as $srrt = (\alpha \times srtt) + ((1 - \alpha) \times rtt)$ where *rtt* is the round-trip-time measured according to the above procedure and α a smoothing factor (e.g. 0.8 or 0.9)
- *rto* : the retransmission timeout is computed as $rto = min(60, max(1, \beta \times srtt))$ where β is used to take into account the delay variance (value : 1.3 to 2.0). The *60* and *1* constants are used to ensure that the *rto* is not larger than one minute nor smaller than 1 second.

[24] Some security experts have raised concerns that using the real-time clock to set the *TSval* in the timestamp option can leak information such as the system's up-time. Solutions proposed to solve this problem may be found in [CNPI09]

[25] As a TCP client often establishes several parallel or successive connections with the same server, **RFC 2140** has proposed to reuse for a new connection some information that was collected in the TCB of a previous connection, such as the measured rtt. However, this solution has not been widely implemented.

However, in practice, this computation for the retransmission timeout did not work well. The main problem was that the computed *rto* did not correctly take into account the variations in the measured round-trip-time. *Van Jacobson* proposed in his seminal paper [Jacobson1988] an improved algorithm to compute the *rto* and implemented it in the BSD Unix distribution. This algorithm is now part of the TCP standard **RFC 2988**.

Jacobson's algorithm uses two state variables, *srtt* the smoothed *rtt* and *rttvar* the estimation of the variance of the *rtt* and two parameters : α and β. When a TCP connection starts, the first *rto* is set to *3* seconds. When a first estimation of the *rtt* is available, the *srtt*, *rttvar* and *rto* are computed as follows :

```
srtt=rtt
rttvar=rtt/2
rto=srtt+4*rttvar
```

Then, when other rtt measurements are collected, *srtt* and *rttvar* are updated as follows :

$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt|$$

$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt$$

$$rto = srtt + 4 \times rttvar$$

The proposed values for the parameters are $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$. This allows a TCP implementation, implemented in the kernel, to perform the *rtt* computation by using shift operations instead of the more costly floating point operations [Jacobson1988]. The figure below illustrates the computation of the *rto* upon *rtt* changes.

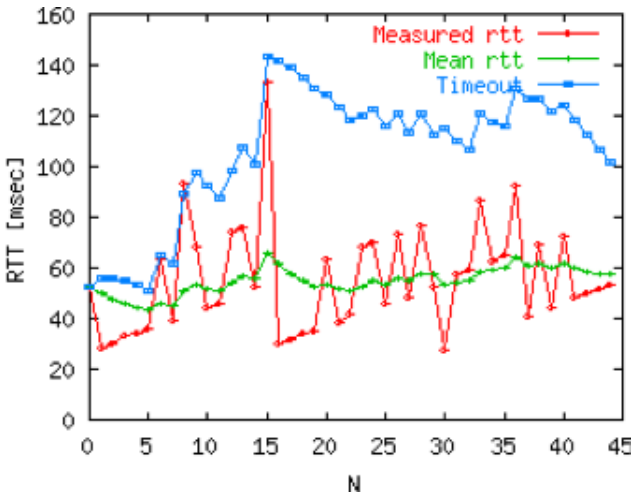


Figure 3.28: Example computation of the *rto*

3.8.5 Advanced retransmission strategies

The default go-back-n retransmission strategy was defined in **RFC 793**. When the retransmission timer expires, TCP retransmits the first unacknowledged segment (i.e. the one having sequence number *snd.una*). After each expiration of the retransmission timeout, **RFC 2988** recommends to double the value of the retransmission timeout. This is called an *exponential backoff*. This doubling of the retransmission timeout after a retransmission was included in TCP to deal with issues such as network/receiver overload and incorrect initial estimations of the retransmission timeout. If the same segment is retransmitted several times, the retransmission timeout is doubled after every retransmission until it reaches a configured maximum. **RFC 2988** suggests a maximum retransmission timeout of at least 60 seconds. Once the retransmission timeout reaches this configured maximum, the remote host is considered to be unreachable and the TCP connection is closed.

This retransmission strategy has been refined based on the experience of using TCP on the Internet. The first refinement was a clarification of the strategy used to send acknowledgements. As TCP uses piggybacking, the easiest and less costly method to send acknowledgements is to place them in the data segments sent in the other direction. However, few application layer protocols exchange data in both directions at the same time and thus this method rarely works. For an application that is sending data segments in one direction only, the remote TCP entity

returns empty TCP segments whose only useful information is their acknowledgement number. This may cause a large overhead in wide area network if a pure *ACK* segment is sent in response to each received data segment. Most TCP implementations use a *delayed acknowledgement* strategy. This strategy ensures that piggybacking is used whenever possible, otherwise pure *ACK* segments are sent for every second received data segments when there are no losses. When there are losses or reordering, *ACK* segments are more important for the sender and they are sent immediately **RFC 813 RFC 1122**. This strategy relies on a new timer with a short delay (e.g. 50 milliseconds) and one additional flag in the TCB. It can be implemented as follows :

```
reception of a data segment:
   if pkt.seq==rcv.nxt:   # segment received in sequence
      if delayedack :
         send pure ack segment
         cancel acktimer
         delayedack=False
      else:
         delayedack=True
         start acktimer
   else:                      # out of sequence segment
      send pure ack segment
      if delayedack:
         delayedack=False
         cancel acktimer

transmission of a data segment:  # piggyback ack
   if delayedack:
      delayedack=False
      cancel acktimer

acktimer expiration:
   send pure ack segment
   delayedack=False
```

Due to this delayed acknowledgement strategy, during a bulk transfer, a TCP implementation usually acknowledges every second TCP segment received.

The default go-back-n retransmission strategy used by TCP has the advantage of being simple to implement, in particular on the receiver side, but when there are losses, a go-back-n strategy provides a lower performance than a selective repeat strategy. The TCP developers have designed several extensions to TCP to allow it to use a selective repeat strategy while maintaining backward compatibility with older TCP implementations. These TCP extensions assume that the receiver is able to buffer the segments that it receives out-of-sequence.

The first extension that was proposed is the fast retransmit heuristic. This extension can be implemented on TCP senders and thus does not require any change to the protocol. It only assumes that the TCP receiver is able to buffer out-of-sequence segments.

From a performance point of view, one issue with TCP's *retransmission timeout* is that when there are isolated segment losses, the TCP sender often remains idle waiting for the expiration of its retransmission timeouts. Such isolated losses are frequent in the global Internet [Paxson99]. A heuristic to deal with isolated losses without waiting for the expiration of the retransmission timeout has been included in many TCP implementations since the early 1990s. To understand this heuristic, let us consider the figure below that shows the segments exchanged over a TCP connection when an isolated segment is lost.

As shown above, when an isolated segment is lost the sender receives several *duplicate acknowledgements* since the TCP receiver immediately sends a pure acknowledgement when it receives an out-of-sequence segment. A duplicate acknowledgement is an acknowledgement that contains the same *acknowledgement number* as a previous segment. A single duplicate acknowledgement does not necessarily imply that a segment was lost, as a simple reordering of the segments may cause duplicate acknowledgements as well. Measurements [Paxson99] have shown that segment reordering is frequent in the Internet. Based on these observations, the *fast retransmit* heuristic has been included in most TCP implementations. It can be implemented as follows :

```
ack arrival:
    if tcp.ack==snd.una:    # duplicate acknowledgement
       dupacks++
```

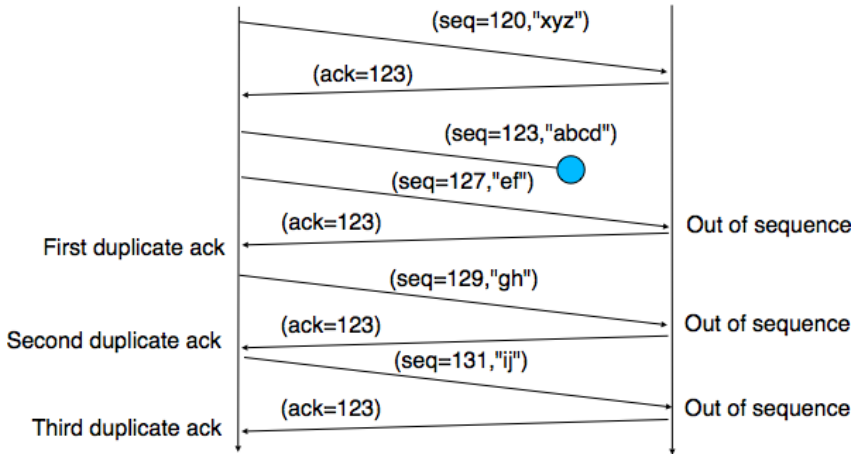


Figure 3.29: Detecting isolated segment losses

```
    if dupacks==3:
        retransmit segment(snd.una)
else:
    dupacks=0
    # process acknowledgement
```

This heuristic requires an additional variable in the TCB (*dupacks*). Most implementations set the default number of duplicate acknowledgements that trigger a retransmission to 3. It is now part of the standard TCP specification **RFC 2581**. The *fast retransmit* heuristic improves the TCP performance provided that isolated segments are lost and the current window is large enough to allow the sender to send three duplicate acknowledgements.

The figure below illustrates the operation of the *fast retransmit* heuristic.

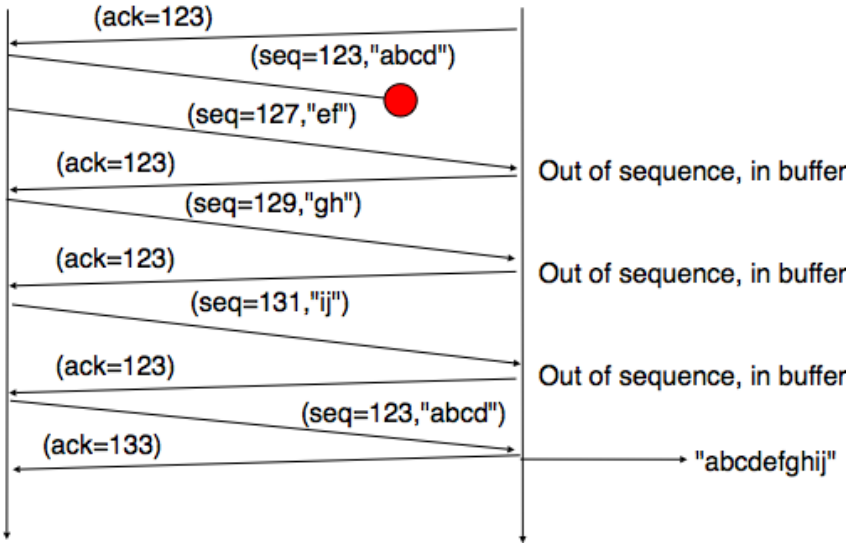


Figure 3.30: TCP fast retransmit heuristics

When losses are not isolated or when the windows are small, the performance of the *fast retransmit* heuristic decreases. In such environments, it is necessary to allow a TCP sender to use a selective repeat strategy instead of the default go-back-n strategy. Implementing selective-repeat requires a change to the TCP protocol as the receiver needs to be able to inform the sender of the out-of-order segments that it has already received. This can be done by using the Selective Acknowledgements (SACK) option defined in **RFC 2018**. This TCP option is negotiated during the establishment of a TCP connection. If both TCP hosts support the option, SACK blocks can be attached by the receiver to the segments that it sends. SACK blocks allow a TCP receiver to indicate the blocks of data that it has received correctly but out of sequence. The figure below illustrates the utilisation of the SACK

blocks.

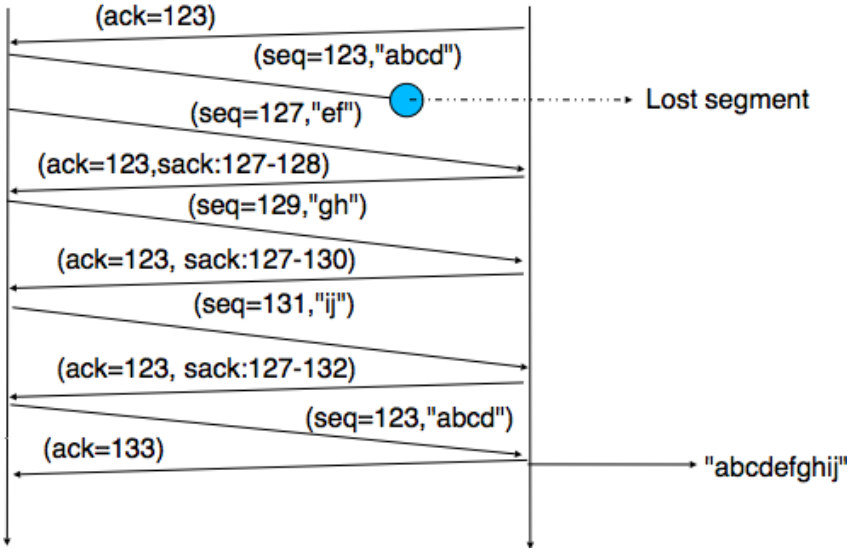


Figure 3.31: TCP selective acknowledgements

An SACK option contains one or more blocks. A block corresponds to all the sequence numbers between the *left edge* and the *right edge* of the block. The two edges of the block are encoded as 32 bit numbers (the same size as the TCP sequence number) in an SACK option. As the SACK option contains one byte to encode its type and one byte for its length, a SACK option containing b blocks is encoded as a sequence of $2 + 8 \times b$ bytes. In practice, the size of the SACK option can be problematic as the optional TCP header extension cannot be longer than 44 bytes. As the SACK option is usually combined with the **RFC 1323** timestamp extension, this implies that a TCP segment cannot usually contain more than three SACK blocks. This limitation implies that a TCP receiver cannot always place in the SACK option that it sends, information about all the received blocks.

To deal with the limited size of the SACK option, a TCP receiver currently having more than 3 blocks inside its receiving buffer must select the blocks to place in the SACK option. A good heuristic is to put in the SACK option the blocks that have most recently changed, as the sender is likely to be already aware of the older blocks.

When a sender receives an SACK option indicating a new block and thus a new possible segment loss, it usually does not retransmit the missing segments immediately. To deal with reordering, a TCP sender can use a heuristic similar to *fast retransmit* by retransmitting a gap only once it has received three SACK options indicating this gap. It should be noted that the SACK option does not supersede the *acknowledgement number* of the TCP header. A TCP sender can only remove data from its sending buffer once they have been acknowledged by TCP's cumulative acknowledgements. This design was chosen for two reasons. First, it allows the receiver to discard parts of its receiving buffer when it is running out of memory without loosing data. Second, as the SACK option is not transmitted reliably, the cumulative acknowledgements are still required to deal with losses of *ACK* segments carrying only SACK information. Thus, the SACK option only serves as a hint to allow the sender to optimise its retransmissions.

3.8.6 TCP connection release

TCP, like most connection-oriented transport protocols, supports two types of connection releases :

- graceful connection release, where each TCP user can release its own direction of data transfer after having transmitted all data
- abrupt connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection (e.g. because the remote host does not reply anymore or due to lack of resources)

The abrupt connection release mechanism is very simple and relies on a single segment having the *RST* bit set. A TCP segment containing the *RST* bit can be sent for the following reasons :

- a non-*SYN* segment was received for a non-existing TCP connection **RFC 793**
- by extension, some implementations respond with an *RST* segment to a segment that is received on an existing connection but with an invalid header **RFC 3360**. This causes the corresponding connection to be closed and has caused security attacks **RFC 4953**
- by extension, some implementations send an *RST* segment when they need to close an existing TCP connection (e.g. because there are not enough resources to support this connection or because the remote host is considered to be unreachable). Measurements have shown that this usage of TCP *RST* is widespread [AW05]

When an *RST* segment is sent by a TCP entity, it should contain the current value of the *sequence number* for the connection (or 0 if it does not belong to any existing connection) and the *acknowledgement number* should be set to the next expected in-sequence *sequence number* on this connection.

Note: TCP *RST* wars

TCP implementers should ensure that two TCP entities never enter a TCP *RST* war where host *A* is sending a *RST* segment in response to a previous *RST* segment that was sent by host *B* in response to a TCP *RST* segment sent by host *A* ... To avoid such an infinite exchange of *RST* segments that do not carry data, a TCP entity is *never* allowed to send a *RST* segment in response to another *RST* segment.

The normal way of terminating a TCP connection is by using the graceful TCP connection release. This mechanism uses the *FIN* flag of the TCP header and allows each host to release its own direction of data transfer. As for the *SYN* flag, the utilisation of the *FIN* flag in the TCP header consumes one sequence number. The figure *FSM for TCP connection release* shows the part of the TCP FSM used when a TCP connection is released.

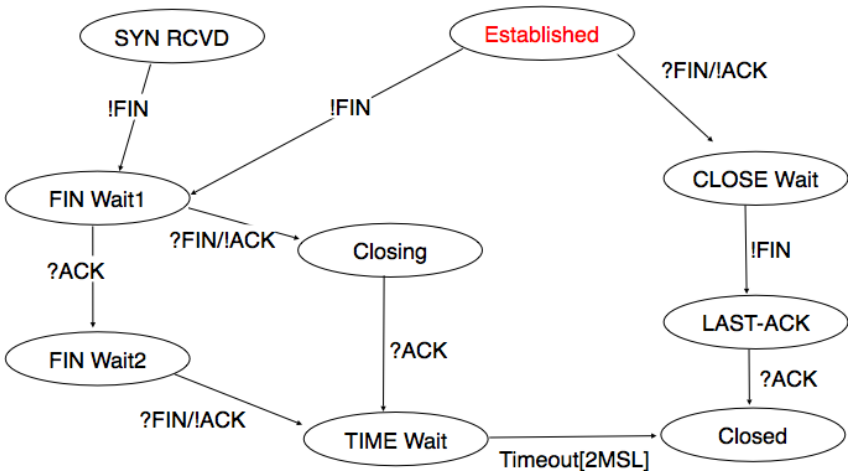


Figure 3.32: FSM for TCP connection release

Starting from the *Established* state, there are two main paths through this FSM.

The first path is when the host receives a segment with sequence number *x* and the *FIN* flag set. The utilisation of the *FIN* flag indicates that the byte before *sequence number x* was the last byte of the byte stream sent by the remote host. Once all of the data has been delivered to the user, the TCP entity sends an *ACK* segment whose *ack* field is set to $(x + 1)\ mod\ 2^{32}$ to acknowledge the *FIN* segment. The *FIN* segment is subject to the same retransmission mechanisms as a normal TCP segment. In particular, its transmission is protected by the retransmission timer. At this point, the TCP connection enters the *CLOSE_WAIT* state. In this state, the host can still send data to the remote host. Once all its data have been sent, it sends a *FIN* segment and enter the *LAST_ACK* state. In this state, the TCP entity waits for the acknowledgement of its *FIN* segment. It may still retransmit unacknowledged data segments e.g. if the retransmission timer expires. Upon reception of the acknowledgement for the *FIN* segment, the TCP connection is completely closed and its *TCB* can be discarded.

The second path is when the host has transmitted all data. Assume that the last transmitted sequence number is z. Then, the host sends a *FIN* segment with sequence number $(z + 1) mod 2^{32}$ and enters the *FIN_WAIT1* state. It this state, it can retransmit unacknowledged segments but cannot send new data segments. It waits for

an acknowledgement of its *FIN* segment (i.e. sequence number $(z+1) mod 2^{32}$), but may receive a *FIN* segment sent by the remote host. In the first case, the TCP connection enters the *FIN_WAIT2* state. In this state, new data segments from the remote host are still accepted until the reception of the *FIN* segment. The acknowledgement for this *FIN* segment is sent once all data received before the *FIN* segment have been delivered to the user and the connection enters the *TIME_WAIT* state. In the second case, a *FIN* segment is received and the connection enters the *Closing* state once all data received from the remote host have been delivered to the user. In this state, no new data segments can be sent and the host waits for an acknowledgement of its *FIN* segment before entering the *TIME_WAIT* state.

The *TIME_WAIT* state is different from the other states of the TCP FSM. A TCP entity enters this state after having sent the last *ACK* segment on a TCP connection. This segment indicates to the remote host that all the data that it has sent have been correctly received and that it can safely release the TCP connection and discard the corresponding *TCB*. After having sent the last *ACK* segment, a TCP connection enters the *TIME_WAIT* and remains in this state for $2 * MSL$ seconds. During this period, the TCB of the connection is maintained. This ensures that the TCP entity that sent the last *ACK* maintains enough state to be able to retransmit this segment if this *ACK* segment is lost and the remote host retransmits its last *FIN* segment or another one. The delay of $2 * MSL$ seconds ensures that any duplicate segments on the connection would be handled correctly without causing the transmission of an *RST* segment. Without the *TIME_WAIT* state and the $2 * MSL$ seconds delay, the connection release would not be graceful when the last *ACK* segment is lost.

Note: TIME_WAIT on busy TCP servers

The $2 * MSL$ seconds delay in the *TIME_WAIT* state is an important operational problem on servers having thousands of simultaneously opened TCP connections [FTY99]. Consider for example a busy web server that processes 10.000 TCP connections every second. If each of these connections remain in the *TIME_WAIT* state for 4 minutes, this implies that the server would have to maintain more than 2 million TCBs at any time. For this reason, some TCP implementations prefer to perform an abrupt connection release by sending a *RST* segment to close the connection [AW05] and immediately discard the corresponding *TCB*. However, if the *RST* segment is lost, the remote host continues to maintain a *TCB* for a connection no longer exists. This optimisation reduces the number of TCBs maintained by the host sending the *RST* segment but at the potential cost of increased processing on the remote host when the *RST* segment is lost.

3.9 The Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) **RFC 4960** was defined in the late 1990s, early 2000s as an alternative to the Transmission Control Protocol. The initial design of SCTP was motivated by the need to efficiently support signaling protocols that are used in Voice over IP networks. These signaling protocols allow to create, control and terminate voice calls. They have different requirements than regular applications like email, http that are well served by TCP's bytestream service.

One of the first motivations for SCTP was the need to efficiently support multihomed hosts, i.e. hosts equipped with two or more network interfaces. The Internet architecture and TCP in particular were not designed to handle efficiently such hosts. On the Internet, when a host is multihomed, it needs to use several IP addresses, one per interface. Consider for example a smartphone connected to both WiFi and 3G. The smartphone uses one IP address on its WiFi interface and a different one on its 3G interface. When it establishes a TCP connection through its WiFi interface, this connection is bound to the IP address of the WiFi interface and the segments corresponding to this connection must always be transmitted through the WiFi interface. If the WiFi interface is not anymore connected to the network (e.g. because the smartphone user moved), the TCP connection stops and need to be explicitly reestablished by the application over the 3G interface. SCTP was designed to support seamless failover from one interface to another during the lifetime of a connection. This is a major change compared to TCP [26].

A second motivation for designing SCTP was to provide a different service than TCP's bytestream to the applications. A first service brought by SCTP is the ability exchange messages instead of only a stream of bytes. This is a major modification which has many benefits for applications. Unfortunately, there are many deployed applications that have been designed under the assumption of the bytestream service. Rewriting them to benefit from

[26] Recently, the IETF approved the Multipath TCP extension **RFC 6824** that allows TCP to efficiently support multihomed hosts. A detailed presentation of Multipath TCP is outside the scope of this document, but may be found in [RIB2013] and on http://www.multipath-tcp.org

a message-mode service will require a lot of effort. It seems unlikely as of this writing to expect old applications to be rewritten to fully support SCTP and use it. However, some new applications are considering using SCTP instead of TCP. Voice over IP signaling protocols are a frequently cited example. The Real-Time Communication in Web-browsers working group is also considering the utilization of SCTP for some specific data channels [JLT2013]. From a service viewpoint, a second advantage of SCTP compared to TCP is its ability to support several simultaneous streams. Consider a web application that needs to retrieve five objects from a remote server. With TCP, one possibility is to open one TCP connection for each object, send a request over each connection and retrieve one object per connection. This is the solution used by HTTP/1.0 as explained earlier. The drawback of this approach is that the application needs to maintain several concurrent TCP connections. Another solution is possible with HTTP/1.1 [NGB+1997] . With HTTP/1.1, the client can use pipelining to send several HTTP Requests without waiting for the answer of each request. The server replies to these requests in sequence, one after the other. If the server replies to the requests in the sequence, this may lead to *head-of-line blocking* problems. Consider that the objects different sizes. The first object is a large 10 MBytes image while the other objects are small javascript files. In this case, delivering the objects in sequence will cause a very long delay for the javascript files since they will only be transmitted once the large image has been sent.

With SCTP, *head-of-line blocking* can be mitigated. SCTP can open a single connection and divide it in five logical streams so that the five objects are sent in parallel over the single connection. SCTP controls the transmission of the segments over the connection and ensures that the data is delivered efficiently to the application. In the example above, the small javascript files could be delivered as independent messages before the large image.

Another extension to SCTP **RFC 3758** supports partially-reliable delivery. With this extension, an SCTP sender can be instructed to "expire" data based on one of several events, such as a timeout, the sender can signal the SCTP receiver to move on without waiting for the *expired* data. This partially reliable service could be useful to provide timed delivery for example. With this service, there is an upper limit on the time required to deliver a message to the receiver. If the transport layer cannot deliver the data within the specified delay, the data is discarded by the sender without causing any stall in the stream.

3.9.1 SCTP segments

SCTP entities exchange segments. In contrast with TCP that uses a simple segment format with a limited space for the options, the designers of SCTP have learned from the experience of using and extending TCP during almost two decades. An SCTP segment is always composed of a fixed size *common header* followed by a variable number of chunks. The *common header* is 12 bytes long and contains four fields. The first two fields and the *Source* and *Destination* ports that allow to identify the SCTP connection. The *Verification tag* is a field that is set during connection establishment and placed in all segments exchanged during a connection to validate the received segments. The last field of the common header is a 32bits CRC. This CRC is computed over the entire segment (common header and all chunks). It is computed by the sender and verified by the receiver. Note that although this field is named *Checksum* **RFC 4960** it is computed by using the CRC-32 algorithm that has much stronger error detection capabilities than the Internet checksum algorithm used by TCP [SGP98].

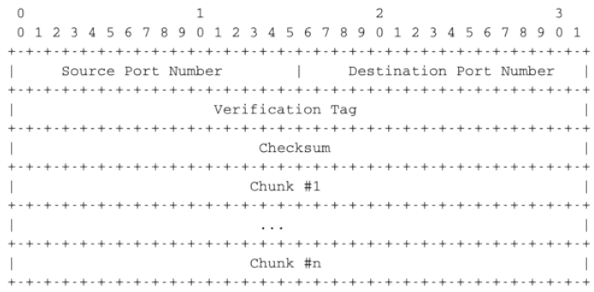


Figure 3.33: The SCTP segment format

The SCTP chunks play a key role in the extensibility of SCTP. In TCP, the extensibility of the protocol is provided by the utilisation of options that allow to extend the TCP header. However, even with options the TCP header cannot be longer than 64 bytes. This severely restricts our ability to significantly extend TCP [RIB2013]. In SCTP, a segment, which must be transmitted inside a single network packet, like a TCP segment, can contain a variable number of chunks and each chunk has a variable length. The payload that contains the data provided by

the user is itself a chunk. The SCTP chunks are a good example of a protocol format that can be easily extended. Each chunk is encoded as four fields shown in the figure below.

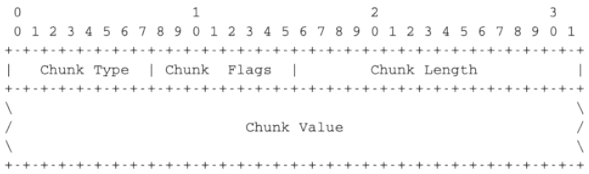


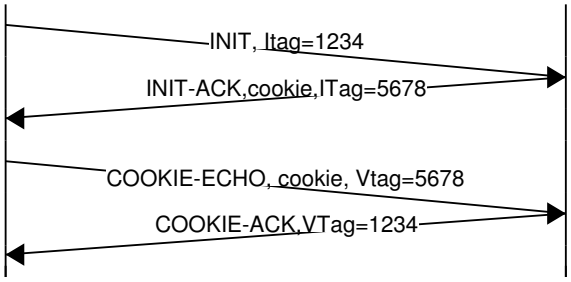
Figure 3.34: The SCTP chunk format

The first byte indicates the chunk type. 15 chunk types are defined in **RFC 4960** and new ones can be easily added. The low-order 16 bits of the first word contain the length of the chunk in bytes. The presence of the length field ensures that any SCTP implementation will be able to correctly parse any received SCTP segment, even if it contains unknown or new chunks. To further ease the processing of unknown chunks, **RFC 4960** uses the first two bits of the chunk type to specify how an SCTP implementation should react when receiving an unknown chunk. If the two high-order bits of the type of the unknown are set to `00`, then the entire SCTP segment containing the chunk should be discarded. It is expected that all SCTP implementations are capable of recognizing and processing these chunks. If the first two bits of the chunk type are set to `01` the SCTP segment must be discarded and an error reported to the sender. If the two high order bits of the type are set to `10` (resp. `11`), the chunk must be ignored, but the processing of the other chunks in the SCTP segment continues (resp. and an error is reported). The second byte contains flags that are used for some chunks.

3.9.2 Connection establishment

The SCTP protocol was designed shortly after the first Denial of Service attacks against the three-way handshake used by TCP. These attacks have heavily influenced the connection establishment mechanism chosen for SCTP. An SCTP connection is established by using a four-way handshake.

The SCTP connection establishment uses several chunks to specify the values of some parameters that are exchanged. The SCTP four-way handshake uses four segments as shown in the figure below.



The first segment contains the `INIT` chunk. To establish an SCTP connection with a server, the client first creates some local state for this connection. The most important parameter of the `INIT` chunk is the *Initiation tag*. This value is a random number that is used to identify the connection on the client host for its entire lifetime. This *Initiation tag* is placed as the *Verification tag* in all segments sent by the server. This is an important change compared to TCP where only the source and destination ports are used to identify a given connection. The *INIT*' chunk may also contain the other addresses owned by the client. The server responds by sending an `INIT-ACK` chunk. This chunk also contains an *Initiation tag* chosen by the server and a copy of the *Initiation tag* chosen by the client. The `INIT` and `INIT-ACK` chunks also contain an initial sequence number. A key difference between TCP's three-way handshake and SCTP's four-way handshake is that an SCTP server does not create any state when receiving an `INIT` chunk. For this, the server places inside the `INIT-ACK` reply a *State cookie* chunk. This *State cookie* is an opaque block of data that contains information computed from the `INIT` and `INIT-ACK` chunks that the server would have had stored locally, some lifetime information and a signature. The format of the *State cookie* is flexible and the server could in theory place almost any information inside this chunk. The only requirement is that the *State cookie* must be echoed back by the client to confirm the establishment of the connection. Upon reception of the `COOKIE-ECHO` chunk, the server verifies the signature of the *State cookie*.

The client may provide some user data and an initial sequence number inside the COOKIE-ECHO chunk. The server then responds with a COOKIE-ACK chunk that acknowledges the COOKIE-ECHO chunk. The SCTP connection between the client and the server is now established. This four-way handshake is both more secure and more flexible than the three-way handshake used by TCP. The detailed formats of the INIT, INIT-ACK, COOKIE-ECHO and COOKIE-ACK chunks may be found in **RFC 4960**.

3.9.3 Reliable data transfert

SCTP provides a slightly different service model **RFC 3286**. Once an SCTP connection has been established, the communicating hosts can access two or more message streams. A message stream is a stream of variable length messages. Each message is composed of an integer number of bytes. The connection-oriented service provided by SCTP preserves the message boundaries. It is interesting to analyze how SCTP provides the message-mode service and contrast SCTP with TCP. Data is exchanged by using data chunks. The format of these chunks is shown in the figure below.

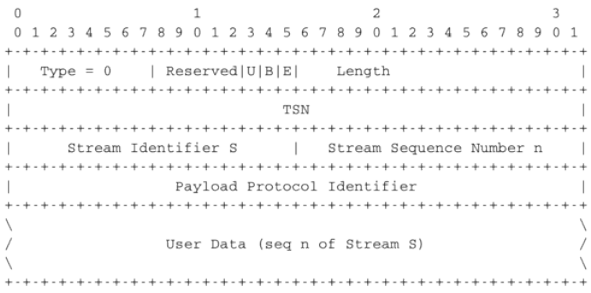


Figure 3.35: The SCTP DATA chunk

An SCTP DATA chunk contains several fields as shown in the figure above. The detailed description of this chunk may be found in **RFC 4960**. For simplicity, we focus on an SCTP connection that supports a single stream. SCTP uses the *Transmission Sequence Number* (TSN) to sequence the data chunks that are sent. The TSN is also used to reorder the received DATA chunks and detect lost chunks. This TSN is encoded as a 32 bits field, as the sequence number by the TCP. However, the TSN is only incremented by one for each data chunk. This implies that the TSN space does not wrap as quickly as the TCP sequence number. When a small message needs to be sent, the SCTP entity creates a new data chunk with the next available TSN and places the data inside the chunk. A single SCTP segment may contain several data chunks, e.g. when small messages are transmitted. Each message is identified by its TSN and within a stream all messages are delivered in sequence. If the message to be transmitted is larger than the underlying network packet, SCTP needs to fragment the message in several chunks that are placed in subsequent segments. The packing of the message in successive segments must still enable the receiver to detect the message boundaries. This is achieved by using the B and E bits of the second high-order byte of the data chunk. The B (Begin) bit is set when the first byte of the User data field of the data chunk is the first byte of the message. The E (End) bit is set when the last byte of the User data field of the data chunk is the last byte of the message. A small message is always sent as chunk whose B and E bits are set to zero. A message which is larger than one network packet will be fragmented in several chunks. Consider for example a message that needs to be divided in three chunks sent in three different SCTP segments. The first chunk will have its B bit set to 1 and its E bit set to 0 and a TSN (say *x*). The second chunk will have both its B and E bits set to 0 and its TSN will be *x+1*. The third, and last, chunk will have its B bit set to 0, its E bit set to 1 and its TSN will be *x+2*. All the chunks that correspond to a given message must have successive TSNs. The B and E bits allow the receiver to recover the message from the received data chunks.

The data chunks are only one part of the reliable data transfert. To reliably transfer data, a transport protocol must also use acknowledgements, retransmissions and flow-control. In SCTP, all these mechanisms rely on the Selective Acknowledgements (Sack) chunk whose format is shown in the figure below.

This chunk is sent by a sender when it needs to send feedback about the reception of data chunks or its buffer space to the remote sender. The *Cumulative TSN ack* contains the TSN of the last data chunk that was received in sequence. This cumulative indicates which TSN has been reliably received by the receiver. The evolution of this field shows the progress of the reliable transmission. This is the first feedback provided by SCTP. Note that in SCTP the acknowledgements are at the chunk level and not at the byte level in contrast with TCP. While SCTP

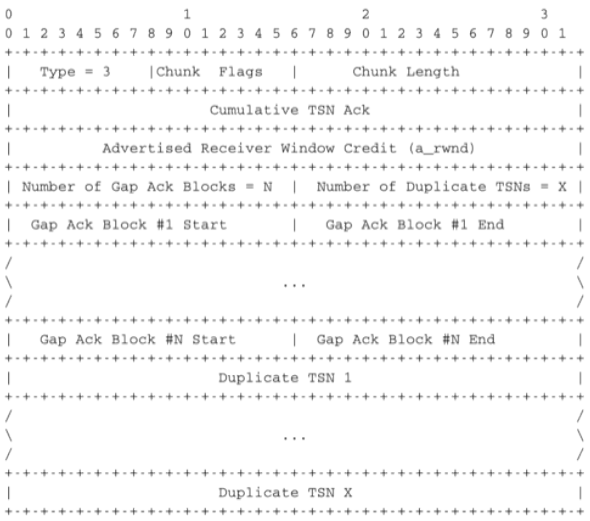


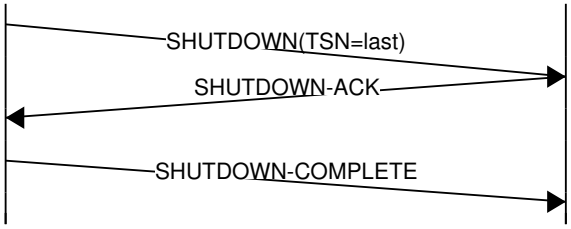
Figure 3.36: The SCTP Sack chunk

transfers messages divided in chunks, buffer space is still measured in bytes and not in variable-length messages or chunks. The *Advertised Receiver Window Credit* field of the Sack chunk provides the current receive window of the receiver. This window is measured in bytes and its left edge is the last byte of the last in-sequence data chunk.

The Sack chunk also provides information about the received out-of-sequence chunks (if any). The Sack chunk contains gap blocks that are in principle similar to the TCP Sack option. However, there are some differences between TCP and SCTP. The Sack option used by TCP has a limited size. This implies that if there are many gaps that need to be reported, a TCP receiver must decide which gaps to include in the SACK option. The SCTP Sack chunk is only limited by the network packet length, which is not a problem in practice. A second difference is that SCTP can also provide feedback about the reception of duplicate chunks. If several copies of the same data chunk have been received, this probably indicates a bad heuristic on the sender. The last part of the Sack chunk provides the list of duplicate TSN received to enable a sender to tune its retransmission mechanism based on this information. Some details on a possible use of this field may be found in **RFC 3708**. The last difference with the TCP SACK option is that the gaps are encoded as deltas relative to the *Cumulative TSN ack*. These deltas are encoded as 16 bits integers and allow to reduce the length of the chunk.

3.9.4 Connection release

SCTP uses a different approach to terminante connections. When an application requests a shutdown of a connection, SCTP performs a three-way handshake. This handshake uses the `SHUTDOWN`, `SHUTDOWN-ACK` and `SHUTDOWN-COMPLETE` chunks. The `SHUTDOWN` chunk is sent once all outgoing data has been acknowledged. It contains the last cumulative sequence number. Upon reception of a `SHUTDOWN` chunk, an SCTP entity informs its application that it cannot accept anymore data over this connection. It then ensures that all outstanding data have been delivered correctly. At that point, it sends a `SHUTDOWN-ACK` to confirm the reception of the `SHUTDOWN` segment. The three-way handshake completes with the transmission of the `SHUTDOWN-COMPLETE` chunk **RFC 4960**.



Note that in contrast with TCP's four-way handshake, the utilisation of a three-way handshake to close an SCTP connection implies that the client (resp. server) may close the connection when the application at the other end

has still some data to transmit. Upon reception of the SHUTDOWN chunk, an SCTP entity must stop accepting new data from the application, but it still needs to retransmit the unacknowledged data chunks (the SHUTDOWN chunk may be placed in the same segment as a Sack chunk that indicates gaps in the received chunks).

SCTP also provides the equivalent to TCP's RST segment. The ABORT chunk can be used to refuse a connection, react to the reception of an invalid segment or immediately close a connection (e.g. due to lack of resources).

3.10 Congestion control

In an internetwork, i.e. a networking composed of different types of networks, such as the Internet, congestion control could be implemented either the network layer or the transport layer. The congestion problem was clearly identified in the later 1980s and the researchers who developed techniques to solve the problem opted for a solution in the transport layer. Adding congestion control to the transport layer makes sense since this layer provides a reliable data transfert and avoiding congestion is a factor in this reliable delivery. The transport layer already deals with heterogeneous networks thanks to its *self-clocking* property that we have already described. In this section, we explain how congestion control has been added to TCP (and SCTP whose congestion control scheme is very close to TCP's congestion control) and how this mechanism could be improved in the future.

The TCP congestion control scheme was initially proposed by Van Jacobson in [Jacobson1988]. The current specification may be found in **RFC 5681**. TCP relies on *Additive Increase and Multiplicative Decrease (AIMD)*. To implement *AIMD*, a TCP host must be able to control its transmission rate. A first approach would be to use timers and adjust their expiration times in function of the rate imposed by *AIMD*. Unfortunately, maintaining such timers for a large number of TCP connections can be difficult. Instead, Van Jacobson noted that the rate of TCP congestion can be artificially controlled by constraining its sending window. A TCP connection cannot send data faster than $\frac{window}{rtt}$ where $window$ is the maximum between the host's sending window and the window advertised by the receiver.

TCP's congestion control scheme is based on a *congestion window*. The current value of the congestion window (*cwnd*) is stored in the TCB of each TCP connection and the window that can be used by the sender is constrained by $min(cwnd, rwin, swin)$ where $swin$ is the current sending window and $rwin$ the last received receive window. The *Additive Increase* part of the TCP congestion control increments the congestion window by *MSS* bytes every round-trip-time. In the TCP literature, this phase is often called the *congestion avoidance* phase. The *Multiplicative Decrease* part of the TCP congestion control divides the current value of the congestion window once congestion has been detected.

When a TCP connection begins, the sending host does not know whether the part of the network that it uses to reach the destination is congested or not. To avoid causing too much congestion, it must start with a small congestion window. [Jacobson1988] recommends an initial window of MSS bytes. As the additive increase part of the TCP congestion control scheme increments the congestion window by MSS bytes every round-trip-time, the TCP connection may have to wait many round-trip-times before being able to efficiently use the available bandwidth. This is especially important in environments where the $bandwidth \times rtt$ product is high. To avoid waiting too many round-trip-times before reaching a congestion window that is large enough to efficiently utilise the network, the TCP congestion control scheme includes the *slow-start* algorithm. The objective of the TCP *slow-start* phase is to quickly reach an acceptable value for the *cwnd*. During *slow-start*, the congestion window is doubled every round-trip-time. The *slow-start* algorithm uses an additional variable in the TCB : *sstresh* (*slow-start threshold*). The *ssthresh* is an estimation of the last value of the *cwnd* that did not cause congestion. It is initialised at the sending window and is updated after each congestion event.

A key question that must be answered by any congestion control scheme is how congestion is detected. The first implementations of the TCP congestion control scheme opted for a simple and pragmatic approach : packet losses indicate congestion. If the network is congested, router buffers are full and packets are discarded. In wired networks, packet losses are mainly caused by congestion. In wireless networks, packets can be lost due to transmission errors and for other reasons that are independent of congestion. TCP already detects segment losses to ensure a reliable delivery. The TCP congestion control scheme distinguishes between two types of congestion :

- *mild congestion*. TCP considers that the network is lightly congested if it receives three duplicate acknowledgements and performs a fast retransmit. If the fast retransmit is successful, this implies that only one segment has been lost. In this case, TCP performs multiplicative decrease and the congestion window is divided by *2*. The slow-start threshold is set to the new value of the congestion window.

- *severe congestion*. TCP considers that the network is severely congested when its retransmission timer expires. In this case, TCP retransmits the first segment, sets the slow-start threshold to 50% of the congestion window. The congestion window is reset to its initial value and TCP performs a slow-start.

The figure below illustrates the evolution of the congestion window when there is severe congestion. At the beginning of the connection, the sender performs *slow-start* until the first segments are lost and the retransmission timer expires. At this time, the *ssthresh* is set to half of the current congestion window and the congestion window is reset at one segment. The lost segments are retransmitted as the sender again performs slow-start until the congestion window reaches the *sshtresh*. It then switches to congestion avoidance and the congestion window increases linearly until segments are lost and the retransmission timer expires ...

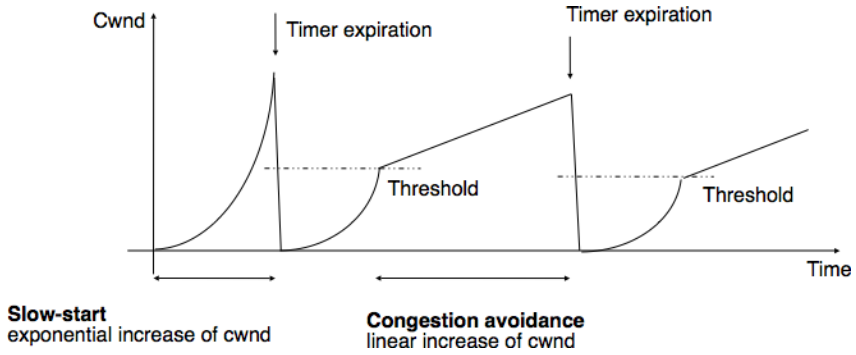


Figure 3.37: Evaluation of the TCP congestion window with severe congestion

The figure below illustrates the evolution of the congestion window when the network is lightly congested and all lost segments can be retransmitted using fast retransmit. The sender begins with a slow-start. A segment is lost but successfully retransmitted by a fast retransmit. The congestion window is divided by 2 and the sender immediately enters congestion avoidance as this was a mild congestion.

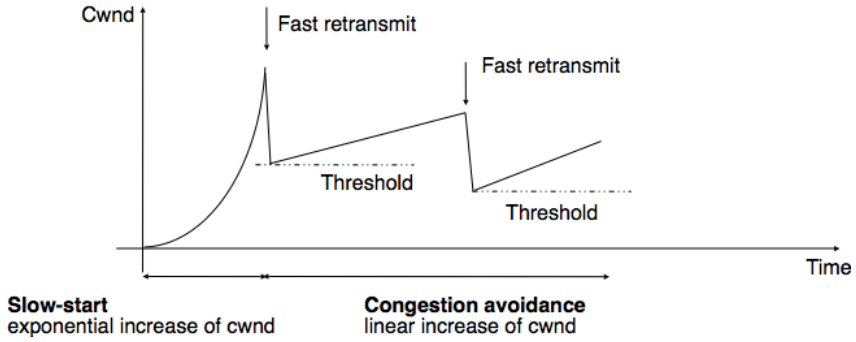


Figure 3.38: Evaluation of the TCP congestion window when the network is lightly congested

Most TCP implementations update the congestion window when they receive an acknowledgement. If we assume that the receiver acknowledges each received segment and the sender only sends MSS sized segments, the TCP congestion control scheme can be implemented using the simplified pseudo-code [27] below.

```
# Initialization
cwnd = MSS  # congestion window in bytes
ssthresh= swin # in bytes

# Ack arrival
if tcp.ack > snd.una :  # new ack, no congestion
  if  cwnd < ssthresh :
    # slow-start : increase quickly cwnd
    # double cwnd  every rtt
```

[27] In this pseudo-code, we assume that TCP uses unlimited sequence and acknowledgement numbers. Furthermore, we do not detail how the *cwnd* is adjusted after the retransmission of the lost segment by fast retransmit. Additional details may be found in RFC 5681.

```
    cwnd = cwnd + MSS
  else:
    # congestion avoidance : increase slowly cwnd
    # increase cwnd by one mss every rtt
    cwnd = cwnd+ mss*(mss/cwnd)
else: # duplicate or old ack
  if tcp.ack==snd.una:    # duplicate acknowledgement
    dupacks++
    if dupacks==3:
      retransmitsegment(snd.una)
      ssthresh=max(cwnd/2,2*MSS)
      cwnd=ssthresh
    else:
      dupacks=0
      # ack for old segment, ignored

Expiration of the retransmission timer:
 send(snd.una)      # retransmit first lost segment
 sshtresh=max(cwnd/2,2*MSS)
 cwnd=MSS
```

Furthermore when a TCP connection has been idle for more than its current retransmission timer, it should reset its congestion window to the congestion window size that it uses when the connection begins, as it no longer knows the current congestion state of the network.

Note: Initial congestion window

The original TCP congestion control mechanism proposed in [Jacobson1988] recommended that each TCP connection should begin by setting $cwnd = MSS$. However, in today's higher bandwidth networks, using such a small initial congestion window severely affects the performance for short TCP connections, such as those used by web servers.In 2002, **RFC 3390** allowed an initial congestion window of about 4 KBytes, which corresponds to 3 segments in many environments. Recently, researchers from google proposed to further increase the initial window up to 15 KBytes [DRC+2010]. The measurements that they collected show that this increase would not significantly increase congestion but would significantly reduce the latency of short HTTP responses. Unsurprisingly, the chosen initial window corresponds to the average size of an HTTP response from a search engine. This proposed modification has been adopted as an experimental modification in **RFC 6928** and popular TCP implementations support it.

3.10.1 Controlling congestion without losing data

In today's Internet, congestion is controlled by regularly sending packets at a higher rate than the network capacity. These packets fill the buffers of the routers and are eventually discarded. But shortly after, TCP senders retransmit packets containing exactly the same data. This is potentially a waste of ressources since these successive retransmissions consume resources upstream of the router that discards the packets. Packet losses are not the only signal to detect congestion inside the network. An alternative is to allow to routers to explicitly indicate their current level of congestion when forwarding packets. This approach was proposed in the late 1980s [RJ1995] and used in some networks. Unfortunately, it took almost a decade before the Internet community agreed to consider this approach. In the mean time, a large number of TCP implementations and routers were deployed on the Internet.

As explained earlier, Explicit Congestion Notification **RFC 3168**, improves the detection of congestion by allowing routers to explicitly mark packets when they are lightly congested. In theory, a single bit in the packet header [RJ1995] is sufficient to support this congestion control scheme. When a host receives a marked packet, it returns the congestion information to the source that adapts its transmission rate accordingly. Although the idea is relatively simple, deploying it on the entire Internet has proven to be challenging [KNT2013]. It is interesting to analyze the different factors that have hindered the deployment of this technique.

The first difficulty in adding Explicit Congestion Notification (ECN) in TCP/IP network was to modify the format of the network packet and transport segment headers to carry the required information. In the network layer, one bit was required to allow the routers to mark the packets they forward during congestion periods. In the IP network layer, this bit is called the *Congestion Experienced* (*CE*) bit and is part of the packet header. However, using a

single bit to mark packets is not sufficient. Consider a simple scenario with two sources, one congested router and one destination. Assume that the first sender and the destination support ECN, but not the second sender. If the router is congested it will mark packets from both senders. The first sender will react to the packet markings by reducing its transmission rate. However since the second sender does not support ECN, it will not react to the markings. Furthermore, this sender could continue to increase its transmission rate, which would lead to more packets being marked and the first source would decrease again its transmission rate, ... In the end, the sources that implement ECN are penalized compared to the sources that do not implement it. This unfairness issue is a major hurdle to widely deploy ECN on the public Internet [28]. The solution proposed in **RFC 3168** to deal with this problem is to use a second bit in the network packet header. This bit, called the *ECN-capable transport* (ECT) bit, indicates whether the packet contains a segment produced by a transport protocol that supports ECN or not. Transport protocols that support ECN set the ECT bit in all packets. When a router is congested, it first verifies whether the ECT bit is set. In this case, the CE bit of the packet is set to indicate congestion. Otherwise, the packet is discarded. This improves the deployability of ECN [29].

The second difficulty is how to allow the receiver to inform the sender of the reception of network packets marked with the *CE* bit. In reliable transport protocols like TCP and SCTP, the acknowledgements can be used to provide this feedback. For TCP, two options were possible : change some bits in the TCP segment header or define a new TCP option to carry this information. The designers of ECN opted for reusing spare bits in the TCP header. More precisely, two TCP flags have been added in the TCP header to support ECN. The *ECN-Echo* (ECN) is set in the acknowledgements when the *CE* was set in packets received on the forward path.

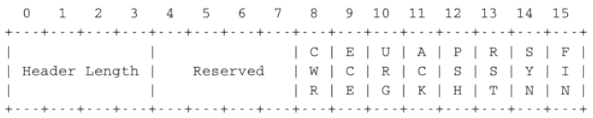


Figure 3.39: The TCP flags

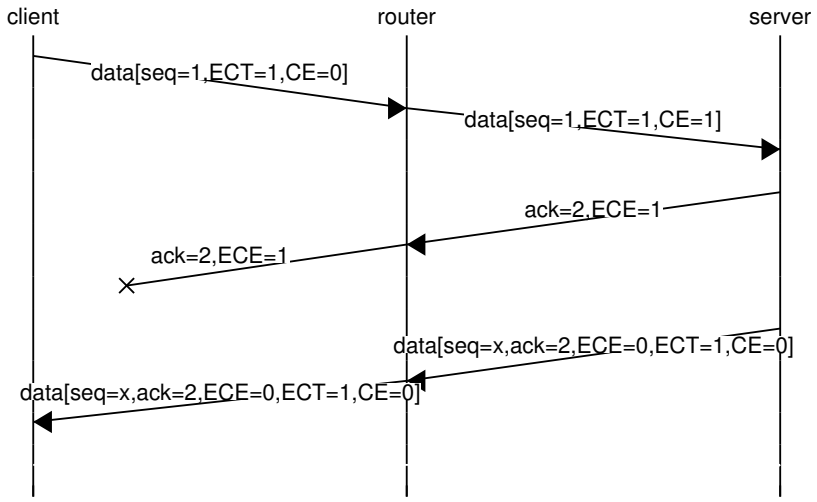
The third difficulty is to allow an ECN-capable sender to detect whether the remote host also supports ECN. This is a classical negotiation of extensions to a transport protocol. In TCP, this could have been solved by defining a new TCP option used during the three-way handshake. To avoid wasting space in the TCP options, the designers of ECN opted in **RFC 3168** for using the *ECN-Echo* and *CWR* bits in the TCP header to perform this negotiation. In the end, the result is the same with fewer bits exchanged. SCTP defines in [STD2013] the *ECN Support parameter* which can be included in the `INIT` and `INIT-ACK` chunks to negotiate the utilization of ECN. The solution adopted for SCTP is cleaner than the solution adopted for TCP.

Thanks to the *ECT*, *CE* and *ECE*, routers can mark packets during congestion and receivers can return the congestion information back to the TCP senders. However, these three bits are not sufficient to allow a server to reliably send the *ECE* bit to a TCP sender. TCP acknowledgements are not sent reliably. A TCP acknowledgement always contains the next expected sequence number. Since TCP acknowledgements are cumulative, the loss of one acknowledgement is recovered by the correct reception of a subsequent acknowledgement.

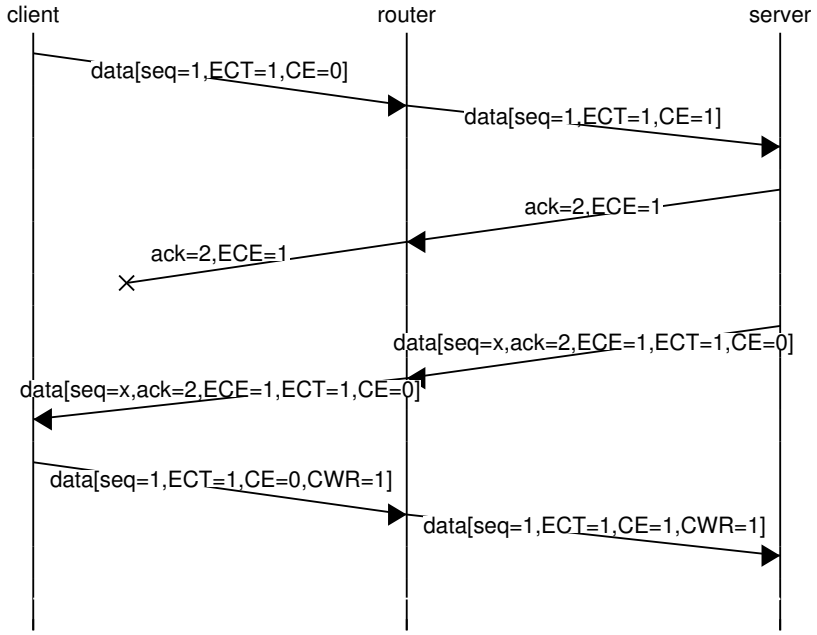
If TCP acknowledgements are overloaded to carry the *ECE* bit, the situation is different. Consider the example shown in the figure below. A client sends packets to a server through a router. In the example below, the first packet is marked. The server returns an acknowledgement with the *ECE* bit set. Unfortunately, this acknowledgement is lost and never reaches the client. Shortly after, the server sends a data segment that also carries a cumulative acknowledgement. This acknowledgement confirms the reception of the data to the client, but it did not receive the congestion information through the *ECE* bit.

[28] In enterprise networks or datacenters, the situation is different since a single company typically controls all the sources and all the routers. In such networks it is possible to ensure that all hosts and routers have been upgraded before turning on ECN on the routers.

[29] With the ECT bit, the deployment issue with ECN is solved provided that all sources cooperate. If some sources do not support ECN but still set the ECT bit in the packets that they sent, they will have an unfair advantage over the sources that correctly react to packet markings. Several solutions have been proposed to deal with this problem **RFC 3540**, but they are outside the scope of this book.



To solve this problem, **RFC 3168** uses an additional bit in the TCP header : the *Congestion Window Reduced* (CWR) bit.



The *CWR* bit of the TCP header provides some form of acknowledgement for the *ECE* bit. When a TCP receiver detects a packet marked with the *CE* bit, it sets the *ECE* bit in all segments that it returns to the sender. Upon reception of an acknowledgement with the *ECE* bit set, the sender reduces its congestion window to reflect a mild congestion and sets the *CWR* bit. This bit remains set as long as the segments received contained the *ECE* bit set. A sender should only react once per round-trip-time to marked packets.

SCTP uses a different approach to inform the sender once congestion has been detected. Instead of using one bit to carry the congestion notification from the receiver to the sender, SCTP defines an entire `ECN Echo` chunk for this. This chunk contains the lowest `TSN` that was received in a packet with the *CE* bit set and the number of marked packets received. The SCTP `CWR` chunk allows to acknowledge the reception of an `ECN Echo` chunk. It echoes the lowest `TSN` placed in the `ECN Echo` chunk.

The last point that needs to be discussed about Explicit Congestion Notification is the algorithm that is used by routers to detect congestion. On a router, congestion manifests itself by the number of packets that are stored inside the router buffers. As explained earlier, we need to distinguish between two types of routers :

- routers that have a single FIFO queue
- routers that have several queues served by a round-robin scheduler

Routers that use a single queue measure their buffer occupancy as the number of bytes of packets stored in the

queue [30]. A first method to detect congestion is to measure the instantaneous buffer occupancy and consider the router to be congested as soon as this occupancy is above a threshold. Typical values of the threshold could be 40% of the total buffer. Measuring the instantaneous buffer occupancy is simple since it only requires one counter. However, this value is fragile from a control viewpoint since it changes frequently. A better solution is to measure the *average* buffer occupancy and consider the router to be congested when this average occupancy is too high. Random Early Detection (RED) [FJ1993] is an algorithm that was designed to support Explicit Congestion Notification. In addition to measuring the average buffer occupancy, it also uses probabilistic marking. When the router is congested, the arriving packets are marked with a probability that increases with the average buffer occupancy. The main advantage of using probabilistic marking instead of marking all arriving packets is that flows will be marked in proportion of the number of packets that they transmit. If the router marks 10% of the arriving packets when congested, then a large flow that sends hundred packets per second will be marked 10 times while a flow that only sends one packet per second will not be marked. This probabilistic marking allows to mark packets in proportion of their usage of the network ressources.

If the router uses several queues served by a scheduler, the situation is different. If a large and a small flow are competing for bandwidth, the scheduler will already favor the small flow that is not using its fair share of the bandwidth. The queue for the small flow will be almost empty while the queue for the large flow will build up. On routers using such schedulers, a good way of marking the packets is to set a threshold on the occupancy of each queue and mark the packets that arrive in a particular queue as soon as its occupancy is above the configured threshold.

3.10.2 Modeling TCP congestion control

Thanks to its congestion control scheme, TCP adapts its transmission rate to the losses that occur in the network. Intuitively, the TCP transmission rate decreases when the percentage of losses increases. Researchers have proposed detailed models that allow the prediction of the throughput of a TCP connection when losses occur [MSMO1997] . To have some intuition about the factors that affect the performance of TCP, let us consider a very simple model. Its assumptions are not completely realistic, but it gives us good intuition without requiring complex mathematics.

This model considers a hypothetical TCP connection that suffers from equally spaced segment losses. If p is the segment loss ratio, then the TCP connection successfully transfers $\frac{1}{p} - 1$ segments and the next segment is lost. If we ignore the slow-start at the beginning of the connection, TCP in this environment is always in congestion avoidance as there are only isolated losses that can be recovered by using fast retransmit. The evolution of the congestion window is thus as shown in the figure below. Note the that *x-axis* of this figure represents time measured in units of one round-trip-time, which is supposed to be constant in the model, and the *y-axis* represents the size of the congestion window measured in MSS-sized segments.

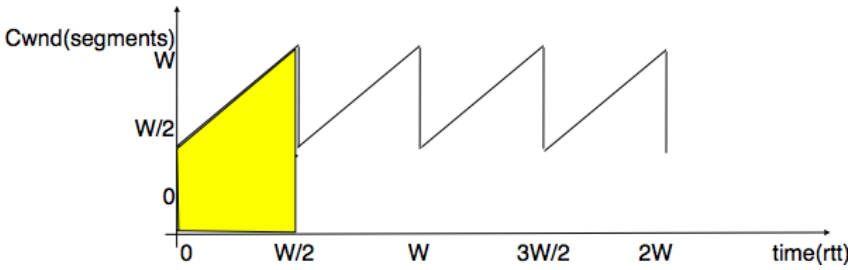


Figure 3.40: Evolution of the congestion window with regular losses

As the losses are equally spaced, the congestion window always starts at some value ($\frac{W}{2}$), and is incremented by one MSS every round-trip-time until it reaches twice this value (W). At this point, a segment is retransmitted and the cycle starts again. If the congestion window is measured in MSS-sized segments, a cycle lasts $\frac{W}{2}$ round-trip-times. The bandwidth of the TCP connection is the number of bytes that have been transmitted during a given period of time. During a cycle, the number of segments that are sent on the TCP connection is equal to the area of the yellow trapeze in the figure. Its area is thus :

[30] The buffers of a router can be implemented as variable or fixed-length slots. If the router uses variable length slots to store the queued packets, then the occupancy is usually measured in bytes. Some routers have use fixed-length slots with each slot large enough to store a maximum-length packet. In this case, the buffer occupancy is measured in packets.

$$area = (\frac{W}{2})^2 + \frac{1}{2} \times (\frac{W}{2})^2 = \frac{3 \times W^2}{8}$$

However, given the regular losses that we consider, the number of segments that are sent between two losses (i.e. during a cycle) is by definition equal to $\frac{1}{p}$. Thus, $W = \sqrt{\frac{8}{3 \times p}} = \frac{k}{\sqrt{p}}$. The throughput (in bytes per second) of the TCP connection is equal to the number of segments transmitted divided by the duration of the cycle :

$$Throughput = \frac{area \times MSS}{time} = \frac{\frac{3 \times W^2}{8}}{\frac{W}{2} \times rtt}$$ or, after having eliminated *W*, $Throughput = \sqrt{\frac{3}{2}} \times \frac{MSS}{rtt \times \sqrt{p}}$

More detailed models and the analysis of simulations have shown that a first order model of the TCP throughput when losses occur was $Throughput \approx \frac{k \times MSS}{rtt \times \sqrt{p}}$. This is an important result which shows that :

- TCP connections with a small round-trip-time can achieve a higher throughput than TCP connections having a longer round-trip-time when losses occur. This implies that the TCP congestion control scheme is not completely fair since it favors the connections that have the shorter round-trip-time
- TCP connections that use a large MSS can achieve a higher throughput that the TCP connections that use a shorter MSS. This creates another source of unfairness between TCP connections. However, it should be noted that today most hosts are using almost the same MSS, roughly 1460 bytes.

In general, the maximum throughput that can be achieved by a TCP connection depends on its maximum window size and the round-trip-time if there are no losses. If there are losses, it depends on the MSS, the round-trip-time and the loss ratio.

$$Throughput < min(\frac{window}{rtt}, \frac{k \times MSS}{rtt \times \sqrt{p}})$$

Note: The TCP congestion control zoo

The first TCP congestion control scheme was proposed by Van Jacobson in [Jacobson1988]. In addition to writing the scientific paper, Van Jacobson also implemented the slow-start and congestion avoidance schemes in release 4.3 *Tahoe* of the BSD Unix distributed by the University of Berkeley. Later, he improved the congestion control by adding the fast retransmit and the fast recovery mechanisms in the *Reno* release of 4.3 BSD Unix. Since then, many researchers have proposed, simulated and implemented modifications to the TCP congestion control scheme. Some of these modifications are still used today, e.g. :

- *NewReno* (**RFC 3782**), which was proposed as an improvement of the fast recovery mechanism in the *Reno* implementation
- *TCP Vegas*, which uses changes in the round-trip-time to estimate congestion in order to avoid it [BOP1994]
- *CUBIC*, which was designed for high bandwidth links and is the default congestion control scheme in the Linux 2.6.19 kernel [HRX2008]
- *Compound TCP*, which was designed for high bandwidth links is the default congestion control scheme in several Microsoft operating systems [STBT2009]

A search of the scientific literature (**RFC 6077**) will probably reveal more than 100 different variants of the TCP congestion control scheme. Most of them have only been evaluated by simulations. However, the TCP implementation in the recent Linux kernels supports several congestion control schemes and new ones can be easily added. We can expect that new TCP congestion control schemes will always continue to appear.

3.11 The network layer

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=8

The main objective of the network layer is to allow endsystems, connected to different networks, to exchange information through intermediate systems called *router*. The unit of information in the network layer is called a *packet*.

Before explaining the network layer in detail, it is useful to begin by analysing the service provided by the *datalink* layer. There are many variants of the datalink layer. Some provide a connection-oriented service while others

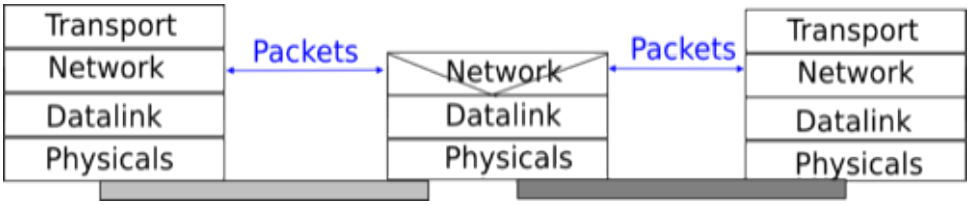


Figure 3.41: The network layer in the reference model

provide a connectionless service. In this section, we focus on connectionless datalink layer services as they are the most widely used. Using a connection-oriented datalink layer causes some problems that are beyond the scope of this chapter. See **RFC 3819** for a discussion on this topic.

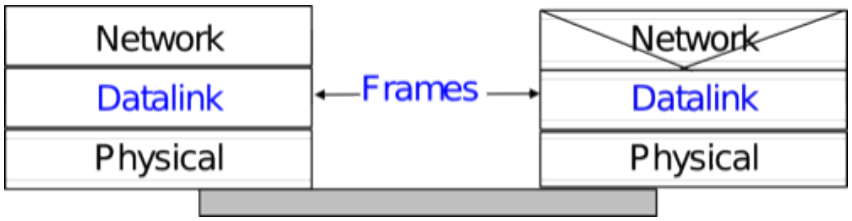


Figure 3.42: The point-to-point datalink layer

There are three main types of datalink layers. The simplest datalink layer is when there are only two communicating systems that are directly connected through the physical layer. Such a datalink layer is used when there is a point-to-point link between the two communicating systems. The two systems can be endsystems or routers. PPP, defined in **RFC 1661**, is an example of such a point-to-point datalink layer. Datalink layers exchange *frames* and a datalink *frame* sent by a datalink layer entity on the left is transmitted through the physical layer, so that it can reach the datalink layer entity on the right. Point-to-point datalink layers can either provide an unreliable service (frames can be corrupted or lost) or a reliable service (in this case, the datalink layer includes retransmission mechanisms similar to the ones used in the transport layer). The unreliable service is frequently used above physical layers (e.g. optical fiber, twisted pairs) having a low bit error ratio while reliability mechanisms are often used in wireless networks to recover locally from transmission errors.

The second type of datalink layer is the one used in Local Area Networks (LAN). Conceptually, a LAN is a set of communicating devices such that any two devices can directly exchange frames through the datalink layer. Both endsystems and routers can be connected to a LAN. Some LANs only connect a few devices, but there are LANs that can connect hundreds or even thousands of devices.

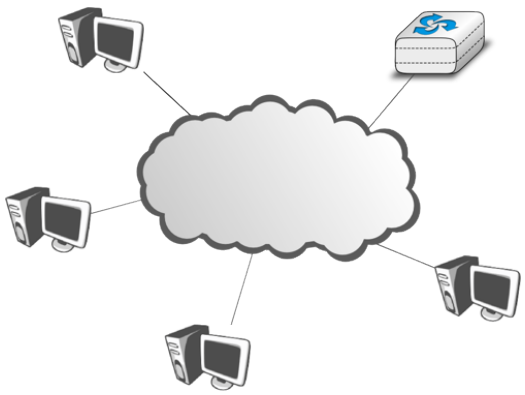


Figure 3.43: A local area network

In the next chapter, we describe the organisation and the operation of Local Area Networks. An important difference between the point-to-point datalink layers and the datalink layers used in LANs is that in a LAN, each communicating device is identified by a unique *datalink layer address*. This address is usually embedded in the

hardware of the device and different types of LANs use different types of datalink layer addresses. Most LANs use 48-bits long addresses that are usually called *MAC* addresses. A communicating device attached to a LAN can send a datalink frame to any other communicating device that is attached to the same LAN. Most LANs also support special broadcast and multicast datalink layer addresses. A frame sent to the broadcast address of the LAN is delivered to all communicating devices that are attached to the LAN. The multicast addresses are used to identify groups of communicating devices. When a frame is sent towards a multicast datalink layer address, it is delivered by the LAN to all communicating devices that belong to the corresponding group.

The third type of datalink layers are used in Non-Broadcast Multi-Access (NBMA) networks. These networks are used to interconnect devices like a LAN. All devices attached to an NBMA network are identified by a unique datalink layer address. However, and this is the main difference between an NBMA network and a traditional LAN, the NBMA service only supports unicast. The datalink layer service provided by an NBMA network supports neither broadcast nor multicast.

Unfortunately no datalink layer is able to send frames of unlimited side. Each datalink layer is characterised by a maximum frame size. There are more than a dozen different datalink layers and unfortunately most of them use a different maximum frame size. The network layer must cope with the heterogeneity of the datalink layer.

3.11.1 IP version 6

In the late 1980s and early 1990s the growth of the Internet was causing several operational problems on routers. Many of these routers had a single CPU and up to 1 MByte of RAM to store their operating system, packet buffers and routing tables. Given the rate of allocation of IPv4 prefixes to companies and universities willing to join the Internet, the routing tables where growing very quickly and some feared that all IPv4 prefixes would quickly be allocated. In 1987, a study cited in **RFC 1752**, estimated that there would be 100,000 networks in the near future. In August 1990, estimates indicated that the class B space would be exhausted by March 1994. Two types of solution were developed to solve this problem. The first short term solution was the introduction of Classless Inter Domain Routing (*CIDR*). A second short term solution was the Network Address Translation (*NAT*) mechanism, defined in **RFC 1631**. NAT allowed multiple hosts to share a single public IP address, it is explained in section *Middleboxes*.

However, in parallel with these short-term solutions, which have allowed the IPv4 Internet to continue to be usable until now, the Internet Engineering Task Force started to work on developing a replacement for IPv4. This work started with an open call for proposals, outlined in **RFC 1550**. Several groups responded to this call with proposals for a next generation Internet Protocol (IPng) :

- TUBA proposed in **RFC 1347** and **RFC 1561**
- PIP proposed in **RFC 1621**
- SIPP proposed in **RFC 1710**

The IETF decided to pursue the development of IPng based on the SIPP proposal. As IP version *5* was already used by the experimental ST-2 protocol defined in **RFC 1819**, the successor of IP version 4 is IP version 6. The initial IP version 6 defined in **RFC 1752** was designed based on the following assumptions :

- IPv6 addresses are encoded as a 128 bits field
- The IPv6 header has a simple format that can easily be parsed by hardware devices
- A host should be able to configure its IPv6 address automatically
- Security must be part of IPv6

Note: The IPng address size

When the work on IPng started, it was clear that 32 bits was too small to encode an IPng address and all proposals used longer addresses. However, there were many discussions about the most suitable address length. A first approach, proposed by SIP in **RFC 1710**, was to use 64 bit addresses. A 64 bits address space was 4 billion times larger than the IPv4 address space and, furthermore, from an implementation perspective, 64 bit CPUs were being considered and 64 bit addresses would naturally fit inside their registers. Another approach was to use an existing address format. This was the TUBA proposal (**RFC 1347**) that reuses the ISO CLNP 20 bytes addresses. The 20 bytes addresses provided room for growth, but using ISO CLNP was not favored by the IETF partially due to

political reasons, despite the fact that mature CLNP implementations were already available. 128 bits appeared to be a reasonable compromise at that time.

IPv6 addressing architecture

The experience of IPv4 revealed that the scalability of a network layer protocol heavily depends on its addressing architecture. The designers of IPv6 spent a lot of effort defining its addressing architecture **RFC 3513**. All IPv6 addresses are 128 bits wide. This implies that there are $340,282,366,920,938,463,463,374,607,431,768,211,456 (3.4 \times 10^{38})$ different IPv6 addresses. As the surface of the Earth is about 510,072,000 km^2, this implies that there are about 6.67×10^{23} IPv6 addresses per square meter on Earth. Compared to IPv4, which offers only 8 addresses per square kilometer, this is a significant improvement on paper.

IPv6 supports unicast, multicast and anycast addresses. An IPv6 unicast address is used to identify one datalink-layer interface on a host. If a host has several datalink layer interfaces (e.g. an Ethernet interface and a WiFi interface), then it needs several IPv6 addresses. In general, an IPv6 unicast address is structured as shown in the figure below.

Note: Textual representation of IPv6 addresses

It is sometimes necessary to write IPv6 addresses in text format, e.g. when manually configuring addresses or for documentation purposes. The preferred format for writing IPv6 addresses is `x:x:x:x:x:x:x:x`, where the `x` 's are hexadecimal digits representing the eight 16-bit parts of the address. Here are a few examples of IPv6 addresses :

- `abcd:Eef01:2345:6789:abcd:ef01:2345:6789`
- `2001:db8:0:0:8:800:200c:417a`
- `fe80:0:0:0:219:e3ff:fed7:1204`

IPv6 addresses often contain a long sequence of bits set to `0`. In this case, a compact notation has been defined. With this notation, *::* is used to indicate one or more groups of 16 bits blocks containing only bits set to *0*. For example,

- `2001:db8:0:0:8:800:200c:417a` is represented as `2001:db8::8:800:200c:417a`
- `ff01:0:0:0:0:0:0:101` is represented as `ff01::101`
- `0:0:0:0:0:0:0:1` is represented as `::1`
- `0:0:0:0:0:0:0:0` is represented as `\:\:`

An IPv6 prefix can be represented as *address/length*, where *length* is the length of the prefix in bits. For example, the three notations below correspond to the same IPv6 prefix :

- `2001:0db8:0000:cd30:0000:0000:0000:0000/60`
- `2001:0db8::cd30:0:0:0:0/60`
- `2001:0db8:0:cd30::/60`

An IPv6 unicast address is composed of three parts :

1. A global routing prefix that is assigned to the Internet Service Provider that owns this block of addresses
2. A subnet identifier that identifies a customer of the ISP
3. An interface identifier that identifies a particular interface on an endsystem

The subnet identifier plays a key role in the scalability of network layer addressing architecture. An important point to be defined in a network layer protocol is the allocation of the network layer addresses. A naive allocation scheme would be to provide an address to each host when the host is attached to the Internet on a first come first served basis. With this solution, a host in Belgium could have address `2001:db8::1` while another host located in Africa would use address `2001:db8::2`. Unfortunately, this would force all routers on the Internet

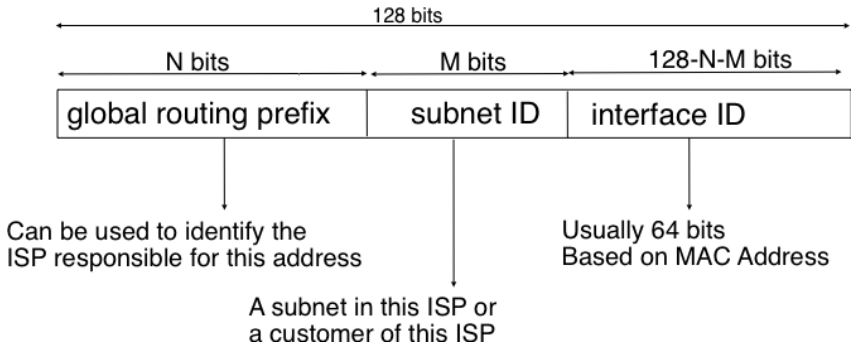


Figure 3.44: Structure of IPv6 unicast addresses

to maintain one route towards each host. In the network layer, scalability is often a function of the number of routes stored on the router. A network will usually work better if its routers store fewer routes and network administrators usually try to minimize the number of routes that are known by their routers. For this, they often divide their network prefix in smaller subblocks. For example, consider a company with three campuses, a large one and two smaller ones. The network administrator would probably divide his block of addresses as follows :

- the bottom half is used for the large campus
- the top half is divided in two smaller blocks, one for each small campus

Inside each campus, the same division can be done, for example on a per building basis, starting from the buildings that host the largest number of nodes, e.g. the company datacenter. In each building, the same division can be done on a per floor basis, ... The advantage of such a hierarchical allocation of the addresses is that the routers in the large campus only need one route to reach a router in the smaller campus. The routers in the large campus would know more routes about the buildings in their campus, but they do not need to know the details of the organisation of each smaller campus.

To preserve the scalability of the routing system, it is important to minimize the number of routes that are stored on each router. A router cannot store and maintain one route for each of the almost 1 billion hosts that are connected to today's Internet. Routers should only maintain routes towards blocks of addresses and not towards individual hosts. For this, hosts are grouped in *subnets* based on their location in the network. A typical subnet groups all the hosts that are part of the same enterprise. An enterprise network is usually composed of several LANs interconnected by routers. A small block of addresses from the Enterprise's block is usually assigned to each LAN.

In today's deployments, interface identifiers are always 64 bits wide. This implies that while there are 2^{128} different IPv6 addresses, they must be grouped in 2^{64} subnets. This could appear as a waste of resources, however using 64 bits for the host identifier allows IPv6 addresses to be auto-configured and also provides some benefits from a security point of view, as explained in section ICMPv6

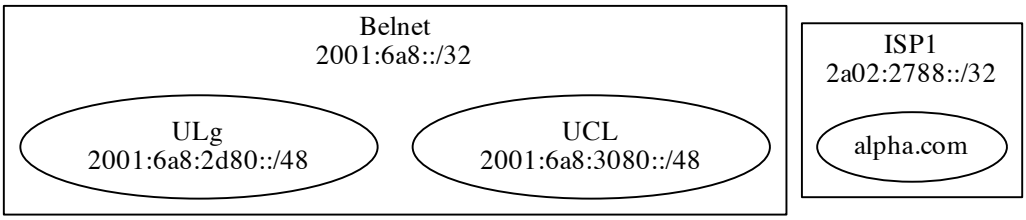
In practice, there are several types of IPv6 unicast address. Most of the IPv6 unicast addresses are allocated in blocks under the responsibility of IANA. The current IPv6 allocations are part of the *2000::/3* address block. Regional Internet Registries (RIR) such as RIPE in Europe, ARIN in North-America or AfriNIC in Africa have each received a block of IPv6 addresses that they sub-allocate to Internet Service Providers in their region. The ISPs then sub-allocate addresses to their customers.

When considering the allocation of IPv6 addresses, two types of address allocations are often distinguished. The RIRs allocate *provider-independent (PI)* addresses. PI addresses are usually allocated to Internet Service Providers and large companies that are connected to at least two different ISPs [CSP2009]. Once a PI address block has been allocated to a company, this company can use its address block with the provider of its choice and change its provider at will. Internet Service Providers allocate *provider-aggregatable (PA)* address blocks from their own PI address block to their customers. A company that is connected to only one ISP should only use PA addresses. The drawback of PA addresses is that when a company using a PA address block changes its provider, it needs to change all the addresses that it uses. This can be a nightmare from an operational perspective and many companies are lobbying to obtain *PI* address blocks even if they are small and connected to a single provider. The typical size of the IPv6 address blocks are :

- `/32` for an Internet Service Provider

- /48 for a single company
- /56 for small user sites
- /64 for a single user (e.g. a home user connected via ADSL)
- /128 in the rare case when it is known that no more than one endhost will be attached

There is one difficulty with the utilisation of these IPv6 prefixes. Consider Belnet, the Belgian research ISP that has been allocated the 2001:6a8::/32 prefix. Universities are connected to Belnet. UCL uses prefix 2001:6a8:3080::/48 while the University of Liege uses 2001:6a8:2d80::/32. A commercial ISP uses prefix 2a02:2788::/32. Both Belnet and the commercial ISP are connected to the global Internet.



The Belnet network advertises prefix 2001:6a8::/32 that includes the prefixes from both UCL and ULg. These two subnetworks can be easily reached from any internet connected host. After a few years, UCL decides to increase the redundancy of its Internet connectivity and buys transit service from ISP1. A direct link between UCL and the commercial ISP appears on the network and UCL expects to receive packets from both Belnet and the commercial ISP.

Now, consider how a router inside alpha.com would reach a host in the UCL network. This router has two routes towards 2001:6a8:3080::1. The first one, for prefix 2001:6a8:3080::/48 is via the direct link between the commercial ISP and UCL. The second one, for prefix 2001:6a8::/32 is via the Internet and Belnet. Since **RFC 1519** when a router knows several routes towards the same destination address, it must forward packets along the route having the longest prefix length. In the case of 2001:6a8:3080::1, this is the route 2001:6a8:3080::/48 that is used to forward the packet. This forwarding rule is called the *longest prefix match* or the *more specific match*. All IP routers implement this forwarding rule.

To understand the *longest prefix match* forwarding, consider the IPv6 routing below.

```
Destination                         Gateway
::/0                                fe80::dead:beef
::1                                 ::1
2a02:2788:2c4:16f::/64              eth0
2001:6a8:3080::/48                  fe80::bad:cafe
2001:6a8:2d80::/48                  fe80::bad:bad
2001:6a8::/32                       fe80::aaaa:bbbb
```

With the longest match rule, the route *::/0* plays a particular role. As this route has a prefix length of *0* bits, it matches all destination addresses. This route is often called the *default* route.

- a packet with destination 2a02:2788:2c4:16f::1 received by router *R* is destined to a host on interface eth0 .
- a packet with destination 2001:6a8:3080::1234 matches three routes : ::/0, 2001::6a8::/32 and 2001::6a8:3080. The packet is forwarded via gateway fe80::bad:cafe
- a packet with destination 2001:1890:123a::1:1e matches one route : ::/0. The packet is forwarded via fe80::dead:beef
- a packet with destination *2001:6a8:3880:40::2'* matches two routes : 2001:6a8::/32 and *::/0*. The packet is forwarded via fe80::aaaa:bbbb

The longest prefix match can be implemented by using different data structures. One possibility is to use a trie. Details on how to implement efficient packet forwarding algorithms may be found in [Varghese2005].

For the companies that want to use IPv6 without being connected to the IPv6 Internet, **RFC 4193** defines the *Unique Local Unicast (ULA)* addresses (`fc00::/7`). These ULA addresses play a similar role as the private IPv4 addresses defined in **RFC 1918**. However, the size of the `fc00::/7` address block allows ULA to be much more flexible than private IPv4 addresses.

Furthermore, the IETF has reserved some IPv6 addresses for a special usage. The two most important ones are :

- `0:0:0:0:0:0:0:1` (`::1` in compact form) is the IPv6 loopback address. This is the address of a logical interface that is always up and running on IPv6 enabled hosts.
- `0:0:0:0:0:0:0:0` (`\:\:` in compact form) is the unspecified IPv6 address. This is the IPv6 address that a host can use as source address when trying to acquire an official address.

The last type of unicast IPv6 addresses are the *Link Local Unicast* addresses. These addresses are part of the *fe80::/10* address block and are defined in **RFC 4291**. Each host can compute its own link local address by concatenating the *fe80::/64* prefix with the 64 bits identifier of its interface. Link local addresses can be used when hosts that are attached to the same link (or local area network) need to exchange packets. They are used notably for address discovery and auto-configuration purposes. Their usage is restricted to each link and a router cannot forward a packet whose source or destination address is a link local address. Link local addresses have also been defined for IPv4 **RFC 3927**. However, the IPv4 link local addresses are only used when a host cannot obtain a regular IPv4 address, e.g. on an isolated LAN.

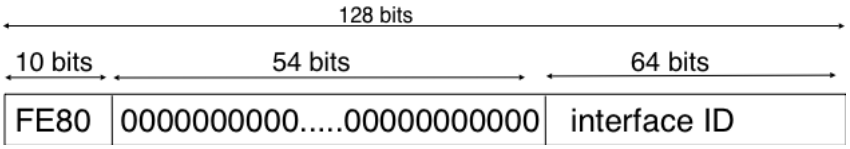
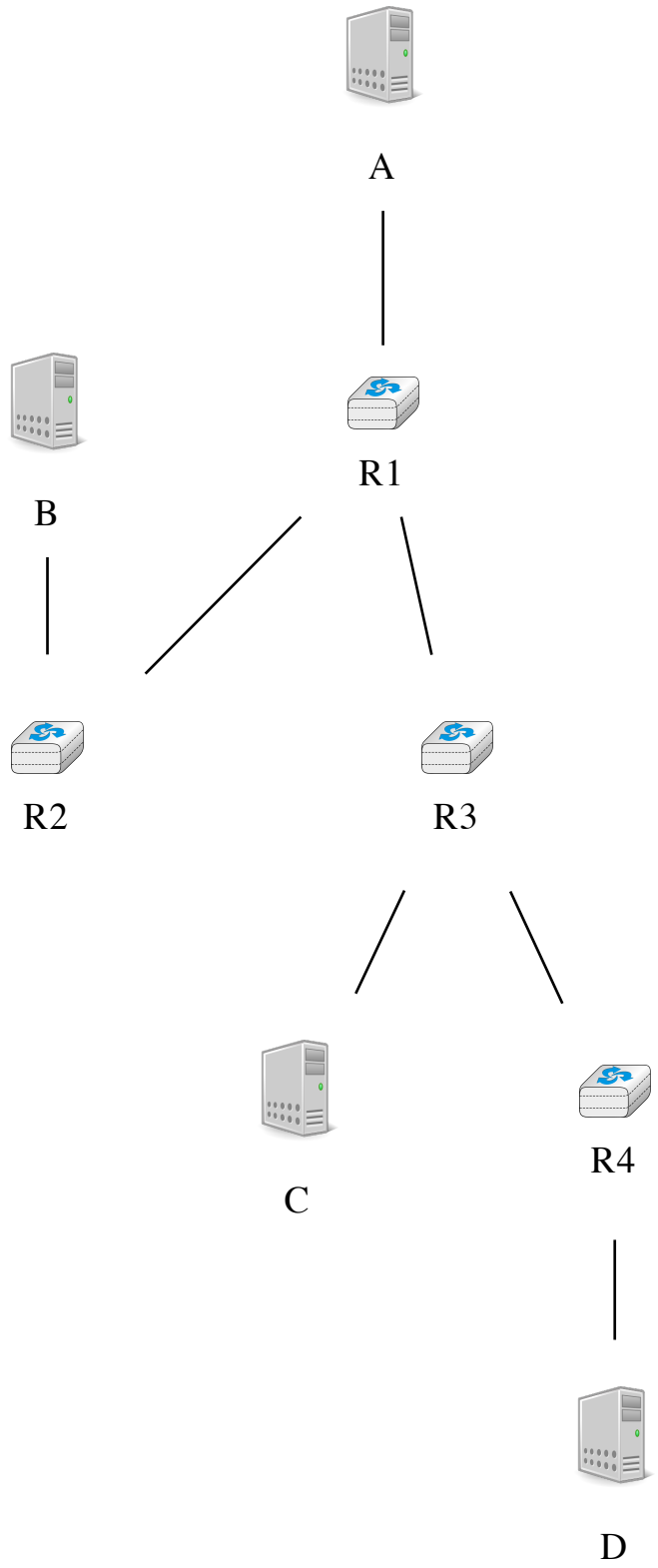


Figure 3.45: IPv6 link local address structure

Note: All IPv6 hosts have several addresses

An important consequence of the IPv6 unicast addressing architecture and the utilisation of link-local addresses is that each IPv6 host has several IPv6 addresses. This implies that all IPv6 stacks must be able to handle multiple IPv6 addresses.

The addresses described above are unicast addresses. These addresses are used to identify (interfaces on) hosts and routers. They can appear as source and destination addresses in the IPv6 packets. When a host sends a packet towards a unicast address, this packet is delivered by the network to its final destination. There are situations, such as when delivering video or television signal to a large number of receivers, where it is useful to have a network that can efficiently deliver the same packet to a large number of receivers. This is the *multicast* service. A multicast service can be provided in a LAN. In this case, a multicast address identifies a set of receivers and each frame sent towards this address is delivered to all receivers in the group. Multicast can also be used in a network containing routers and hosts. In this case, a multicast address identifies also a group of receivers and the network delivers efficiently each multicast packet to all members of the group. Consider for example the network below.


A
R1
B
R2
R3
C
R4
D

Assume that `B` and `D` are part of a multicast group. If `A` sends a multicast packet towards this group, then `R1` will replicate the packet to forward it to `R2` and `R3`. `R2` would forward the packet towards `B`. `R3` would forward the packet towards `R4` that would deliver it to `D`.

Finally, **RFC 4291** defines the structure of the IPv6 multicast addresses [31]. This structure is depicted in the figure below

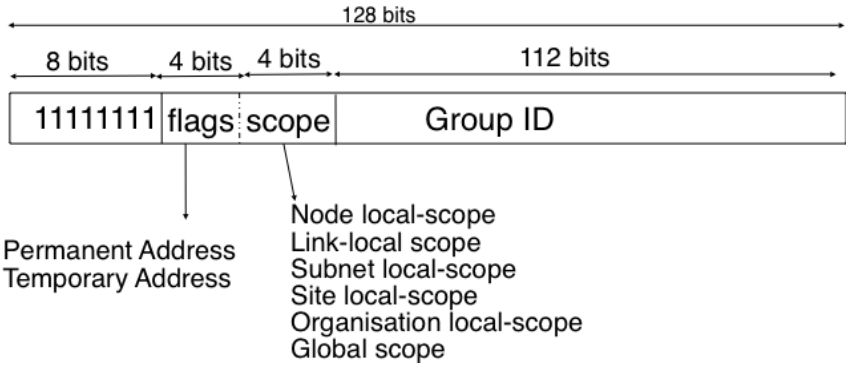


Figure 3.46: IPv6 multicast address structure

The low order 112 bits of an IPv6 multicast address are the group's identifier. The high order bits are used as a marker to distinguish multicast addresses from unicast addresses. Notably, the 4 bits flag field indicates whether the address is temporary or permanent. Finally, the scope field indicates the boundaries of the forwarding of packets destined to a particular address. A link-local scope indicates that a router should not forward a packet destined to such a multicast address. An organisation local-scope indicates that a packet sent to such a multicast destination address should not leave the organisation. Finally the global scope is intended for multicast groups spanning the global Internet.

Among these addresses, some are well known. For example, all endsystem automatically belong to the `ff02::1` multicast group while all routers automatically belong to the `ff02::2` multicast group. A detailed discussion of IPv6 multicast is outside the scope of this chapter.

IPv6 packet format

The IPv6 packet format was heavily inspired by the packet format proposed for the SIPP protocol in **RFC 1710**. The standard IPv6 header defined in **RFC 2460** occupies 40 bytes and contains 8 different fields, as shown in the figure below.

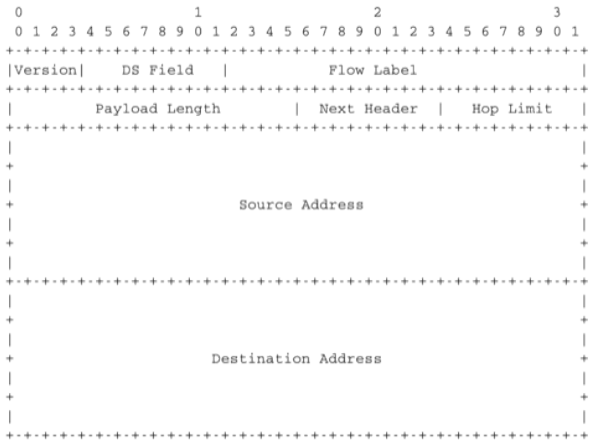


Figure 3.47: The IP version 6 header (**RFC 2460**)

Apart from the source and destination addresses, the IPv6 header contains the following fields :

[31] The full list of allocated IPv6 multicast addresses is available at http://www.iana.org/assignments/ipv6-multicast-addresses

- *version* : a 4 bits field set to *6* and intended to allow IP to evolve in the future if needed
- *Traffic class* : this 8 bits field allows to indicate the type of service expected by this packet and contains the `CE` and `ECT` flags that are used by *Explicit Congestion Notification*
- *Flow label* : this field was initially intended to be used to tag packets belonging to the same *flow*. A recent document, **RFC 6437** describes some possible usages of this field, but it is too early to tell whether it will be really used.
- *Payload length* : this is the size of the packet payload in bytes. As the length is encoded as a 16 bits field, an IPv6 packet can contain up to 65535 bytes of payload.
- *Hop Limit* : this 8 bits field indicates the number of routers that can forward the packet. It is decremented by one by each router and prevents packets from looping forever inside the network.
- *Next Header* : this 8 bits field indicates the type [32] of header that follows the IPv6 header. It can be a transport layer header (e.g. *6* for TCP or *17* for UDP) or an IPv6 option.

It is interesting to note that there is no checksum inside the IPv6 header. This is mainly because all datalink layers and transport protocols include a checksum or a CRC to protect their frames/segments against transmission errors. Adding a checksum in the IPv6 header would have forced each router to recompute the checksum of all packets, with limited benefit in detecting errors. In practice, an IP checksum allows for catching errors that occur inside routers (e.g. due to memory corruption) before the packet reaches its destination. However, this benefit was found to be too small given the reliability of current memories and the cost of computing the checksum on each router [33].

When a host receives an IPv6 packet, it needs to determine which transport protocol (UDP, TCP, SCTP, ...) needs to handle the payload of the packet. This is the first role of the *Next header* field. The IANA which manages the allocation of Internet ressources and protocol parameters, maintains an official list of transport protocols [2]. The following protocol numbers are reserved :

- `TCP` uses *Next Header* number `6`
- `UDP` uses *Next Header* number `17`
- `SCTP` uses *Next Header* number `132`

For example, an IPv6 packet that contains an SCTP segment would appear as shown in the figure below. However,

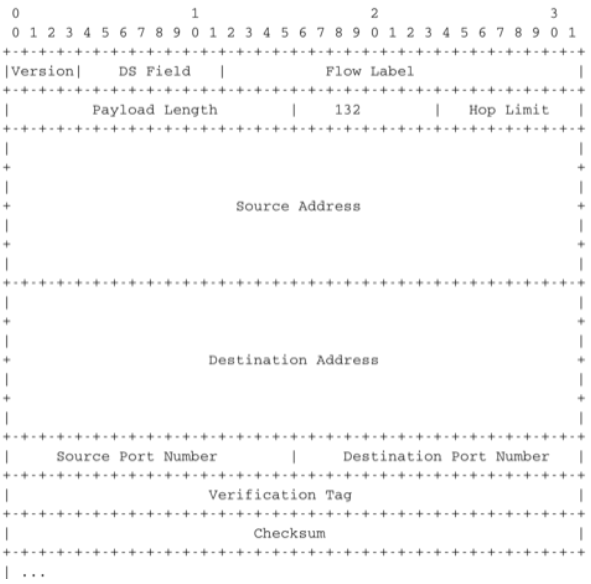


Figure 3.48: An IPv6 packet containing an SCTP segment

[32] The IANA maintains the list of all allocated Next Header types at http://www.iana.org/assignments/protocol-numbers/

[33] When IPv4 was designed, the situation was different. The IPv4 header includes a checksum that only covers the network header. This checksum is computed by the source and updated by all intermediate routers that decrement the TTL, which is the IPv4 equivalent of the *HopLimit* used by IPv6.

the *Next header* has broader usages than simply indicating the transport protocol which is responsible for the packet payload. An IPv6 packet can contain a chain of headers and the last one indicates the transport protocol that is responsible for the packet payload. Supporting a chain of headers is a clever design from an extensibility viewpoint. As we will seen, this chain of headers has several usages.

RFC 2460 defines several types of IPv6 extension headers that could be added to an IPv6 packet :

- *Hop-by-Hop Options* header. This option is processed by routers and endhosts.
- *Destination Options* header. This option is processed only by endhosts.
- *Routing* header. This option is processed by some nodes.
- *Fragment* header. This option is processed only by endhosts.
- *Authentication* header. This option is processed only by endhosts.
- *Encapsulating Security Payload*. This option is processed only by endhosts.

The last two headers are used to add security above IPv6 and implement IPSec. They are described in **RFC 2402** and **RFC 2406** and are outside the scope of this document.

The *Hop-by-Hop Options* header was designed to allow IPv6 to be easily extended. In theory, this option could be used to define new fields that were not foreseen when IPv6 was designed. It is intended to be processed by both routers and endhosts. Deploying an extension to a network protocol can be difficult in practice since some nodes already support the extensions while others still use the old version and do not understand the extension. To deal with this issue, the IPv6 designers opted for a Type-Length-Value encoding of these IPv6 options. The *Hop-by-Hop Options* header is encoded as shown below.

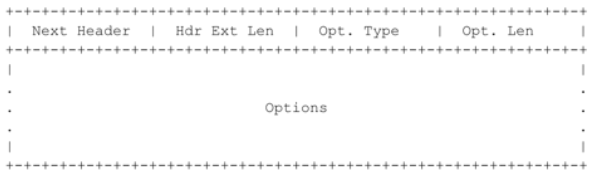


Figure 3.49: The IPv6 *Hop-by-Hop Options* header

In this optional header, the *Next Header* field is used to support the chain of headers. It indicates the type of the next header in the chain. IPv6 headers have different lengths. The *Hdr Ext Len* field indicates the total length of the option header in bytes. The *Opt. Type* field indicates the type of option. These types are encoded such that their high order bits specify how the header needs to be handled by nodes that do not recognize it. The following values are defined for the two high order bits :

- `00` : if a node does not recognize this header, it can be safely skipped and the processing continues with the subsequent header
- `01` : if a node does not recognize this header, the packet must be discarded
- `10` (resp. `11`) : if a node does not recognize this header, it must return a control packet (ICMP, see later) back to the source (resp. except if the destination was a multicast address)

This encoding allows the designers of protocol extensions to specify whether the option must be supported by all nodes on a path or not. Still, deploying such an extension can be difficult in practice.

Two *hop-by-hop* options have been defined. **RFC 2675** specifies the jumbogram that enables IPv6 to support packets containing a payload larger than 65535 bytes. These jumbo packets have their *payload length* set to *0* and the jumbogram option contains the packet length as a 32 bits field. Such packets can only be sent from a source to a destination if all the routers on the path support this option. However, as of this writing it does not seem that the jumbogram option has been implemented. The router alert option defined in **RFC 2711** is the second example of a *hop-by-hop* option. The packets that contain this option should be processed in a special way by intermediate routers. This option is used for IP packets that carry Resource Reservation Protocol (RSVP) messages, but this is outside the scope of this book.

The *Destinations Option* header uses the same format as the *Hop-by-Hop Options* header. It has some usages, e.g. to support mobile nodes **RFC 6275**, but these are outside the scope of this document.

The *Fragment Options* header is more important. An important problem in the network layer is the ability to handle heterogeneous datalink layers. Most datalink layer technologies can only transmit and receive frames that are shorter than a given maximum frame size. Unfortunately, all datalink layer technologies use different maximum frames sizes.

Each datalink layer has its own characteristics and as indicated earlier, each datalink layer is characterised by a maximum frame size. From IP's point of view, a datalink layer interface is characterised by its *Maximum Transmission Unit (MTU)*. The MTU of an interface is the largest packet (including header) that it can send. The table below provides some common MTU sizes **[#f6lowpan]_**.

Datalink layer	MTU
Ethernet	1500 bytes
WiFi	2272 bytes
ATM (AAL5)	9180 bytes
802.15.4	102 or 81 bytes
Token Ring	4464 bytes
FDDI	4352 bytes

Although IPv6 can send 64 KBytes long packets, few datalink layer technologies that are used today are able to send a 64 KBytes packet inside a frame. Furthermore, as illustrated in the figure below, another problem is that a host may send a packet that would be too large for one of the datalink layers used by the intermediate routers.

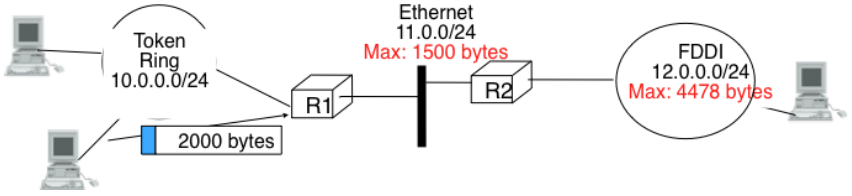


Figure 3.50: The need for fragmentation and reassembly

To solve these problems, IPv6 includes a packet fragmentation and reassembly mechanism. In IPv4, fragmentation was performed by both the endhosts and the intermediate routers. However, experience with IPv4 has shown that fragmenting packets in routers was costly [KM1995]. For this reason, the developers of IPv6 have decided that routers would not fragment packets anymore. In IPv6, fragmentation is only performed by the source host. If a source has to send a packet which is larger than the MTU of the outgoing interface, the packet needs to be fragmented before being transmitted. In IPv6, each packet fragment is an IPv6 packet that includes the *Fragmentation* header. This header is included by the source in each packet fragment. The receiver uses them to reassemble the received fragments.

Figure 3.51: IPv6 fragmentation header

If a router receives a packet that is too long to be forwarded, the packet is dropped and the router returns an ICMPv6 message to inform the sender of the problem. The sender can then either fragment the packet or perform Path MTU discovery. In IPv6, packet fragmentation is performed only by the source by using IPv6 options.

In IPv6, fragmentation is performed exclusively by the source host and relies on the fragmentation header. This 64 bits header is composed of six fields :

- a *Next Header* field that indicates the type of the header that follows the fragmentation header
- a *reserved* field set to *0*.
- the *Fragment Offset* is a 13-bit unsigned integer that contains the offset, in 8 bytes units, of the data following this header, relative to the start of the original packet.
- the *More* flag, which is set to *0* in the last fragment of a packet and to *1* in all other fragments.

- the 32 bits *Identification* field indicates to which original packet a fragment belongs. When a host sends fragmented packets, it should ensure that it does not reuse the same *identification* field for packets sent to the same destination during a period of *MSL* seconds. This is easier with the 32 bits *identification* used in the IPv6 fragmentation header, than with the 16 bits *identification* field of the IPv4 header.

Some IPv6 implementations send the fragments of a packet in increasing fragment offset order, starting from the first fragment. Others send the fragments in reverse order, starting from the last fragment. The latter solution can be advantageous for the host that needs to reassemble the fragments, as it can easily allocate the buffer required to reassemble all fragments of the packet upon reception of the last fragment. When a host receives the first fragment of an IPv6 packet, it cannot know a priori the length of the entire IPv6 packet.

The figure below provides an example of a fragmented IPv6 packet containing a UDP segment. The *Next Header* type reserved for the IPv6 fragmentation option is 44.

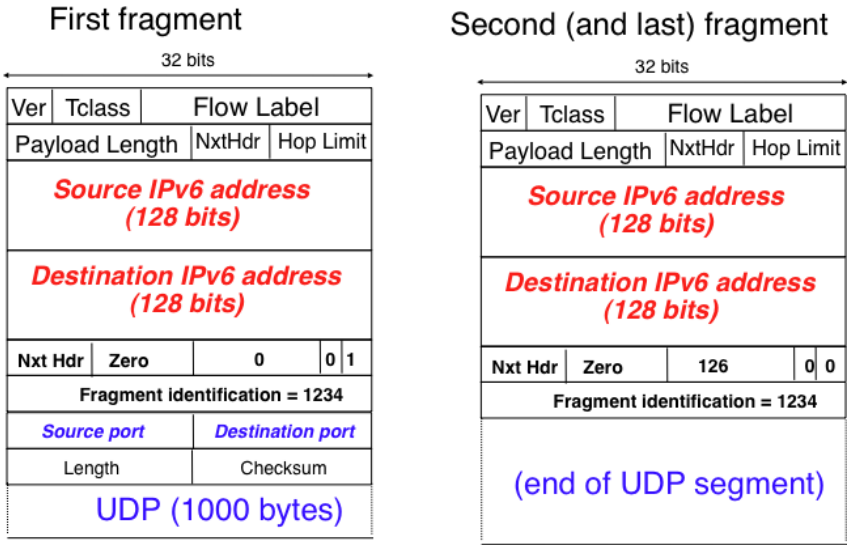


Figure 3.52: IPv6 fragmentation example

The following pseudo-code details the IPv6 fragmentation, assuming that the packet does not contain options.

```
#mtu : maximum size of the packet (including header) of outgoing link
if p.len <  mtu :
   send(p)
# packet is too large
maxpayload=8*int((mtu-40)/8)   # must be n times 8 bytes
# packet must be fragmented
payload=p[IP].payload
pos=0
id=globalCounter;
globalCounter++;
while len(payload) > 0 :
   if len(payload) > maxpayload :
      toSend=IP(dest=p.dest,src=p.src,
                hoplimit=p.hoplimit, id,
                frag=p.frag+(pos/8), m=false,
                len=mtu, nextheader=p.nextheader)/payload[0:maxpayload]
      pos=pos+maxpayload
      payload=payload[maxpayload+1:]
   else
      toSend=IP(dest=p.dest,src=p.src,
                hoplimit=p.hoplimit, id,
                frag=p.frag+(pos/8), m=true,
                len=len(payload), nextheader=p.nextheader)/payload
   forward(toSend)
```

In the above pseudocode, we maintain a single 32 bits counter that is incremented for each packet that needs to be fragmented. Other implementations to compute the packet identification are possible. **RFC 2460** only requires that two fragmented packets that are sent within the MSL between the same pair of hosts have different identifications.

The fragments of an IPv6 packet may arrive at the destination in any order, as each fragment is forwarded independently in the network and may follow different paths. Furthermore, some fragments may be lost and never reach the destination.

The reassembly algorithm used by the destination host is roughly as follows. First, the destination can verify whether a received IPv6 packet is a fragment or not by checking whether it contains a fragment header. If so, all fragments with the some identification must be reassembled together. The reassembly algorithm relies on the *Identification* field of the received fragments to associate a fragment with the corresponding packet being reassembled. Furthermore, the *Fragment Offset* field indicates the position of the fragment payload in the original unfragmented packet. Finally, the packet with the *M* flag reset allows the destination to determine the total length of the original unfragmented packet.

Note that the reassembly algorithm must deal with the unreliability of the IP network. This implies that a fragment may be duplicated or a fragment may never reach the destination. The destination can easily detect fragment duplication thanks to the *Fragment Offset*. To deal with fragment losses, the reassembly algorithm must bound the time during which the fragments of a packet are stored in its buffer while the packet is being reassembled. This can be implemented by starting a timer when the first fragment of a packet is received. If the packet has not been reassembled upon expiration of the timer, all fragments are discarded and the packet is considered to be lost.

Note: Header compression on low bandwidth links

Given the size of the IPv6 header, it can cause huge overhead on low bandwidth links, especially when small packets are exchanged such as for Voice over IP applications. In such environments, several techniques can be used to reduce the overhead. A first solution is to use data compression in the datalink layer to compress all the information exchanged [Thomborson1992]. These techniques are similar to the data compression algorithms used in tools such as `compress(1)` or `gzip(1)` **RFC 1951**. They compress streams of bits without taking advantage of the fact that these streams contain IP packets with a known structure. A second solution is to compress the IP and TCP header. These header compression techniques, such as the one defined in **RFC 5795** take advantage of the redundancy found in successive packets from the same flow to significantly reduce the size of the protocol headers. Another solution is to define a compressed encoding of the IPv6 header that matches the capabilities of the underlying datalink layer **RFC 4944**.

The last type of *IPv6 header extension* is the *Routing'header. The ''type 0'* routing header defined in **RFC 2460** is an example of an IPv6 option that must be processed by some routers. This option is encoded as shown below.

The type 0 routing option was intended to allow a host to indicate a loose source route that should be followed by a packet by specifying the addresses of some of the routers that must forward this packet. Unfortunately, further work with this routing header, including an entertaining demonstration with scapy [BE2007] , revealed severe security problems with this routing header. For this reason, loose source routing with the type 0 routing header has been removed from the IPv6 specification **RFC 5095**.

3.11.2 ICMP version 6

It is sometimes necessary for intermediate routers or the destination host to inform the sender of the packet of a problem that occurred while processing a packet. In the TCP/IP protocol suite, this reporting is done by the Internet Control Message Protocol (ICMP). ICMPv6 is defined in **RFC 4443**. It is used both to report problems that occurred while processing an IPv6 packet, but also when distributing addresses.

ICMPv6 messages are carried inside IPv6 packets (the *Next Header* field for ICMPv6 is `58`). Each ICMP message contains an 8 bits header with a *type* field, a *code* field and a 16 bits checksum computed over the entire ICMPv6 message. The message body contains a copy of the IPv6 packet in error.

ICMPv6 specifies two classes of messages : error messages that indicate a problem in handling a packet and informational messages. Four types of error messages are defined in **RFC 4443** :

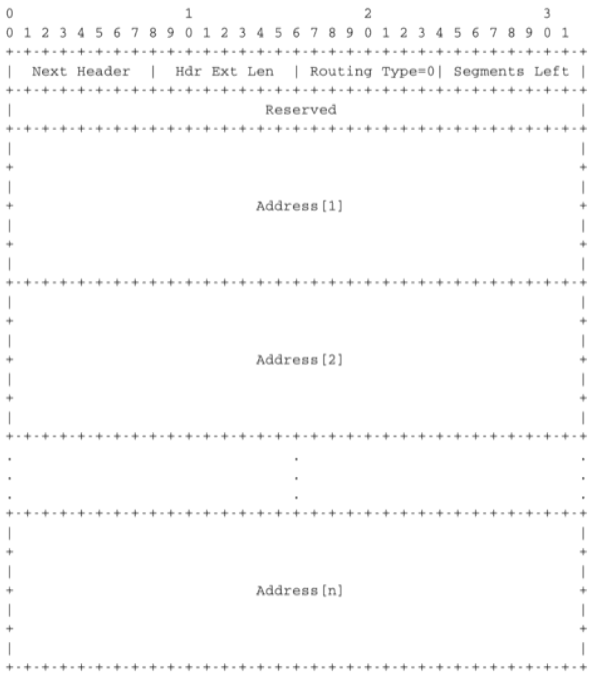


Figure 3.53: The Type 0 routing header (**RFC 2460**)

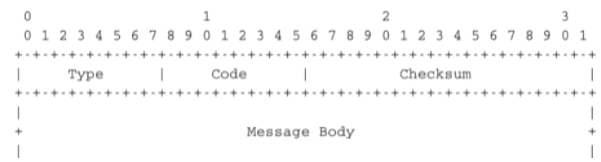
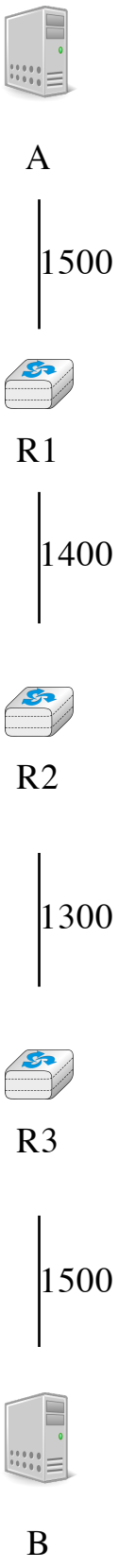


Figure 3.54: ICMP version 6 packet format

- **1** [Destination Unreachable. Such an ICMPv6 message is sent when the destination address of a packet is unreachable. The *code* field of the ICMP header contains additional information about the type of unreachability. The following codes are specified in **RFC 4443**]
  - `0` : No route to destination. This indicates that the router that sent the ICMPv6 message did not have a route towards the packet's destination
  - `1` : Communication with destination administratively prohibited. This indicates that a firewall has refused to forward the packet towards its final destination.
  - `2` : Beyond scope of source address. This message can be sent if the source is using link-local addresses to reach a global unicast address outside its subnet.
  - `3` : Address unreachable. This message indicates that the packet reached the subnet of the destination, but the host that owns this destination address cannot be reached.
  - `4` : Port unreachable. This message indicates that the IPv6 packet was received by the destination, but there was no application listening to the specified port.
- `2` : Packet Too Big. The router that was to send the ICMPv6 message received an IPv6 packet that is larger than the MTU of the outgoing link. The ICMPv6 message contains the MTU of this link in bytes. This allows the sending host to implement Path MTU discovery **RFC 1981**
- `3` : Time Exceeded. This error message can be sent either by a router or by a host. A router would set *code* to *0* to report the reception of a packet whose *Hop Limit* reached *0*. A host would set *code* to *1* to report that it was unable to reassemble received IPv6 fragments.
- `4` : Parameter Problem. This ICMPv6 message is used to report either the reception of an IPv6 packet with an erroneous header field (type *0*) or an unknown *Next Header* or IP option (types *1* and *2*). In this case, the message body contains the erroneous IPv6 packet and the first 32 bits of the message body contain a pointer to the error.

The *Destination Unreachable* ICMP error message is returned when a packet cannot be forwarded to its final destination. The first four ICMPv6 error messages (type `1`, codes `0-3`) are generated by routers while endhosts may return code `4` when there is no application bound to the corresponding port number.

The *Packet Too Big* ICMP messages enable the source endhost to discover the MTU size that it can safely use to reach a given destination. To understand its operation, consider the (academic) scenario shown in the figure below. In this figure, the labels on each link represent the maximum packet size supported by this link.


A
1500
R1
1400
R2
1300
R3
1500
B

If A sends a 1500 bytes packet, R1 will return an ICMPv6 error message indicating a maximum packet length of 1400 bytes. A would then fragment the packet before retransmitting it. The small fragment would go through, but the large fragment will be refused by R2 that would return an ICMPv6 error message. A can refragment the packet and send it to the final destination as two fragments.

In practice, an IPv6 implementation does not store the transmitted packets to be able to retransmit them if needed. However, since TCP (and SCTP) buffer the segments that they transmit, a similar approach can be used in transport protocols to detect the maximum MTU on a path towards a given destination. This technique is called PathMTU Discovery **RFC 1981**.

When a TCP segment is transported in an IP packet that is fragmented in the network, the loss of a single fragment forces TCP to retransmit the entire segment (and thus all the fragments). If TCP was able to send only packets that do not require fragmentation in the network, it could retransmit only the information that was lost in the network. In addition, IP reassembly causes several challenges at high speed as discussed in **RFC 4963**. Using IP fragmentation to allow UDP applications to exchange large messages raises several security issues [KPS2003].

ICMPv6 is used by TCP implementations to discover the largest MTU size that is allowed to reach a destination host without causing network fragmentation. A TCP implementation parses the *Packets Too Big* ICMP messages that it receives. These ICMP messages contain the MTU of the router's outgoing link in their *Data* field. Upon reception of such an ICMP message, the source TCP implementation adjusts its Maximum Segment Size (MSS) so that the packets containing the segments that it sends can be forwarded by this router without requiring fragmentation.

Two types of informational ICMPv6 messages are defined in **RFC 4443** : *echo request* and *echo reply*, which are used to test the reachability of a destination by using *ping6(8)*. Each host is supposed [34] to reply with an ICMP *Echo reply* message when its receives an ICMP *Echo request* message. A sample usage of *ping6(8)* is shown below.

```
#ping6 www.ietf.org
PING6(56=40+8+8 bytes) 2001:6a8:3080:2:3403:bbf4:edae:afc3 --> 2001:1890:123a::1:1e
16 bytes from 2001:1890:123a::1:1e, icmp_seq=0 hlim=49 time=156.905 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=1 hlim=49 time=155.618 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=2 hlim=49 time=155.808 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=3 hlim=49 time=155.325 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=4 hlim=49 time=155.493 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=5 hlim=49 time=155.801 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=6 hlim=49 time=155.660 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=7 hlim=49 time=155.869 ms
^C
--- www.ietf.org ping6 statistics ---
8 packets transmitted, 8 packets received, 0.0% packet loss
round-trip min/avg/max/std-dev = 155.325/155.810/156.905/0.447 ms
```

Another very useful debugging tool is *traceroute6(8)*. The traceroute man page describes this tool as *"print the route packets take to network host"*. traceroute uses the *Time exceeded* ICMP messages to discover the intermediate routers on the path towards a destination. The principle behind traceroute is very simple. When a router receives an IP packet whose *Hop Limit* is set to 1 it is forced to return to the sending host a *Time exceeded* ICMP message containing the header and the first bytes of the discarded packet. To discover all routers on a network path, a simple solution is to first send a packet whose *Hop Limit* is set to *1*, then a packet whose *Hop Limit* is set to *2*, etc. A sample traceroute6 output is shown below.

```
#traceroute6 www.ietf.org
traceroute6 to www.ietf.org (2001:1890:1112:1::20) from 2001:6a8:3080:2:217:f2ff:fed6:65c0, 30 hop
 1  2001:6a8:3080:2::1  13.821 ms  0.301 ms  0.324 ms
 2  2001:6a8:3000:8000::1  0.651 ms  0.51 ms  0.495 ms
 3  10ge.cr2.bruvil.belnet.net  3.402 ms  3.34 ms  3.33 ms
 4  10ge.cr2.brueve.belnet.net  3.668 ms 10ge.cr2.brueve.belnet.net  3.988 ms 10ge.cr2.brueve.beln
 5  belnet.rt1.ams.nl.geant2.net  10.598 ms  7.214 ms  10.082 ms
 6  so-7-0-0.rt2.cop.dk.geant2.net  20.19 ms  20.002 ms  20.064 ms
 7  kbn-ipv6-b1.ipv6.telia.net  21.078 ms  20.868 ms  20.864 ms
```

[34] Until a few years ago, all hosts replied to *Echo request* ICMP messages. However, due to the security problems that have affected TCP/IP implementations, many of these implementations can now be configured to disable answering *Echo request* ICMP messages.

```
8  s-ipv6-b1-link.ipv6.telia.net  31.312 ms  31.113 ms  31.411 ms
9  s-ipv6-b1-link.ipv6.telia.net  61.986 ms  61.988 ms  61.994 ms
10  2001:1890:61:8909::1  121.716 ms  121.779 ms  121.177 ms
11  2001:1890:61:9117::2  203.709 ms  203.305 ms  203.07 ms
12  mail.ietf.org  204.172 ms  203.755 ms  203.748 ms
```

Note: Rate limitation of ICMP messages

High-end hardware based routers use special purpose chips on their interfaces to forward IPv6 packets at line rate. These chips are optimised to process *correct* IP packets. They are not able to create ICMP messages at line rate. When such a chip receives an IP packet that triggers an ICMP message, it interrupts the main CPU of the router and the software running on this CPU processes the packet. This CPU is much slower than the hardware acceleration found on the interfaces [Gill2004]. It would be overloaded if it had to process IP packets at line rate and generate one ICMP message for each received packet. To protect this CPU, high-end routers limit the rate at which the hardware can interrupt the main CPU and thus the rate at which ICMP messages can be generated. This implies that not all erroneous IP packets cause the transmission of an ICMP message. The risk of overloading the main CPU of the router is also the reason why using hop-by-hop IPv6 options, including the router alter option is discouraged [35].

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=9

3.12 The IPv6 subnet

Until now, we have focussed our discussion on the utilisation of IPv6 on point-to-point links. Although there are point-to-point links in the Internet, mainly between routers and sometimes for endhosts, most of the endhosts are attached to datalink layer networks such as Ethernet LANs or WiFi networks. These datalink layer networks play an important role in today's Internet and have heavily influenced the design of the operation of IPv6. To understand IPv6 and ICMPv6 completely, we first need to correctly understand the key principles behind these datalink layer technologies.

As explained earlier, devices attached to a Local Area Network can directly exchange frames among themselves. For this, each datalink layer interface on a device (endhost, router, ...) attached to such a network is identified by a MAC address. Each datalink layer interface includes a unique hardwired MAC address. MAC addresses are allocated to manufacturers in blocks and interface is numbered with a unique address. Thanks to the global unicity of the MAC addresses, the datalink layer service can assume that two hosts attached to a LAN have different addresses. Most LANs provide an unreliable connectionless service and a datalink layer frame has a header containing :

- the source MAC address
- the destination MAC address
- some multiplexing information to indicate the network layer protocol that is responsible for the payload of the frame

LANs also provide a broadcast and a multicast service. The broadcast service enables a device to send a single frame to all the devices attached to the same LAN. This is done by reserving a special broadcast MAC address (typically all bits of the address are set to one). To broadcast a frame, a device simply needs to send a frame whose destination is the broadcast address. All devices attached to the datalink network will receive the frame.

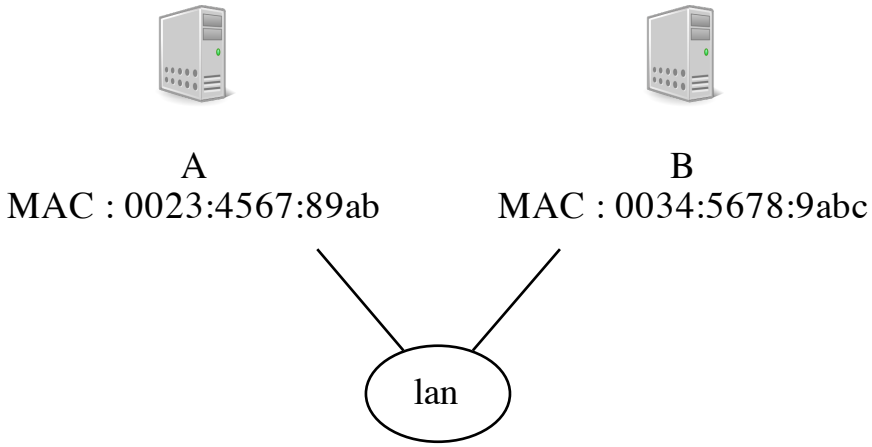
The broadcast service allows to easily reach all devices attached to a datalink layer network. It has been widely used to support IP version 4. A drawback of using the broadcast service to support a network layer protocol is that a broadcast frame that contains a network layer packet is always delivered to all devices attached to the datalink network, even if some of these devices do not support the network layer protocol. The multicast service is a useful alternative to the broadcast service. To understand its operation, it is important to understand how a datalink layer

[35] For a discussion of the issues with the router alert IP option, see http://tools.ietf.org/html/draft-rahman-rtg-router-alert-dangerous-00 or http://tools.ietf.org/html/draft-rahman-rtg-router-alert-considerations-03

interface operates. In shared media LANs, all devices are attached to the same physical medium and all frames are delivered to all devices. When such a frame is received by a datalink layer interface, it compares the destination address with the MAC address of the device. If the two addresses match, or the destination address is the broadcast address, the frame is destined to the device and its payload is delivered to the network layer protocol. The multicast service exploits this principle. A multicast address is a logical address. To receive frames destined to a multicast address in a shared media LAN, a device captures all frames having this multicast address as their destination. All IPv6 nodes are capable of capturing datalink layer frames destined to different multicast addresses.

3.12.1 Interactions between IPv6 and the datalink layer

IPv6 hosts and routers frequently interact with the datalink layer service. To understand the main interactions, it is useful to analyze all the packets that are exchanged when a simple network containing a few hosts and routers is built. Let us first start with a LAN containing two hosts [36].



Hosts `A` and `B` are attached to the same datalink layer network. They can thus exchange frames by using the MAC addresses shown in the figure above. To be able to use IPv6 to exchange packets, they need to have an IPv6 address. One possibility would be to manually configure an IPv6 address on each host. However, IPv6 provides a better solution thanks to the *link-local* IPv6 addresses. A *link-local* IPv6 address is an address that is composed by concatenating the `fe80::/64` prefix with the MAC address of the device. In the example above, host A would use IPv6 *link-local* address `fe80::0223:45FF:FE67:89ab` and host B `fe80::0234:5678:9aFF:FEbc:dede`. With these two IPv6 addresses, the hosts can exchange IPv6 packets.

Note: Converting MAC addresses in host identifiers

Appendix A of **RFC 4291** provides the algorithm used to convert a 48 bits MAC address into a 64 bits host identifier. This algorithm builds upon the structure of the MAC addresses. A MAC address is represented as shown in the figure below.

MAC addresses are allocated in blocks of 2^{20}. When a company registers for a block of MAC addresses, it receives an identifier. company identifier is then used to populated the c bits of the MAC addresses. The company can allocate all addresses in starting with this prefix and mangages the m bits as it wishes.

[36] For simplicity, you assume that each datalink layer interface is assigned a 64 bits MAC address. As we will see later, today's datalink layer technologies mainly use 48 bits MAC addresses, but the smaller addresses can easily be converted into 64 bits addresses.

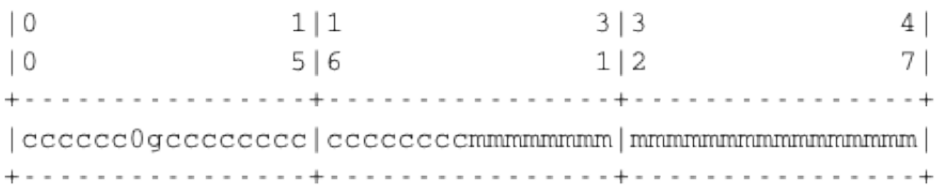


Figure 3.55: A MAC address

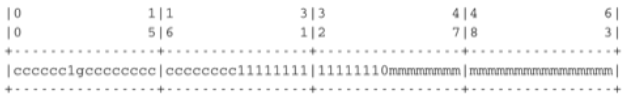
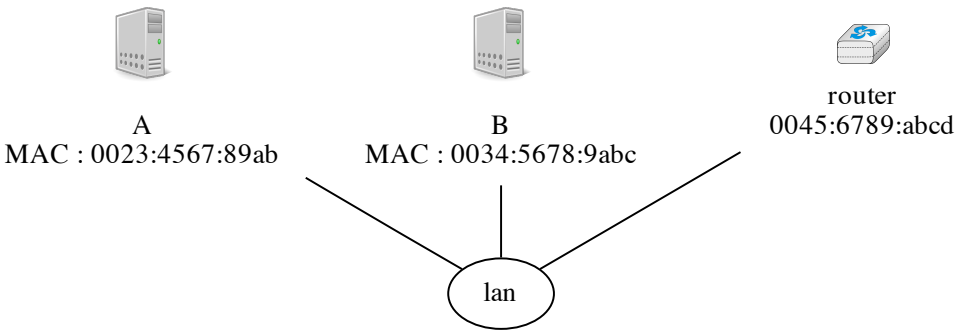


Figure 3.56: A MAC address converted into a 64 bits host identifier

Inside a MAC address, the two bits indicated as *0* and *g* in the figure above play a special role. The first bit indicates whether the address is universal or local. The *g* bit indicates whether this is a multicast address or a unicast address. The MAC address can be converted into a 64 bits host identifier by flipping the value of the *0* bit and inserting `FFFE`, i.e. `1111111111111110` in binary, in the middle of the address as shown in the figure below. The *c*, *m* and *g* bits of the MAC address are not modified.

The next step is to connect the LAN to the Internet. For this, a router is attached to the LAN.



Assume that the LAN containing the two hosts and the router is assigned prefix `2001:db8:1234:5678/64`. A first solution to configure the IPv6 addresses in this network is to assign them manually. A possible assignment is :

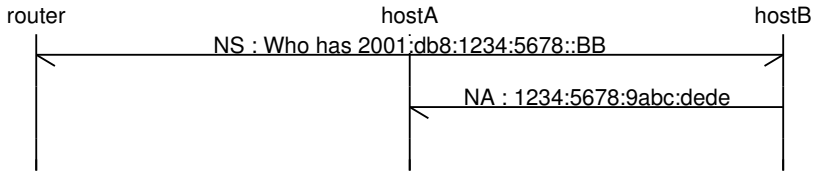
- `2001:db8:1234:5678::1` is assigned to `router`
- `2001:db8:1234:5678::AA` is assigned to `hostA`
- `2001:db8:1234:5678::BB` is assigned to `hostB`

To be able to exchange IPv6 packets with `hostB`, `hostA` needs to know the MAC address of the interface of `hostB` on the LAN. This is the *address resolution* problem. In IPv6, this problem is solved by using the Neighbor Discovery Protocol (NDP). NDP is specified in **RFC 4861**. This protocol is part of ICMPv6 and uses the multicast datalink layer service.

NDP allows a host to discover the MAC address used by any other host attached to the same LAN. NDP operates in two steps. First, the querier sends a multicast ICMPv6 Neighbor Solicitation message that contains as parameter the queried IPv6 address. This multicast ICMPv6 NS is placed inside a multicast frame [37]. The queried node

[37] **RFC 4291** and **RFC 4861** explain in more details how the IPv6 multicast address is determined from the target IPv6 unicast address. These details are outside the scope of this book, but may matter if you try to understand a packet trace.

receives the frame, parses it and replies with a unicast ICMPv6 Neighbor Advertisement that provides its own IPv6 and MAC addresses. Upon reception of the Neighbor Advertisement message, the querier stores the mapping between the IPv6 and the MAC address inside its NDP table. This table is a data structure that maintains a cache of the recently received Neighbor Advertisement. Thanks to this cache, a host only needs to send a Neighbor Sollicitation message for the first packet that it sends to a given host. After this initial packet, the NDP table can provide the mapping between the destination IPv6 address and the corresponding MAC address.



The NS message can also be used to verify the reachability of a host in the local subnet. For this usage, NS messages can be sent in unicast since other nodes on the subnet do not need to process the message.

When an entry in the NDP table times out on a host, it may either be deleted or the host may try to revalidate it by sending the NS message again.

This is not the only usage of the Neighbor Solicitation and Neighbor Advertisement messages. They are also used to detect the utilization of duplicate addresses. In the network above, consider what happens when a new host is connected to the LAN. If this host is configured by mistake with the same address as `hostA` (i.e. `2001:db8:1234:5678::AA`), problems could occur. Indeed, if two hosts have the same IPv6 address on the LAN, but different MAC addresses, it will be difficult to correctly reach them. IPv6 anticipated this problem and includes a *Duplicate Address Detection* Algorithm (DAD). When an IPv6 address [38] is configured on a host, by any means, the host must verify the uniqueness of this address on the LAN. For this, it multicasts an ICMPv6 Neighbor Solicitation that queries the network for its newly configured address. The IPv6 source address of this NS is set to `::` (i.e. the reserved unassigned address) if the host does not already have an IPv6 address on this subnet). If the NS does not receive any answer, the new address is considered to be unique and can safely be used. Otherwise, the new address is refused and an error message should be returned to the system administrator or a new IPv6 address should be generated. The *Duplicate Address Detection* Algorithm can prevent various operational problems that are often difficult to debug.

Few users manually configure the IPv6 addresses on their hosts. They prefer to rely on protocols that can automatically configure their IPv6 addresses. IPv6 supports two such protocols : DHCPv6 and the Stateless Address Autoconfiguration (SLAAC).

The Stateless Address Autoconfiguration (SLAAC) mechanism defined in **RFC 4862** enables hosts to automatically configure their addresses without maintaining any state. When a host boots, it derives its identifier from its datalink layer address [39] as explained earlier and concatenates this 64 bits identifier to the *FE80::/64* prefix to obtain its link-local IPv6 address. It then multicasts a Neighbour Solicitation with its link-local address as a target to verify whether another host is using the same link-local address on this subnet. If it receives a Neighbour Advertisement indicating that the link-local address is used by another host, it generates another 64 bits identifier and sends again a Neighbour Solicitation. If there is no answer, the host considers its link-local address to be valid. This address will be used as the source address for all NDP messages sent on the subnet.

To automatically configure its global IPv6 address, the host must know the globally routable IPv6 prefix that is used on the local subnet. IPv6 routers regularly multicast ICMPv6 Router Advertisement messages that indicate the IPv6 prefix assigned to the subnet. The Router Advertisement message contains several interesting fields.

This message is sent from the link-local address of the router on the subnet. Its destination is the IPv6 multicast address that targets all IPv6 enabled hosts (i.e. `ff02::1`). The *Cur Hop Limit* field, if different from zero, allows to specify the default *Hop Limit* that hosts should use when sending IPv6 from this subnet. `64` is a frequently used value. The *M* and *O* bits are used to indicate that some information can be obtained from DHCPv6. The *Router Lifetime* parameter provides the expected lifetime (in seconds) of the sending router acting as a default router. This lifetime allows to plan the replacement of a router by another one in the same subnet. The *Reachable Time*

[38] The DAD algorithm is also used with *link-local* addresses.

[39] Using a datalink layer address to derive a 64 bits identifier for each host raises privacy concerns as the host will always use the same identifier. Attackers could use this to track hosts on the Internet. An extension to the Stateless Address Configuration mechanism that does not raise privacy concerns is defined in **RFC 4941**. These privacy extensions allow a host to generate its 64 bits identifier randomly every time it attaches to a subnet. It then becomes impossible for an attacker to use the 64-bits identifier to track a host.

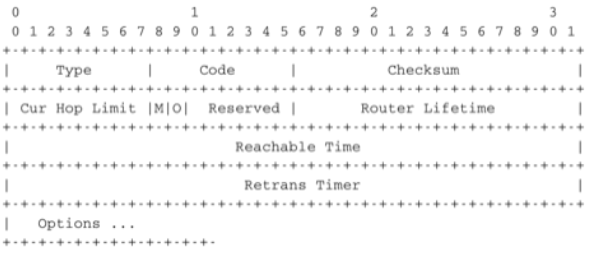


Figure 3.57: Format of the ICMPv6 Router Advertisement message

and the *Retrans Timer* parameter are used to configure the utilisation of the NDP protocol on the hosts attached to the subnet.

Several options can be included in the Router Advertisement message. The simplest one is the MTU option that indicates the MTU to be used within the subnet. Thanks to this option, it is possible to ensure that all devices attached to the same subnet use the same MTU. Otherwise, operational problems could occur. The *Prefix* option is more important. It provides information about the prefix(es) that is (are) advertised by the router on the subnet.

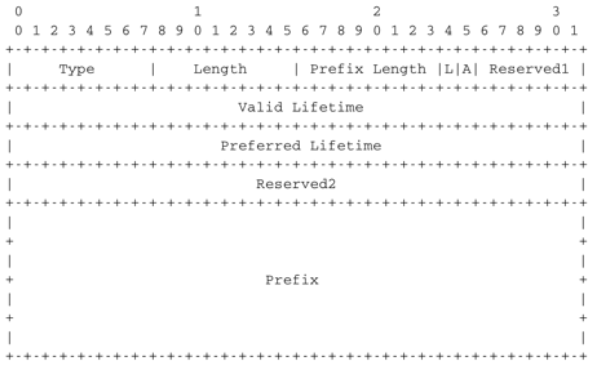


Figure 3.58: The Prefix information option

The key information placed in this option are the prefix and its length. This allows the hosts attached to the subnet to automatically configure their own IPv6 address. The *Valid* and *Preferred Lifetimes* provide information about the expected lifetime of the prefixes. Associating some time validity to prefixes is a good practice from an operational viewpoint. There are some situations where the prefix assigned to a subnet needs to change without impacting the hosts attached to the subnet. This is often called the IPv6 renumbering problem in the literature **RFC 7010**. A very simple scenario is the following. An SME subscribes to one ISP. Its router is attached to another router of this ISP and advertises a prefix assigned by the ISP. The SME is composed of a single subnet and all its hosts rely on stateless address configuration. After a few years, the SME decides to change of network provider. It connects its router to the second ISP and receives a different prefix from this ISP. At this point, two prefixes are advertised on the SME's subnet. The old prefix can be advertised with a short lifetime to ensure that hosts will stop using it while the new one is advertised with a longer lifetime. After sometime, the router stops advertising the old prefix and the hosts stop using it. The old prefix can now be returned back to the first ISP. In larger networks, renumbering an IPv6 remains a difficult operational problem [LeB2009].

Upon reception of this message, the host can derive its global IPv6 address by concatenating its 64 bits identifier with the received prefix. It concludes the SLAAC by sending a Neighbour Solicitation message targeted at its global IPv6 address to ensure that no other host is not using the same IPv6 address.

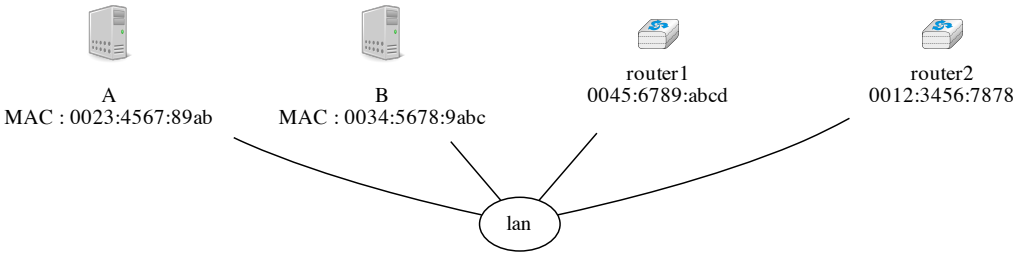
Note: Router Advertisements and Hop Limits

ICMPv6 Router Advertisements messages are regularly sent by routers. They are destined to all devices attached to the local subnet and no router should ever forward them to another subnet. Still, these messages are sent inside IPv6 packets whose *Hop Limit* is always set to `255`. Given that the packet should not the forwarded outside of the local subnet, the reader could expect instead a *Hop Limit* set to `1`. Using a *Hop Limit* set to `255` provides one important benefit from a security viewpoint and this hack has been adapted in several Internet protocols. When a host receives a *Router Advertisement* message, it expects that this message has been generated by a router attached

to the same subnet. Using a *Hop Limit* of `255` provides a simple check for this. If the message was generated by an attacker outside the subnet, it would reach the subnet with a decremented *Hop Limit*. Checking that the *Hop Limit* is set to `255` is a simple [40] verification that the packet was generated on this particular subnet. **RFC 5082** provides other examples of protocols that use this hack and discuss its limitations.

Routers regularly send Router Advertisement messages. These messages are triggered by a timer that is often set at approximately 30 seconds. Usually, hosts wait for the arrival of a Router Advertisement message to configure their address. This implies that hosts could sometimes need to wait 30 seconds before being able to configure their address. If this delay is too long, a host can also send a *Router Solicitation* message. This message is sent towards the multicast address that corresponds to all IPv6 routers (i.e. `FF01::2`) and the default router will reply.

The last point that needs to be explained about ICMPv6 is the *Redirect* message. This message is used when there is more than one router on a subnet as shown in the figure below.



In this network, `router1` is the default router for all hosts. The second router, `router2` provides connectivity to a specific IPv6 subnet, e.g. `2001:db8:abcd::/48`. These two routers attached to the same subnet can be used in different ways. First, it is possible to manually configure the routing tables on all hosts to add a route towards `2001:db8:abcd::/48` via `router2`. Unfortunately, forcing such manual configuration boils down all the benefits of using address auto-configuration in IPv6. The second approach is to automatically configure a default route via `router1` on all hosts. With such route, when a host needs to send a packet to any address within `2001:db8:abcd::/48`, it will send it to `router1`. `router1` would consult its routing table and find that the packet needs to be sent again on the subnet to reach `router2`. This is a waste of time. A better approach would be to enable the hosts to automatically learn the new route. This is possible thanks to the ICMPv6 *Redirect* message. When `router1` receives a packet that needs to be forwarded back on the same interface, it replies with a *Redirect* message that indicates that the packet should have been sent via `router2`. Upon reception of a *Redirect* message, the host updates it forwarding table to include a new transient entry for the destination reported in the message. A timeout is usually associated with this transient entry to automatically delete it after some time.

An alternative is the Dynamic Host Configuration Protocol (DHCP) defined in **RFC 2131** and **RFC 3315**. DHCP allows a host to automatically retrieve its assigned IPv6 address, but relies on server. A DHCP server is associated to each subnet [41]. Each DHCP server manages a pool of IPv6 addresses assigned to the subnet. When a host is first attached to the subnet, it sends a DHCP request message in a UDP segment (the DHCP server listens on port 67). As the host knows neither its IPv6 address nor the IPv6 address of the DHCP server, this UDP segment is sent inside a multicast packet target at the DHCP servers. The DHCP request may contain various options such as the name of the host, its datalink layer address, etc. The server captures the DHCP request and selects an unassigned address in its address pool. It then sends the assigned IPv6 address in a DHCP reply message which contains the datalink layer address of the host and additional information such as the subnet mask, the address of the default router or the address of the DNS resolver. The DHCP reply also specifies the lifetime of the address allocation. This forces the host to renew its address allocation once it expires. Thanks to the limited lease time, IP addresses are automatically returned to the pool of addresses when hosts are powered off.

Both SLAAC and DHCPv6 can be extended to provide additional information beyond the IPv6 prefix/address. For

[40] Using a *Hop Limit* of `255` prevents one family of attacks against ICMPv6, but other attacks still remain possible. A detailed discussion of the security issues with IPv6 is outside the scope of this book. It is possible to secure NDP by using the *Cryptographically Generated IPv6 Addresses* (CGA) defined in **RFC 3972**. The Secure Neighbour Discovery Protocol is defined in **RFC 3971**. A detailed discussion of the security of IPv6 may be found in [HV2008].

[41] In practice, there is usually one DHCP server per group of subnets and the routers capture on each subnet the DHCP messages and forward them to the DHCP server.

example, **RFC 6106** defines options for the ICMPv6 ND message that can carry the IPv6 address of the recursive DNS resolver and a list of default domain search suffixes. It is also possible to combine SLAAC with DHCPv6. **RFC 3736** defines a stateless variant of DHCPv6 that can be used to distribute DNS information while SLAAC is used to distribute the prefixes.

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues/new

3.13 Routing in IP networks

In a large IP network such as the global Internet, routers need to exchange routing information. The Internet is an interconnection of networks, often called domains, that are under different responsibilities. As of this writing, the Internet is composed on more than 40,000 different domains and this number is still growing [42]. A domain can be a small enterprise that manages a few routers in a single building, a larger enterprise with a hundred routers at multiple locations, or a large Internet Service Provider managing thousands of routers. Two classes of routing protocols are used to allow these domains to efficiently exchange routing information.

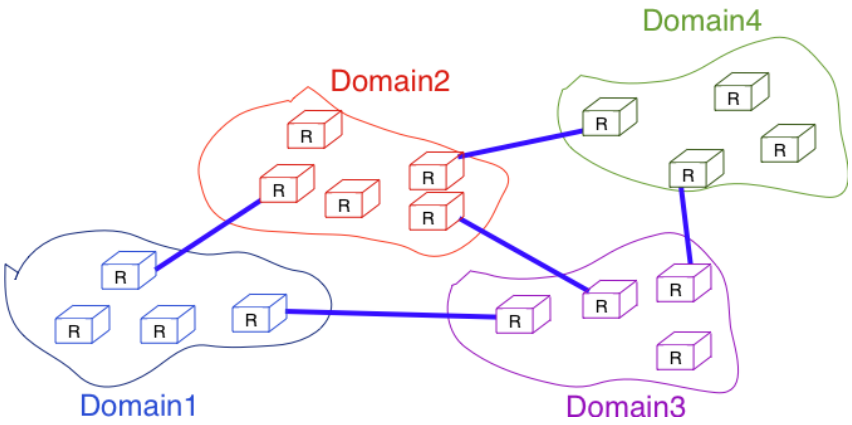


Figure 3.59: Organisation of a small Internet

The first class of routing protocols are the *intradomain routing protocols* (sometimes also called the interior gateway protocols or *IGP*). An intradomain routing protocol is used by all routers inside a domain to exchange routing information about the destinations that are reachable inside the domain. There are several intradomain routing protocols. Some domains use *RIP*, which is a distance vector protocol. Other domains use link-state routing protocols such as *OSPF* or *IS-IS*. Finally, some domains use static routing or proprietary protocols such as *IGRP* or *EIGRP*.

These intradomain routing protocols usually have two objectives. First, they distribute routing information that corresponds to the shortest path between two routers in the domain. Second, they should allow the routers to quickly recover from link and router failures.

The second class of routing protocols are the *interdomain routing protocols* (sometimes also called the exterior gateway protocols or *EGP*). The objective of an interdomain routing protocol is to distribute routing information between domains. For scalability reasons, an interdomain routing protocol must distribute aggregated routing information and considers each domain as a black box.

A very important difference between intradomain and interdomain routing are the *routing policies* that are used by each domain. Inside a single domain, all routers are considered equal, and when several routes are available to reach a given destination prefix, the best route is selected based on technical criteria such as the route with the shortest delay, the route with the minimum number of hops or the route with the highest bandwidth.

When we consider the interconnection of domains that are managed by different organisations, this is no longer true. Each domain implements its own routing policy. A routing policy is composed of three elements : an *import*

[42] See http://bgp.potaroo.net/index-as.html for reports on the evolution of the number of Autonomous Systems over time.

filter that specifies which routes can be accepted by a domain, an *export filter* that specifies which routes can be advertised by a domain and a ranking algorithm that selects the best route when a domain knows several routes towards the same destination prefix. As we will see later, another important difference is that the objective of the interdomain routing protocol is to find the *cheapest* route towards each destination. There is only one interdomain routing protocol : *BGP*.

3.14 Intradomain routing

In this section, we briefly describe the key features of the two main intradomain unicast routing protocols : RIP and OSPF. The basic principles of distance vector and link-state routing have been presented earlier.

3.14.1 RIP

The Routing Information Protocol (RIP) is the simplest routing protocol that was standardised for the TCP/IP protocol suite. RIP is defined in **RFC 2453**. Additional information about RIP may be found in [Malkin1999]

RIP routers periodically exchange RIP messages. The format of these messages is shown below. A RIP message is sent inside a UDP segment whose destination port is set to *521*. A RIP message contains several fields. The *Cmd* field indicates whether the RIP message is a request or a response. When a router boots, its routing table is empty and it cannot forward any packet. To speedup the discovery of the network, it can send a request message to the RIP IPv6 multicast address, `FF02::9`. All RIP routers listen to this multicast address and any router attached to the subnet will reply by sending its own routing table as a sequence of RIP messages. In steady state, routers multicast one of more RIP response messages every 30 seconds. These messages contain the distance vectors that summarize the router's routing table. The current version of RIP is version 2 defined in **RFC 2453** for IPv4 and **RFC 2080** for IPv6.

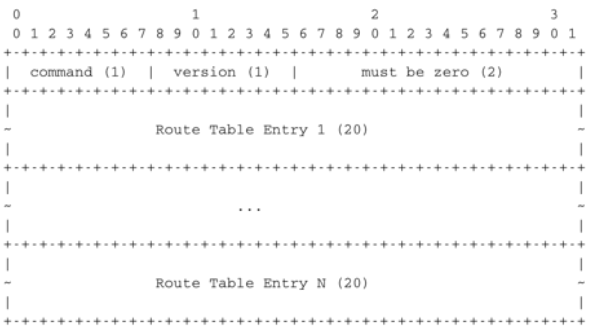


Figure 3.60: The RIP message format

Each RIP message contains a set of route entries. Each route entry is encoded as a 20 bytes field whose format is shown below. RIP was initially designed to be suitable for different network layer protocols. Some implementations of RIP were used in XNS or IPX networks **RFC 2453**. The format of the route entries used by **RFC 2080** is shown below. *Plen* is the length of the subnet identifier in bits and the metric is encoded as one byte. The maximum metric supported by RIP is *15*.

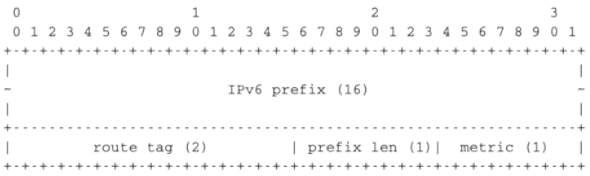


Figure 3.61: Format of the RIP IPv6 route entries

Note: A note on timers

The first RIP implementations sent their distance vector exactly every 30 seconds. This worked well in most networks, but some researchers noticed that routers were sometimes overloaded because they were processing too many distance vectors at the same time [FJ1994]. They collected packet traces in these networks and found that after some time the routers' timers became synchronised, i.e. almost all routers were sending their distance vectors at almost the same time. This synchronisation of the transmission times of the distance vectors caused an overload on the routers' CPU but also increased the convergence time of the protocol in some cases. This was mainly due to the fact that all routers set their timers to the same expiration time after having processed the received distance vectors. **'Sally Floyd'_** and **'Van Jacobson'_** proposed in [FJ1994] a simple solution to solve this synchronisation problem. Instead of advertising their distance vector exactly after 30 seconds, a router should send its next distance vector after a delay chosen randomly in the [15,45] interval **RFC 2080**. This randomisation of the delays prevents the synchronisation that occurs with a fixed delay and is now a recommended practice for protocol designers.

3.14.2 OSPF

Link-state routing protocols are used in IP networks. Open Shortest Path First (OSPF), defined in **RFC 2328**, is the link state routing protocol that has been standardised by the IETF. The last version of OSPF, which supports IPv6, is defined in **RFC 5340**. OSPF is frequently used in enterprise networks and in some ISP networks. However, ISP networks often use the IS-IS link-state routing protocol [ISO10589] , which was developed for the ISO CLNP protocol but was adapted to be used in IP **RFC 1195** networks before the finalisation of the standardisation of OSPF. A detailed analysis of ISIS and OSPF may be found in [BMO2006] and [Perlman2000]. Additional information about OSPF may be found in [Moy1998].

Compared to the basics of link-state routing protocols that we discussed in section *Link state routing*, there are some particularities of OSPF that are worth discussing. First, in a large network, flooding the information about all routers and links to thousands of routers or more may be costly as each router needs to store all the information about the entire network. A better approach would be to introduce hierarchical routing. Hierarchical routing divides the network into regions. All the routers inside a region have detailed information about the topology of the region but only learn aggregated information about the topology of the other regions and their interconnections. OSPF supports a restricted variant of hierarchical routing. In OSPF's terminology, a region is called an *area*.

OSPF imposes restrictions on how a network can be divided into areas. An area is a set of routers and links that are grouped together. Usually, the topology of an area is chosen so that a packet sent by one router inside the area can reach any other router in the area without leaving the area [43] . An OSPF area contains two types of routers **RFC 2328**:

- Internal router : A router whose directly connected networks belong to the area
- Area border routers : A router that is attached to several areas.

For example, the network shown in the figure below has been divided into three areas : *area 1*, containing routers *R1*, *R3*, *R4*, *R5* and *RA*, *area 2* containing *R7*, *R8*, *R9*, *R10*, *RB* and *RC*. OSPF areas are identified by a 32 bit integer, which is sometimes represented as an IP address. Among the OSPF areas, *area 0*, also called the *backbone area* has a special role. The backbone area groups all the area border routers (routers *RA*, *RB* and *RC* in the figure below) and the routers that are directly connected to the backbone routers but do not belong to another area (router *RD* in the figure below). An important restriction imposed by OSPF is that the path between two routers that belong to two different areas (e.g. *R1* and *R8* in the figure below) must pass through the backbone area.

Inside each non-backbone area, routers distribute the topology of the area by exchanging link state packets with the other routers in the area. The internal routers do not know the topology of other areas, but each router knows how to reach the backbone area. Inside an area, the routers only exchange link-state packets for all destinations that are reachable inside the area. In OSPF, the inter-area routing is done by exchanging distance vectors. This is illustrated by the network topology shown below.

Let us first consider OSPF routing inside *area 2*. All routers in the area learn a route towards *2001:db8:1234::/48* and *2001:db8:5678::/48*. The two area border routers, *RB* and *RC*, create network summary advertisements.

[43] OSPF can support *virtual links* to connect routers together that belong to the same area but are not directly connected. However, this goes beyond this introduction to OSPF.

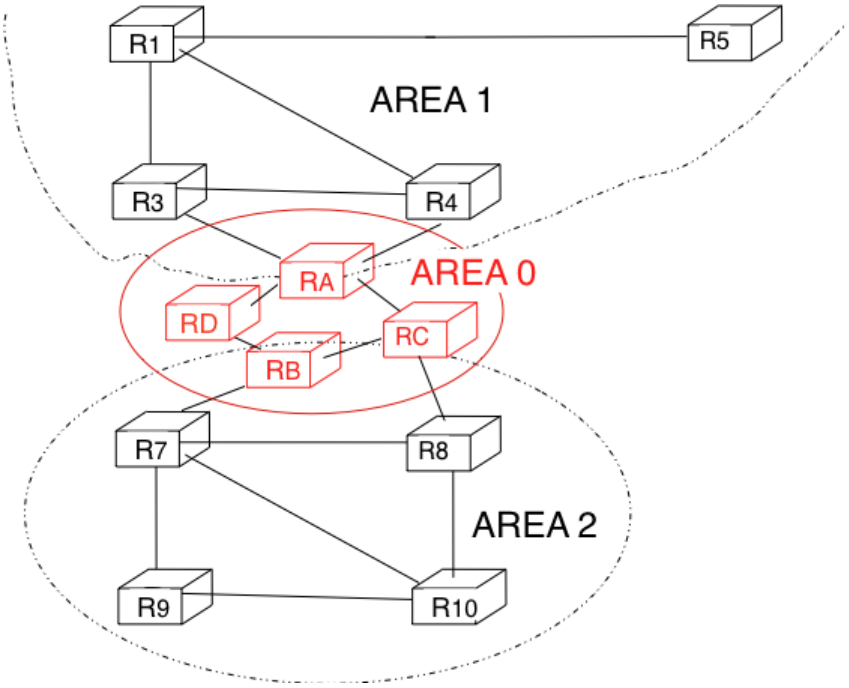


Figure 3.62: OSPF areas

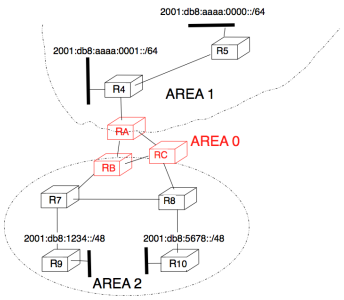


Figure 3.63: Hierarchical routing with OSPF

Assuming that all links have a unit link metric, these would be:

- *RB* advertises *2001:db8:1234::/48* at a distance of *2* and *2001:db8:5678::/48* at a distance of *3*
- *RC* advertises *2001:db8:5678::/48* at a distance of *2* and *2001:db8:1234::/48* at a distance of *3*

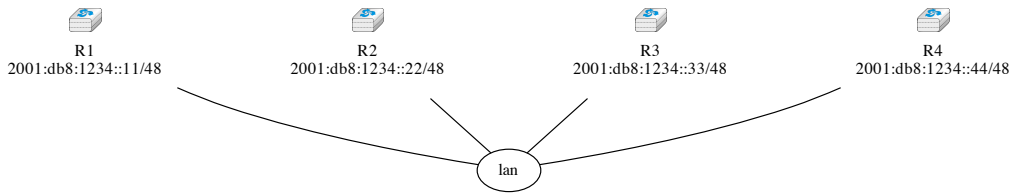
These summary advertisements are flooded through the backbone area attached to routers *RB* and *RC*. In its routing table, router *RA* selects the summary advertised by *RB* to reach *2001:db8:1234::/48* and the summary advertised by *RC* to reach *2001:db8:5678::/48*. Inside *area 1*, router *RA* advertises a summary indicating that *2001:db8:1234::/48* and *2001:db8:5678::/48* are both at a distance of *3* from itself.

On the other hand, consider the prefixes *2001:db8:aaaa:0000::/64* and *2001:db8:aaaa:0001::/64* that are inside *area 1*. Router *RA* is the only area border router that is attached to this area. This router can create two different network summary advertisements :

- *2001:db8:aaaa:0000::/64* at a distance of *1* and *2001:db8:aaaa:0001::/64* at a distance of *2* from *RA*
- *2001:db8:aaaa:0000::/63* at a distance of *2* from *RA*

The first summary advertisement provides precise information about the distance used to reach each prefix. However, all routers in the network have to maintain a route towards *2001:db8:aaaa:0000::/64* and a route towards *2001:db8:aaaa:0001::/64* that are both via router *RA*. The second advertisement would improve the scalability of OSPF by reducing the number of routes that are advertised across area boundaries. However, in practice this requires manual configuration on the border routers.

The second OSPF particularity that is worth discussing is the support of Local Area Networks (LAN). As shown in the example below, several routers may be attached to the same LAN.



A first solution to support such a LAN with a link-state routing protocol would be to consider that a LAN is equivalent to a full-mesh of point-to-point links as if each router can directly reach any other router on the LAN. However, this approach has two important drawbacks :

1. Each router must exchange HELLOs and link state packets with all the other routers on the LAN. This increases the number of OSPF packets that are sent and processed by each router.
2. Remote routers, when looking at the topology distributed by OSPF, consider that there is a full-mesh of links between all the LAN routers. Such a full-mesh implies a lot of redundancy in case of failure, while in practice the entire LAN may completely fail. In case of a failure of the entire LAN, all routers need to detect the failures and flood link state packets before the LAN is completely removed from the OSPF topology by remote routers.

To better represent LANs and reduce the number of OSPF packets that are exchanged, OSPF handles LAN differently. When OSPF routers boot on a LAN, they elect [44] one of them as the *Designated Router (DR)* **RFC 2328**. The *DR* router *represents* the local area network, and advertises the LAN's subnet. Furthermore, LAN routers only exchange HELLO packets with the *DR*. Thanks to the utilisation of a *DR*, the topology of the LAN appears as a set of point-to-point links connected to the *DR* router.

Note: How to quickly detect a link failure ?

Network operators expect an OSPF network to be able to quickly recover from link or router failures [VPD2004]. In an OSPF network, the recovery after a failure is performed in three steps [FFEB2005] :

[44] The OSPF Designated Router election procedure is defined in **RFC 2328**. Each router can be configured with a router priority that influences the election process since the router with the highest priority is preferred when an election is run.

- the routers that are adjacent to the failure detect it quickly. The default solution is to rely on the regular exchange of HELLO packets. However, the interval between successive HELLOs is often set to 10 seconds... Setting the HELLO timer down to a few milliseconds is difficult as HELLO packets are created and processed by the main CPU of the routers and these routers cannot easily generate and process a HELLO packet every millisecond on each of their interfaces. A better solution is to use a dedicated failure detection protocol such as the Bidirectional Forwarding Detection (BFD) protocol defined in [KW2009] that can be implemented directly on the router interfaces. Another solution to be able to detect the failure is to instrument the physical and the datalink layer so that they can interrupt the router when a link fails. Unfortunately, such a solution cannot be used on all types of physical and datalink layers.
- the routers that have detected the failure flood their updated link state packets in the network
- all routers update their routing table

A last, but operationally important, point needs to be discussed about intradomain routing protocols such as OSPF and IS-IS. Intradomain routing protocols always select the shortest path for each destination. In practice, there are often several equal paths towards the same destination. When a router computes several equal cost paths towards one destination, it can use these paths in different ways.

A first approach is to select one of the equal cost paths (e.g. the first or the last path found by the SPF computation) and install it in the forwarding table. In this case, only one path is used to reach each destination.

A second approach is to install all equal cost paths [45] in the forwarding table and load-balance the packets on the different paths. Consider the case where a router has *N* different outgoing interfaces to reach destination *d*. A first possibility to load-balance the traffic among these interfaces is to use *round-robin*. *Round-robin* allows to equally balance the packets among the *N* outgoing interfaces. This equal load-balancing is important in practice because it allows to better spread the load throughout the network. However, few networks use this *round-robin* strategy to load-balance traffic on routers. The main drawback of *round-robin* is that packets that belong to the same flow (e.g. TCP connection) may be forwarded over different paths. If packets belonging to the same TCP connection are sent over different paths, they will probably experience different delays and arrive out-of-sequence at their destination. When a TCP receiver detects out-of-order segments, it sends duplicate acknowledgements that may cause the sender to initiate a fast retransmission and enter congestion avoidance. Thus, out-of-order segments may lead to lower TCP performance. This is annoying for a load-balancing technique whose objective is to improve the network performance by spreading the load.

To efficiently spread the load over different paths, routers need to implement *per-flow* load-balancing. This implies that they must forward all the packets that belong to the same flow on the same path. Since a TCP connection is always identified by the four-tuple (source and destination addresses, source and destination ports), one possibility would be to select an outgoing interface upon arrival of the first packet of the flow and store this decision in the router's memory. Unfortunately, such a solution does not scale since the required memory grows with the number of TCP connections that pass through the router.

Fortunately, it is possible to perform *per-flow* load balancing without maintaining any state on the router. Most routers today use hash functions for this purpose **RFC 2991**. When a packet arrives, the router extracts the Next Header information and the four-tuple from the packet and computes :

$$hash(NextHeader, IP_{src}, IP_{dst}, Port_{src}, Port_{dst}) mod N$$

In this formula, *N* is the number of outgoing interfaces on the equal cost paths towards the packet's destination. Various hash functions are possible, including CRC, checksum or MD5 **RFC 2991**. Since the hash function is computed over the four-tuple, the same hash value will be computed for all packets belonging to the same flow. This prevents reordering due to load balancing inside the network. Most routers support this kind of load-balancing today [ACO+2006].

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues/new

[45] In some networks, there are several dozens of paths towards a given destination. Some routers, due to hardware limitations, cannot install more than 8 or 16 paths in their forwarding table. In this case, a subset of the computed paths is installed in the forwarding table.

3.15 Interdomain routing

As explained earlier, the Internet is composed of more than 45,000 different networks [46] called *domains*. Each domain is composed of a group of routers and hosts that are managed by the same organisation. Example domains include belnet, sprint, level3, geant, abilene, cisco or google ...

Each domain contains a set of routers. From a routing point of view, these domains can be divided into two classes : the *transit* and the *stub* domains. A *stub* domain sends and receives packets whose source or destination are one of its own hosts. A *transit* domain is a domain that provides a transit service for other domains, i.e. the routers in this domain forward packets whose source and destination do not belong to the transit domain. As of this writing, about 85% of the domains in the Internet are stub domains [47]. A *stub* domain that is connected to a single transit domain is called a *single-homed stub*. A *multihomed stub* is a *stub* domain connected to two or more transit providers.

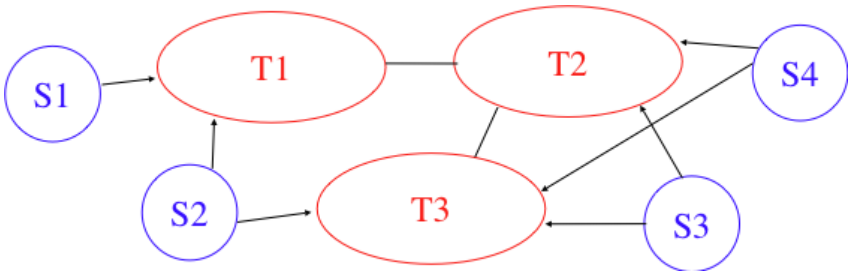


Figure 3.64: Transit and stub domains

The stub domains can be further classified by considering whether they mainly send or receive packets. An *access-rich* stub domain is a domain that contains hosts that mainly receive packets. Typical examples include small ADSL- or cable modem-based Internet Service Providers or enterprise networks. On the other hand, a *content-rich* stub domain is a domain that mainly produces packets. Examples of *content-rich* stub domains include google, yahoo, microsoft, facebook or content distribution networks such as akamai or limelight For the last few years, we have seen a rapid growth of these *content-rich* stub domains. Recent measurements [ATLAS2009] indicate that a growing fraction of all the packets exchanged on the Internet are produced in the data centers managed by these content providers.

Domains need to be interconnected to allow a host inside a domain to exchange IP packets with hosts located in other domains. From a physical perspective, domains can be interconnected in two different ways. The first solution is to directly connect a router belonging to the first domain with a router inside the second domain. Such links between domains are called private interdomain links or *private peering links*. In practice, for redundancy or performance reasons, distinct physical links are usually established between different routers in the two domains that are interconnected.

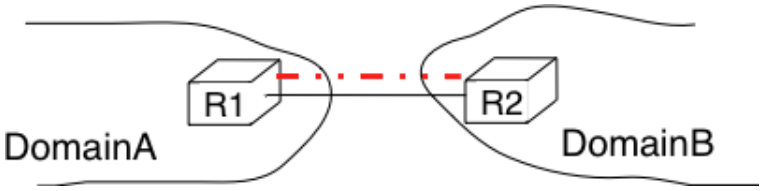


Figure 3.65: Interconnection of two domains via a private peering link

Such *private peering links* are useful when, for example, an enterprise or university network needs to be connected to its Internet Service Provider. However, some domains are connected to hundreds of other domains [48] . For some

[46] An analysis of the evolution of the number of domains on the global Internet during the last ten years may be found in http://www.potaroo.net/tools/asn32/

[47] Several web sites collect and analyse data about the evolution of BGP in the global Internet. http://bgp.potaroo.net provides lots of statistics and analyses that are updated daily.

[48] See http://as-rank.caida.org/ for an analysis of the interconnections between domains based on measurements collected in the global Internet

of these domains, using only private peering links would be too costly. A better solution to allow many domains to interconnect cheaply are the *Internet eXchange Points* (*IXP*). An *IXP* is usually some space in a data center that hosts routers belonging to different domains. A domain willing to exchange packets with other domains present at the *IXP* installs one of its routers on the *IXP* and connects it to other routers inside its own network. The IXP contains a Local Area Network to which all the participating routers are connected. When two domains that are present at the IXP wish [49] to exchange packets, they simply use the Local Area Network. IXPs are very popular in Europe and many Internet Service Providers and Content providers are present in these IXPs.

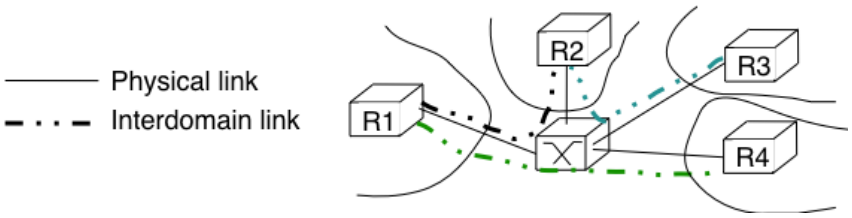


Figure 3.66: Interconnection of two domains at an Internet eXchange Point

In the early days of the Internet, domains would simply exchange all the routes they know to allow a host inside one domain to reach any host in the global Internet. However, in today's highly commercial Internet, this is no longer true as interdomain routing mainly needs to take into account the economical relationships between the domains. Furthermore, while intradomain routing usually prefers some routes over others based on their technical merits (e.g. prefer route with the minimum number of hops, prefer route with the minimum delay, prefer high bandwidth routes over low bandwidth ones, etc) interdomain routing mainly deals with economical issues. For interdomain routing, the cost of using a route is often more important than the quality of the route measured by its delay or bandwidth.

There are different types of economical relationships that can exist between domains. Interdomain routing converts these relationships into peering relationships between domains that are connected via peering links.

The first category of peering relationship is the *customer->provider* relationship. Such a relationship is used when a customer domain pays an Internet Service Provider to be able to exchange packets with the global Internet over an interdomain link. A similar relationship is used when a small Internet Service Provider pays a larger Internet Service Provider to exchange packets with the global Internet.

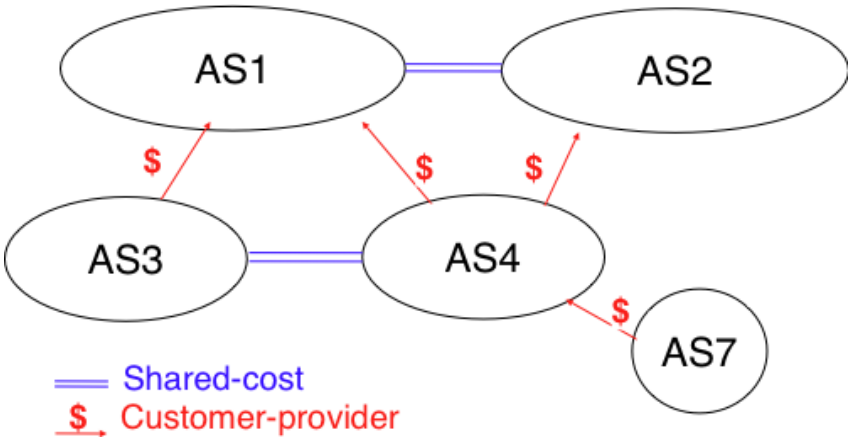


Figure 3.67: A simple Internet with peering relationships

To understand the *customer->provider* relationship, let us consider the simple internetwork shown in the figure above. In this internetwork, *AS7* is a stub domain that is connected to one provider : *AS4*. The contract between *AS4* and *AS7* allows a host inside *AS7* to exchange packets with any host in the internetwork. To enable this exchange of packets, *AS7* must know a route towards any domain and all the domains of the internetwork must

[49] Two routers that are attached to the same IXP only exchange packets when the owners of their domains have an economical incentive to exchange packets on this IXP. Usually, a router on an IXP is only able to exchange packets with a small fraction of the routers that are present on the same IXP.

know a route via *AS4* that allows them to reach hosts inside *AS7*. From a routing perspective, the commercial contract between *AS7* and *AS4* leads to the following routes being exchanged :

- over a *customer->provider* relationship, the *customer* domain advertises to its *provider* all its routes and all the routes that it has learned from its own customers.
- over a *provider->customer* relationship, the *provider* advertises all the routes that it knows to its *customer*.

The second rule ensures that the customer domain receives a route towards all destinations that are reachable via its provider. The first rule allows the routes of the customer domain to be distributed throughout the Internet.

Coming back to the figure above, *AS4* advertises to its two providers *AS1* and *AS2* its own routes and the routes learned from its customer, *AS7*. On the other hand, *AS4* advertises to *AS7* all the routes that it knows.

The second type of peering relationship is the *shared-cost* peering relationship. Such a relationship usually does not involve a payment from one domain to the other in contrast with the *customer->provider* relationship. A *shared-cost* peering relationship is usually established between domains having a similar size and geographic coverage. For example, consider the figure above. If *AS3* and *AS4* exchange many packets via *AS1*, they both need to pay *AS1*. A cheaper alternative for *AS3* and *AS4* would be to establish a *shared-cost* peering. Such a peering can be established at IXPs where both *AS3* and *AS4* are present or by using private peering links. This *shared-cost* peering should be used to exchange packets between hosts inside *AS3* and hosts inside *AS4*. However, *AS3* does not want to receive on the *AS3-AS4 shared-cost* peering links packets whose destination belongs to *AS1* as *AS3* would have to pay to send these packets to *AS1*.

From a routing perspective, over a *shared-cost* peering relationship a domain only advertises its internal routes and the routes that it has learned from its customers. This restriction ensures that only packets destined to the local domain or one of its customers is received over the *shared-cost* peering relationship. This implies that the routes that have been learned from a provider or from another *shared-cost* peer is not advertised over a *shared-cost* peering relationship. This is motivated by economical reasons. If a domain were to advertise the routes that it learned from a provider over a *shared-cost* peering relationship that does not bring revenue, it would have allowed its *shared-cost* peer to use the link with its provider without any payment. If a domain were to advertise the routes it learned over a *shared cost* peering over another *shared-cost* peering relationship, it would have allowed these *shared-cost* peers to use its own network (which may span one or more continents) freely to exchange packets.

Finally, the last type of peering relationship is the *sibling*. Such a relationship is used when two domains exchange all their routes in both directions. In practice, such a relationship is only used between domains that belong to the same company.

These different types of relationships are implemented in the *interdomain routing policies* defined by each domain. The *interdomain routing policy* of a domain is composed of three main parts :

- the *import filter* that specifies, for each peering relationship, the routes that can be accepted from the neighbouring domain (the non-acceptable routes are ignored and the domain never uses them to forward packets)
- the *export filter* that specifies, for each peering relationship, the routes that can be advertised to the neighbouring domain
- the *ranking* algorithm that is used to select the best route among all the routes that the domain has received towards the same destination prefix

A domain's import and export filters can be defined by using the Route Policy Specification Language (RPSL) specified in **RFC 2622** [GAVE1999] . Some Internet Service Providers, notably in Europe, use RPSL to document [50] their import and export policies. Several tools help to easily convert a RPSL policy into router commands.

The figure below provides a simple example of import and export filters for two domains in a simple internetwork. In RPSL, the keyword *ANY* is used to replace any route from any domain. It is typically used by a provider to indicate that it announces all its routes to a customer over a *provider->customer* relationship. This is the case for *AS4*'s export policy. The example below clearly shows the difference between a *provider->customer* and a *shared-cost* peering relationship. *AS4*'s export filter indicates that it announces only its internal routes (*AS4*) and the routes learned from its clients (*AS7*) over its *shared-cost* peering with *AS3*, while it advertises all the routes that it uses (including the routes learned from *AS3*) to *AS7*.

[50] See ftp://ftp.ripe.net/ripe/dbase for the RIPE database that contains the import and export policies of many European ISPs

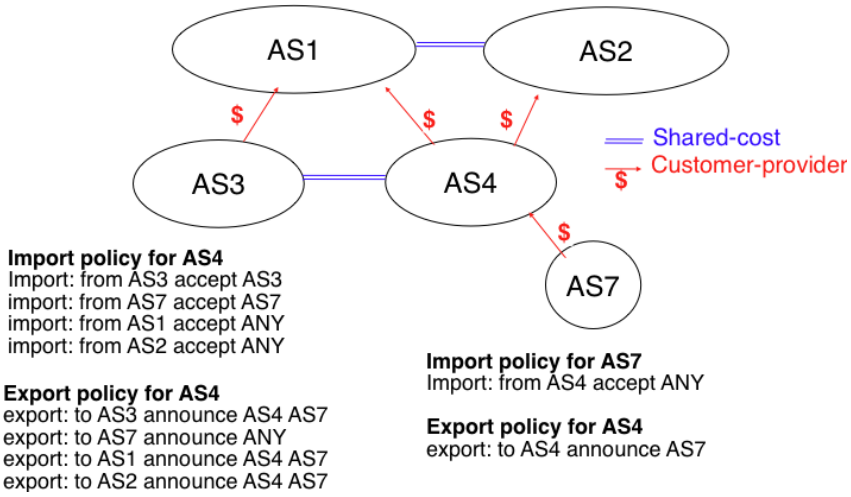


Figure 3.68: Import and export policies

3.15.1 The Border Gateway Protocol

The Internet uses a single interdomain routing protocol : the Border Gateway Protocol (BGP). The current version of BGP is defined in **RFC 4271**. BGP differs from the intradomain routing protocols that we have already discussed in several ways. First, BGP is a *path-vector* protocol. When a BGP router advertises a route towards a prefix, it announces the IP prefix and the interdomain path used to reach this prefix. From BGP's point of view, each domain is identified by a unique *Autonomous System* (AS) number [51] and the interdomain path contains the AS numbers of the transit domains that are used to reach the associated prefix. This interdomain path is called the *AS Path*. Thanks to these AS-Paths, BGP does not suffer from the count-to-infinity problems that affect distance vector routing protocols. Furthermore, the AS-Path can be used to implement some routing policies. Another difference between BGP and the intradomain routing protocols is that a BGP router does not send the entire contents of its routing table to its neighbours regularly. Given the size of the global Internet, routers would be overloaded by the number of BGP messages that they would need to process. BGP uses incremental updates, i.e. it only announces the routes that have changed to its neighbors.

The figure below shows a simple example of the BGP routes that are exchanged between domains. In this example, prefix *2001:db8:1234/48* is announced by *AS1*. *AS1* advertises a BGP route towards this prefix to *AS2*. The AS-Path of this route indicates that *AS1* is the originator of the prefix. When *AS4* receives the BGP route from *AS1*, it re-announces it to *AS2* and adds its AS number to the AS-Path. *AS2* has learned two routes towards prefix *2001:db8:1234/48*. It compares the two routes and prefers the route learned from *AS4* based on its own ranking algorithm. *AS2* advertises to *AS5* a route towards *2001:db8:1234/48* with its AS-Path set to *AS2:AS4:AS1*. Thanks to the AS-Path, *AS5* knows that if it sends a packet towards *2001:db8:1234/48* the packet first passes through *AS2*, then through *AS4* before reaching its destination inside *AS1*.

BGP routers exchange routes over BGP sessions. A BGP session is established between two routers belonging to two different domains that are directly connected. As explained earlier, the physical connection between the two routers can be implemented as a private peering link or over an Internet eXchange Point. A BGP session between two adjacent routers runs above a TCP connection (the default BGP port is 179). In contrast with intradomain routing protocols that exchange IP packets or UDP segments, BGP runs above TCP because TCP ensures a reliable delivery of the BGP messages sent by each router without forcing the routers to implement acknowledgements, checksums, etc. Furthermore, the two routers consider the peering link to be up as long as the BGP session and the underlying TCP connection remain up [52]. The two endpoints of a BGP session are called *BGP peers*.

In practice, to establish a BGP session between routers *R1* and *R2* in the figure above, the network administrator of *AS3* must first configure on *R1* the IP address of *R2* on the *R1-R2* link and the AS number of *R2*. Router *R1* then regularly tries to establish the BGP session with *R2*. *R2* only agrees to establish the BGP session with *R1* once it

[51] In this text, we consider Autonomous System and domain as synonyms. In practice, a domain may be divided into several Autonomous Systems, but we ignore this detail.

[52] The BGP sessions and the underlying TCP connection are typically established by the routers when they boot based on information found in their configuration. The BGP sessions are rarely released, except if the corresponding peering link fails or one of the endpoints crashes or needs to be rebooted.

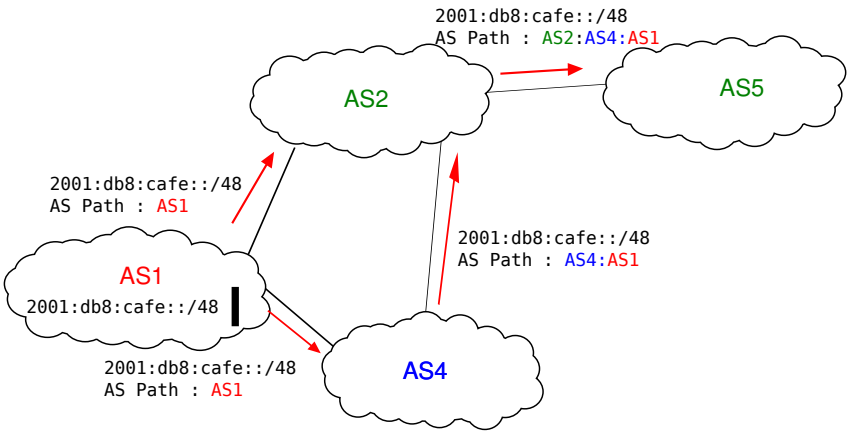


Figure 3.69: Simple exchange of BGP routes

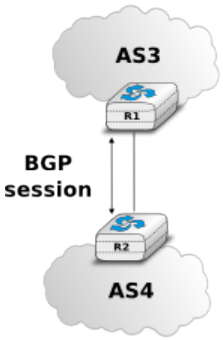


Figure 3.70: A BGP peering session between two directly connected routers

has been configured with the IP address of *R1* and its AS number. For security reasons, a router never establishes a BGP session that has not been manually configured on the router.

The BGP protocol **RFC 4271** defines several types of messages that can be exchanged over a BGP session :

- *OPEN* : this message is sent as soon as the TCP connection between the two routers has been established. It initialises the BGP session and allows the negotiation of some options. Details about this message may be found in **RFC 4271**
- *NOTIFICATION* : this message is used to terminate a BGP session, usually because an error has been detected by the BGP peer. A router that sends or receives a *NOTIFICATION* message immediately shutdowns the corresponding BGP session.
- *UPDATE*: this message is used to advertise new or modified routes or to withdraw previously advertised routes.
- *KEEPALIVE* : this message is used to ensure a regular exchange of messages on the BGP session, even when no route changes. When a BGP router has not sent an *UPDATE* message during the last 30 seconds, it shall send a *KEEPALIVE* message to confirm to the other peer that it is still up. If a peer does not receive any BGP message during a period of 90 seconds [53], the BGP session is considered to be down and all the routes learned over this session are withdrawn.

As explained earlier, BGP relies on incremental updates. This implies that when a BGP session starts, each router first sends BGP *UPDATE* messages to advertise to the other peer all the exportable routes that it knows. Once all these routes have been advertised, the BGP router only sends BGP *UPDATE* messages about a prefix if the route is new, one of its attributes has changed or the route became unreachable and must be withdrawn. The BGP *UPDATE* message allows BGP routers to efficiently exchange such information while minimising the number of bytes exchanged. Each *UPDATE* message contains :

[53] 90 seconds is the default delay recommended by **RFC 4271**. However, two BGP peers can negotiate a different timer during the establishment of their BGP session. Using a too small interval to detect BGP session failures is not recommended. BFD [KW2009] can be used to replace BGP's KEEPALIVE mechanism if fast detection of interdomain link failures is required.

- a list of IP prefixes that are withdrawn
- a list of IP prefixes that are (re-)advertised
- the set of attributes (e.g. AS-Path) associated to the advertised prefixes

In the remainder of this chapter, and although all routing information is exchanged using BGP *UPDATE* messages, we assume for simplicity that a BGP message contains only information about one prefix and we use the words :

- *Withdraw message* to indicate a BGP *UPDATE* message containing one route that is withdrawn
- *Update message* to indicate a BGP *UPDATE* containing a new or updated route towards one destination prefix with its attributes

From a conceptual point of view, a BGP router connected to *N* BGP peers, can be described as being composed of four parts as shown in the figure below.

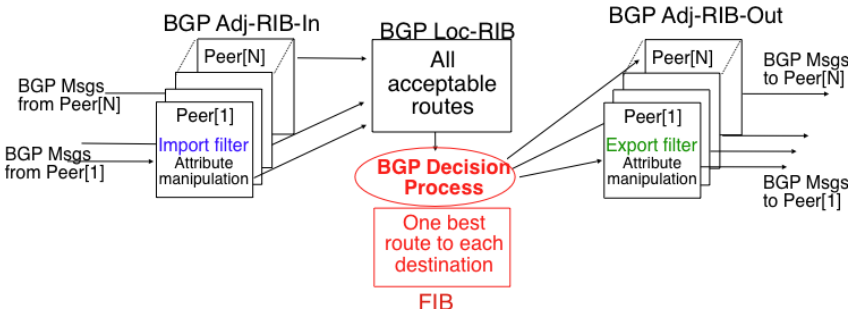


Figure 3.71: Organisation of a BGP router

In this figure, the router receives BGP messages on the left part of the figure, processes these messages and possibly sends BGP messages on the right part of the figure. A BGP router contains three important data structures :

- the *Adj-RIB-In* contains the BGP routes that have been received from each BGP peer. The routes in the *Adj-RIB-In* are filtered by the *import filter* before being placed in the *BGP-Loc-RIB*. There is one *import filter* per BGP peer.
- the *Local Routing Information Base* (*Loc-RIB*) contains all the routes that are considered as acceptable by the router. The *Loc-RIB* may contain several routes, learned from different BGP peers, towards the same destination prefix.
- the *Forwarding Information Base* (*FIB*) is used by the dataplane to forward packets towards their destination. The *FIB* contains, for each destination, the best route that has been selected by the *BGP decision process*. This decision process is an algorithm that selects, for each destination prefix, the best route according to the router's ranking algorithm that is part of its policy.
- the *Adj-RIB-Out* contains the BGP routes that have been advertised to each BGP peer. The *Adj-RIB-Out* for a given peer is built by applying the peer's *export filter* on the routes that have been installed in the *FIB*. There is one *export filter* per BGP peer. For this reason, the Adj-RIB-Out of a peer may contain different routes than the Adj-RIB-Out of another peer.

When a BGP session starts, the routers first exchange *OPEN* messages to negotiate the options that apply throughout the entire session. Then, each router extracts from its FIB the routes to be advertised to the peer. It is important to note that, for each known destination prefix, a BGP router can only advertise to a peer the route that it has itself installed inside its *FIB*. The routes that are advertised to a peer must pass the peer's *export filter*. The *export filter* is a set of rules that define which routes can be advertised over the corresponding session, possibly after having modified some of its attributes. One *export filter* is associated to each BGP session. For example, on a *shared-cost peering*, the *export filter* only selects the internal routes and the routes that have been learned from a *customer*. The pseudo-code below shows the initialisation of a BGP session.

```
def initialize_BGP_session( RemoteAS, RemoteIP):
  # Initialize and start BGP session
  # Send BGP OPEN Message to RemoteIP on port 179
  # Follow BGP state machine
```

```
  # advertise local routes and routes learned from peers*/
  for d in BGPLocRIB :
      B=build_BGP_Update(d)
      S=Apply_Export_Filter(RemoteAS,B)
      if (S != None) :
         send_Update(S,RemoteAS,RemoteIP)
  # entire RIB has been sent
  # new Updates will be sent to reflect local or distant
  # changes in routers
```

In the above pseudo-code, the *build_BGP_UPDATE(d)* procedure extracts from the *BGP Loc-RIB* the best path towards destination *d* (i.e. the route installed in the FIB) and prepares the corresponding BGP *UPDATE* message. This message is then passed to the *export filter* that returns NULL if the route cannot be advertised to the peer or the (possibly modified) BGP *UPDATE* message to be advertised. BGP routers allow network administrators to specify very complex *export filters*, see e.g. [WMS2004]. A simple *export filter* that implements the equivalent of *split horizon* is shown below.

```
def apply_export_filter(RemoteAS, BGPMsg) :
  # check if RemoteAS already received route
  if RemoteAS is BGPMsg.ASPath :
     BGPMsg=None
     # Many additional export policies can be configured :
     # Accept or refuse the BGPMsg
     # Modify selected attributes inside BGPMsg
  return BGPMsg
```

At this point, the remote router has received all the exportable BGP routes. After this initial exchange, the router only sends *BGP UPDATE* messages when there is a change (addition of a route, removal of a route or change in the attributes of a route) in one of these exportable routes. Such a change can happen when the router receives a BGP message. The pseudo-code below summarizes the processing of these BGP messages.

```
def Recvd_BGPMsg(Msg, RemoteAS) :
    B=apply_import_filer(Msg,RemoteAS)
    if (B== None): # Msg not acceptable
       return
    if IsUPDATE(Msg):
       Old_Route=BestRoute(Msg.prefix)
       Insert_in_RIB(Msg)
       Run_Decision_Process(RIB)
       if (BestRoute(Msg.prefix) != Old_Route) :
          # best route changed
          B=build_BGP_Message(Msg.prefix);
          S=apply_export_filter(RemoteAS,B);
          if (S!=None) : # announce best route
            send_UPDATE(S,RemoteAS,RemoteIP);
          else if (Old_Route != None) :
            send_WITHDRAW(Msg.prefix,RemoteAS, RemoteIP)
     else : # Msg is WITHDRAW
         Old_Route=BestRoute(Msg.prefix)
         Remove_from_RIB(Msg)
         Run_Decision_Process(RIB)
         if (Best_Route(Msg.prefix) !=Old_Route):
           # best route changed
           B=build_BGP_Message(Msg.prefix)
           S=apply_export_filter(RemoteAS,B)
           if (S != None) : # still one best route towards Msg.prefix
              send_UPDATE(S,RemoteAS, RemoteIP);
           else if(Old_Route != None) : # No best route anymore
               send_WITHDRAW(Msg.prefix,RemoteAS,RemoteIP);
```

When a BGP message is received, the router first applies the peer's *import filter* to verify whether the message is acceptable or not. If the message is not acceptable, the processing stops. The pseudo-code below shows a simple *import filter*. This *import filter* accepts all routes, except those that already contain the local AS in their AS-Path.

If such a route was used, it would cause a routing loop. Another example of an *import filter* would be a filter used by an Internet Service Provider on a session with a customer to only accept routes towards the IP prefixes assigned to the customer by the provider. On real routers, *import filters* can be much more complex and some *import filters* modify the attributes of the received BGP *UPDATE* [WMS2004] .

```
def apply_import_filter(RemoteAS, BGPMsg):
    if MysAS in BGPMsg.ASPath :
       BGPMsg=None
       # Many additional import policies can be configured :
       # Accept or refuse the BGPMsg
       # Modify selected attributes inside BGPMsg
    return BGPMsg
```

Note: The bogon filters

Another example of frequently used *import filters* are the filters that Internet Service Providers use to ignore bogon routes. In the ISP community, a bogon route is a route that should not be advertised on the global Internet. Typical examples include the documentation IPv6 prefix (*2001:db8::/32* used for most examples in this book), the loopback address (::1/128‘) or the IPv6 prefixes that have not yet been allocated by IANA. A well managed BGP router should ensure that it never advertises bogons on the global Internet. Detailed information about these bogons may be found in [IMHM2013].

If the import filter accepts the BGP message, the pseudo-code distinguishes two cases. If this is an *Update message* for prefix *p*, this can be a new route for this prefix or a modification of the route's attributes. The router first retrieves from its *RIB* the best route towards prefix *p*. Then, the new route is inserted in the *RIB* and the *BGP decision process* is run to find whether the best route towards destination *p* changes. A BGP message only needs to be sent to the router's peers if the best route has changed. For each peer, the router applies the *export filter* to verify whether the route can be advertised. If yes, the filtered BGP message is sent. Otherwise, a *Withdraw message* is sent. When the router receives a *Withdraw message*, it also verifies whether the removal of the route from its *RIB* caused its best route towards this prefix to change. It should be noted that, depending on the content of the *RIB* and the *export filters*, a BGP router may need to send a *Withdraw message* to a peer after having received an *Update message* from another peer and conversely.

Let us now discuss in more detail the operation of BGP in an IPv6 network. For this, let us consider the simple network composed of three routers located in three different ASes and shown in the figure below.

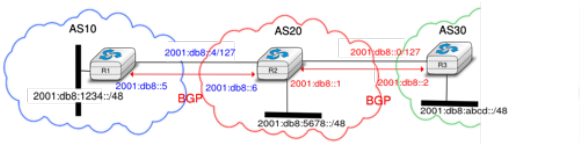


Figure 3.72: Utilisation of the BGP nexthop attribute

This network contains three routers : *R1*, *R2* and *R3*. Each router is attached to a local IPv6 subnet that it advertises using BGP. There are two BGP sessions, one between *R1* and *R2* and the second between *R2* and *R3*. A */127* subnet is used on each interdomain link (*2001:db8::4/127* on *R1-R2* and *2001:db8::0/127* on *R2-R3*) in conformance with the latest recommendation **RFC 6164**. The BGP sessions run above TCP connections established between the neighboring routers (e.g. *2001:db8::5 - 2001:db8::6* for the *R1-R2* session).

Let us assume that the *R1-R2* BGP session is the first to be established. A *BGP Update* message sent on such a session contains three fields :

- the advertised prefix
- the *BGP nexthop*
- the attributes including the AS-Path

We use the notation *U(prefix, nexthop, attributes)* to represent such a *BGP Update* message in this section. Similarly, *W(prefix)* represents a *BGP withdraw* for the specified prefix. Once the *R1-R2* session has been established, *R1* sends *U(2001:db8:1234::/48,2001:db8::5,AS10)* to *R2* and *R2* sends

U(2001:db8:5678:/48,2001:db8::6,AS20). At this point, *R1* can reach *2001:db8:5678::/48* via *2001:db8::6* and *R2* can reach *2001:db8:1234::/47* via *2001:db8::5*.

Once the *R2-R3* has been established, *R3* sends *U(2001:db8:acbd::/48,2001:db8::2,AS30)*. *R2* announces on the *R2-R3* session all the routes inside its RIB. It thus sends to *R3* : *U(2001:db8:1234::/48,2001:db8::1,AS20:AS10)* and *U(2001:db8:5678::/48,2001:db8::1,AS20)*. Note that when *R2* advertises the route that it learned from *R1*, it updates the BGP nexthop and adds its AS number to the AS-Path. *R2* also sends *U(2001:db8:abcd::48,2001:db8::6,AS20:AS30)* to *R1* on the *R1-R3* session. At this point, all BGP routes have been exchanged and all routers can reach *2001:db8::1234/48*, *2001:db8:5678::/48* and *2001:db8:abcd::/48*.

If the link between *R2* and *R3* fails, *R3* detects the failure as it did not receive *KEEPALIVE* messages recently from *R2*. At this time, *R3* removes from its RIB all the routes learned over the *R2-R3* BGP session. *R2* also removes from its RIB the routes learned from *R3*. *R2* also sends *W(2001:db8:acbd::/48)* to *R1* over the *R1-R3* BGP session since it does not have a route anymore towards this prefix.

Note: Origin of the routes advertised by a BGP router

A frequent practical question about the operation of BGP is how a BGP router decides to originate or advertise a route for the first time. In practice, this occurs in two situations :

- the router has been manually configured by the network operator to always advertise one or several routes on a BGP session. For example, on the BGP session between UCLouvain and its provider, belnet , UCLouvain's router always advertises the *2001:6a8:3080/48* IPv6 prefix assigned to the campus network
- the router has been configured by the network operator to advertise over its BGP session some of the routes that it learns with its intradomain routing protocol. For example, an enterprise router may advertise over a BGP session with its provider the routes to remote sites when these routes are reachable and advertised by the intradomain routing protocol

The first solution is the most frequent. Advertising routes learned from an intradomain routing protocol is not recommended, this is because if the route flaps [54], this would cause a large number of BGP messages being exchanged in the global Internet.

The BGP decision process

Besides the import and export filters, a key difference between BGP and the intradomain routing protocols is that each domain can define is own ranking algorithm to determine which route is chosen to forward packets when several routes have been learned towards the same prefix. This ranking depends on several BGP attributes that can be attached to a BGP route.

The first BGP attribute that is used to rank BGP routes is the *local-preference* (local-pref) attribute. This attribute is an unsigned integer that is attached to each BGP route received over an eBGP session by the associated import filter.

When comparing routes towards the same destination prefix, a BGP router always prefers the routes with the highest *local-pref*. If the BGP router knows several routes with the same *local-pref*, it prefers among the routes having this *local-pref* the ones with the shortest AS-Path.

The *local-pref* attribute is often used to prefer some routes over others.

A common utilisation of *local-pref* is to support backup links. Consider the situation depicted in the figure below. *AS1* would always like to use the high bandwidth link to send and receive packets via *AS2* and only use the backup link upon failure of the primary one.

As BGP routers always prefer the routes with the highest *local-pref* attribute, this policy can be implemented using the following import filter on *R1*

```
import: from  AS2 RA at R1 set localpref=100;
        from  AS2 RB at R1 set localpref=200;
        accept ANY
```

[54] A link is said to be flapping if it switches several between an operational state and a disabled state within a short period of time. A router attached to such a link would need to frequently send routing messages.

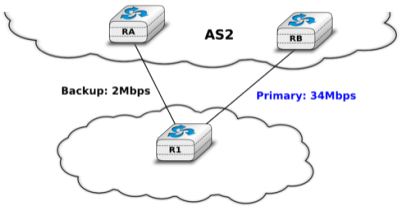


Figure 3.73: How to create a backup link with BGP ?

With this import filter, all the BGP routes learned from *RB* over the high bandwidth links are preferred over the routes learned over the backup link. If the primary link fails, the corresponding routes are removed from *R1*'s RIB and *R1* uses the route learned from *RA*. *R1* reuses the routes via *RB* as soon as they are advertised by *RB* once the *R1-RB* link comes back.

The import filter above modifies the selection of the BGP routes inside *AS1*. Thus, it influences the route followed by the packets forwarded by *AS1*. In addition to using the primary link to send packets, *AS1* would like to receive its packets via the high bandwidth link. For this, *AS2* also needs to set the *local-pref* attribute in its import filter.

```
import: from  AS1 R1 at RA set localpref=100;
        from  AS1 R1 at RB set localpref=200;
        accept AS1
```

Sometimes, the *local-pref* attribute is used to prefer a *cheap* link compared to a more expensive one. For example, in the network below, *AS1* could wish to send and receive packets mainly via its interdomain link with *AS4*.

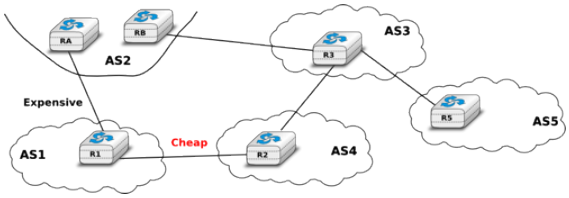


Figure 3.74: How to prefer a cheap link over an more expensive one ?

AS1 can install the following import filter on *R1* to ensure that it always sends packets via *R2* when it has learned a route via *AS2* and another via *AS4*.

```
import: from  AS2 RA at R1 set localpref=100;
        from  AS4 R2 at R1 set localpref=200;
        accept ANY
```

However, this import filter does not influence how *AS3* , for example, prefers some routes over others. If the link between *AS3* and *AS2* is less expensive than the link between *AS3* and *AS4*, *AS3* could send all its packets via *AS2* and *AS1* would receive packets over its expensive link. An important point to remember about *local-pref* is that it can be used to prefer some routes over others to send packets, but it has no influence on the routes followed by received packets.

Another important utilisation of the *local-pref* attribute is to support the *customer->provider* and *shared-cost* peering relationships. From an economic point of view, there is an important difference between these three types of peering relationships. A domain usually earns money when it sends packets over a *provider->customer* relationship. On the other hand, it must pay its provider when it sends packets over a *customer->provider* relationship. Using a *shared-cost* peering to send packets is usually neutral from an economic perspective. To take into account these economic issues, domains usually configure the import filters on their routers as follows :

- insert a high *local-pref* attribute in the routes learned from a customer
- insert a medium *local-pref* attribute in the routes learned over a shared-cost peering
- insert a low *local-pref* attribute in the routes learned from a provider

With such an import filter, the routers of a domain always prefer to reach destinations via their customers whenever such a route exists. Otherwise, they prefer to use *shared-cost* peering relationships and they only send packets via their providers when they do not know any alternate route. A consequence of setting the *local-pref* attribute like this is that Internet paths are often asymmetrical. Consider for example the internetwork shown in the figure below.

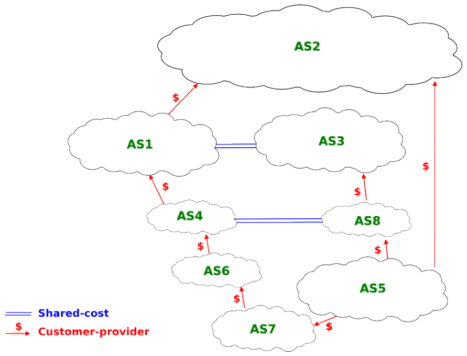


Figure 3.75: Asymmetry of Internet paths

Consider in this internetwork the routes available inside *AS1* to reach *AS5*. *AS1* learns the *AS4:AS6:AS7:AS5* path from *AS4*, the *AS3:AS8:AS5* path from *AS3* and the *AS2:AS5* path from *AS2*. The first path is chosen since it was learned from a customer. *AS5* on the other hand receives three paths towards *AS1* via its providers. It may select any of these paths to reach *AS1* , depending on how it prefers one provider over the others.

BGP convergence

In the previous sections, we have explained the operation of BGP routers. Compared to intradomain routing protocols, a key feature of BGP is its ability to support interdomain routing policies that are defined by each domain as its import and export filters and ranking process. A domain can define its own routing policies and router vendors have implemented many configuration tweaks to support complex routing policies. However, the routing policy chosen by a domain may interfere with the routing policy chosen by another domain. To understand this issue, let us first consider the simple internetwork shown below.

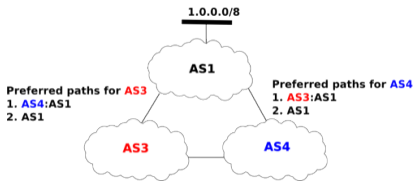


Figure 3.76: The disagree internetwork

In this internetwork, we focus on the route towards *2001:db8::1234/48* which is advertised by *AS1*. Let us also assume that *AS3* (resp. *AS4*) prefers, e.g. for economic reasons, a route learned from *AS4* (*AS3*) over a route learned from *AS1*. When *AS1* sends *U(2001:db8::1234/48,AS1)* to *AS3* and *AS4*, three sequences of exchanges of BGP messages are possible :

1. *AS3* sends first *U(2001:db8:1234/48,AS3:AS1)* to *AS4*. *AS4* has learned two routes towards *2001:db8:1234/48*. It runs its BGP decision process and selects the route via *AS3* and does not advertise a route to *AS3*
2. *AS4* first sends *U(2001:db8:1234/48,AS3:AS1)* to *AS3*. *AS3* has learned two routes towards *2001:db8:1234/48*. It runs its BGP decision process and selects the route via *AS4* and does not advertise a route to *AS4*
3. *AS3* sends *U(2001:db8:1234/48,AS3:AS1)* to *AS4* and, at the same time, *AS4* sends *U(2001:db8:1234/48,AS4:AS1)*. *AS3* prefers the route via *AS4* and thus sends *W(2001:db8:1234/48)* to *AS4*. In the mean time, *AS4* prefers the route via *AS3* and thus sends *W(2001:db8:1234/48)* to *AS3*. Upon

reception of the *BGP Withdraws*, *AS3* and *AS4* only know the direct route towards *2001:db8:1234/48*. *AS3* (resp. *AS4*) sends *U(2001:db8:1234/48,AS3:AS1)* (resp. *U(2001:db8:1234/48,AS4:AS1)*) to *AS4* (resp. *AS3*). *AS3* and *AS4* could in theory continue to exchange BGP messages for ever. In practice, one of them sends one message faster than the other and BGP converges.

The example above has shown that the routes selected by BGP routers may sometimes depend on the ordering of the BGP messages that are exchanged. Other similar scenarios may be found in **RFC 4264**.

From an operational perspective, the above configuration is annoying since the network operators cannot easily predict which paths are chosen. Unfortunately, there are even more annoying BGP configurations. For example, let us consider the configuration below which is often named *Bad Gadget* [GW1999]

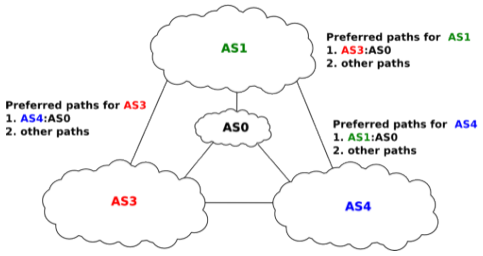


Figure 3.77: The bad gadget internetwork

In this internetwork, there are four ASes. *AS0* advertises one route towards one prefix and we only analyse the routes towards this prefix. The routing preferences of *AS1*, *AS3* and *AS4* are the following :

- *AS1* prefers the path *AS3:AS0* over all other paths
- *AS3* prefers the path *AS4:AS0* over all other paths
- *AS4* prefers the path *AS1:AS0* over all other paths

AS0 sends *U(p,AS0)* to *AS1*, *AS3* and *AS4*. As this is the only route known by *AS1*, *AS3* and *AS4* towards *p*, they all select the direct path. Let us now consider one possible exchange of BGP messages :

1. *AS1* sends *U(p, AS1:AS0)* to *AS3* and *AS4*. *AS4* selects the path via *AS1* since this is its preferred path. *AS3* still uses the direct path.
2. *AS4* advertises *U(p,AS4:AS1:AS0)* to *AS3*.
3. *AS3* sends *U(p, AS3:AS0)* to *AS1* and *AS4*. *AS1* selects the path via *AS3* since this is its preferred path. *AS4* still uses the path via *AS1*.
4. As *AS1* has changed its path, it sends *U(p,AS1:AS3:AS0)* to *AS4* and *W(p)* to *AS3* since its new path is via *AS3*. *AS4* switches back to the direct path.
5. *AS4* sends *U(p,AS4:AS0)* to *AS1* and *AS3*. *AS3* prefers the path via *AS4*.
6. *AS3* sends *U(p,AS3:AS4:AS0)* to *AS1* and *W(p)* to *AS4*. *AS1* switches back to the direct path and we are back at the first step.

This example shows that the convergence of BGP is unfortunately not always guaranteed as some interdomain routing policies may interfere with each other in complex ways. [GW1999] have shown that checking for global convergence is either NP-complete or NP-hard. See [GSW2002] for a more detailed discussion.

Fortunately, there are some operational guidelines [GR2001] [GGR2001] that can guarantee BGP convergence in the global Internet. To ensure that BGP will converge, these guidelines consider that there are two types of peering relationships : *customer->provider* and *shared-cost*. In this case, BGP convergence is guaranteed provided that the following conditions are fulfilled :

1. The topology composed of all the directed *customer->provider* peering links is an acyclic graph
2. An AS always prefers a route received from a *customer* over a route received from a *shared-cost* peer or a *provider*.

The first guideline implies that the provider of the provider of *ASx* cannot be a customer of *ASx*. Such a relationship would not make sense from an economic perspective as it would imply circular payments. Furthermore, providers are usually larger than customers.

The second guideline also corresponds to economic preferences. Since a provider earns money when sending packets to one of its customers, it makes sense to prefer such customer learned routes over routes learned from providers. [GR2001] also shows that BGP convergence is guaranteed even if an AS associates the same preference to routes learned from a *shared-cost* peer and routes learned from a customer.

From a theoretical perspective, these guidelines should be verified automatically to ensure that BGP will always converge in the global Internet. However, such a verification cannot be performed in practice because this would force all domains to disclose their routing policies (and few are willing to do so) and furthermore the problem is known to be NP-hard [GW1999].

In practice, researchers and operators expect that these guidelines are verified [55] in most domains. Thanks to the large amount of BGP data that has been collected by operators and researchers [56], several studies have analysed the AS-level topology of the Internet. [SARK2002] is one of the first analysis. More recent studies include [COZ2008] and [DKF+2007]

Based on these studies and [ATLAS2009], the AS-level Internet topology can be summarised as shown in the figure below.

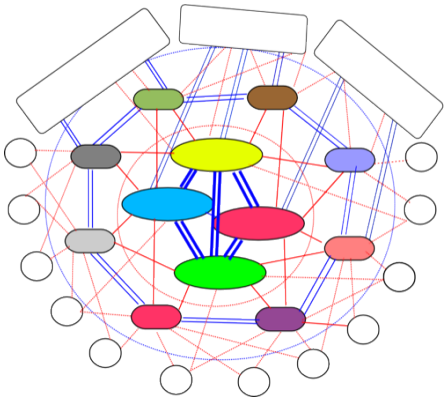


Figure 3.78: The layered structure of the global Internet

The domains on the Internet can be divided in about four categories according to their role and their position in the AS-level topology.

- the core of the Internet is composed of a dozen-twenty *Tier-1* ISPs. A *Tier-1* is a domain that has no *provider*. Such an ISP has *shared-cost* peering relationships with all other *Tier-1* ISPs and *provider->customer* relationships with smaller ISPs. Examples of *Tier-1* ISPs include sprint, level3 or opentransit
- the *Tier-2* ISPs are national or continental ISPs that are customers of *Tier-1* ISPs. These *Tier-2* ISPs have smaller customers and *shared-cost* peering relationships with other *Tier-2* ISPs. Example of *Tier-2* ISPs include France Telecom, Belgacom, British Telecom, ...
- the *Tier-3* networks are either stub domains such as enterprise or campus networks networks and smaller ISPs. They are customers of Tier-1 and Tier-2 ISPs and have sometimes *shared-cost* peering relationships
- the large content providers that are managing large datacenters. These content providers are producing a growing fraction of the packets exchanged on the global Internet [ATLAS2009]. Some of these content providers are customers of Tier-1 or Tier-2 ISPs, but they often try to establish *shared-cost* peering relationships, e.g. at IXPs, with many Tier-1 and Tier-2 ISPs.

[55] Researchers such as [MUF+2007] have shown that modelling the Internet topology at the AS-level requires more than the *shared-cost* and *customer->provider* peering relationships. However, there is no publicly available model that goes beyond these classical peering relationships.

[56] BGP data is often collected by establishing BGP sessions between Unix hosts running a BGP daemon and BGP routers in different ASes. The Unix hosts stores all BGP messages received and regular dumps of its BGP routing table. See http://www.routeviews.org, http://www.ripe.net/ris, http://bgp.potaroo.net or http://irl.cs.ucla.edu/topology/

Due to this organisation of the Internet and due to the BGP decision process, most AS-level paths on the Internet have a length of 3-5 AS hops.

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues/new

3.16 Datalink layer technologies

In this section, we review the key characteristics of several datalink layer technologies. We discuss in more detail the technologies that are widely used today. A detailed survey of all datalink layer technologies would be outside the scope of this book.

3.16.1 The Point-to-Point Protocol

Many point-to-point datalink layers [57] have been developed, starting in the 1960s [McFadyen1976]. In this section, we focus on the protocols that are often used to transport IP packets between hosts or routers that are directly connected by a point-to-point link. This link can be a dedicated physical cable, a leased line through the telephone network or a dial-up connection with modems on the two communicating hosts.

The first solution to transport IP packets over a serial line was proposed in **RFC 1055** and is known as *Serial Line IP* (SLIP). SLIP is a simple character stuffing technique applied to IP packets. SLIP defines two special characters : *END* (decimal 192) and *ESC* (decimal 219). *END* appears at the beginning and at the end of each transmitted IP packet and the sender adds *ESC* before each *END* character inside each transmitted IP packet. SLIP only supports the transmission of IP packets and it assumes that the two communicating hosts/routers have been manually configured with each other's IP address. SLIP was mainly used over links offering bandwidth of often less than 20 Kbps. On such a low bandwidth link, sending 20 bytes of IP header followed by 20 bytes of TCP header for each TCP segment takes a lot of time. This initiated the development of a family of compression techniques to efficiently compress the TCP/IP headers. The first header compression technique proposed in **RFC 1144** was designed to exploit the redundancy between several consecutive segments that belong to the same TCP connection. In all these segments, the IP addresses and port numbers are always the same. Furthermore, fields such as the sequence and acknowledgement numbers do not change in a random way. **RFC 1144** defined simple techniques to reduce the redundancy found in successive segments. The development of header compression techniques continued and there are still improvements being developed now **RFC 5795**.

While SLIP was implemented and used in some environments, it had several limitations discussed in **RFC 1055**. The *Point-to-Point Protocol* (PPP) was designed shortly after and is specified in **RFC 1548**. PPP aims to support IP and other network layer protocols over various types of serial lines. PPP is in fact a family of three protocols that are used together :

1. The *Point-to-Point Protocol* defines the framing technique to transport network layer packets.
2. The *Link Control Protocol* that is used to negotiate options and authenticate the session by using username and password or other types of credentials
3. The *Network Control Protocol* that is specific for each network layer protocol. It is used to negotiate options that are specific for each protocol. For example, IPv4's NCP **RFC 1548** can negotiate the IPv4 address to be used, the IPv4 address of the DNS resolver. IPv6's NCP is defined in **RFC 5072**.

The PPP framing **RFC 1662** was inspired by the datalink layer protocols standardised by ITU-T and ISO. A typical PPP frame is composed of the fields shown in the figure below. A PPP frame starts with a one byte flag containing *01111110*. PPP can use bit stuffing or character stuffing depending on the environment where the protocol is used. The address and control fields are present for backward compatibility reasons. The 16 bit Protocol field contains the identifier [58] of the network layer protocol that is carried in the PPP frame. *0x002d* is used for an IPv4 packet compressed with **RFC 1144** while *0x002f* is used for an uncompressed IPv4 packet. *0xc021* is used by the Link Control Protocol, *0xc023* is used by the Password Authentication Protocol (PAP). *0x0057* is used for IPv6 packets.

[57] LAPB and HDLC were widely used datalink layer protocols.
[58] The IANA maintains the registry of all assigned PPP protocol fields at : http://www.iana.org/assignments/ppp-numbers

PPP supports variable length packets, but LCP can negotiate a maximum packet length. The PPP frame ends with a Frame Check Sequence. The default is a 16 bits CRC, but some implementations can negotiate a 32 bits CRC. The frame ends with the *01111110* flag.

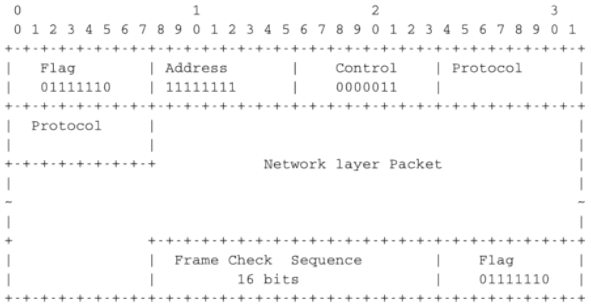


Figure 3.79: PPP frame format

PPP played a key role in allowing Internet Service Providers to provide dial-up access over modems in the late 1990s and early 2000s. ISPs operated modem banks connected to the telephone network. For these ISPs, a key issue was to authenticate each user connected through the telephone network. This authentication was performed by using the *Extensible Authentication Protocol* (EAP) defined in **RFC 3748**. EAP is a simple, but extensible protocol that was initially used by access routers to authenticate the users connected through dialup lines. Several authentication methods, starting from the simple username/password pairs to more complex schemes have been defined and implemented. When ISPs started to upgrade their physical infrastructure to provide Internet access over *Asymmetric Digital Subscriber Lines* (ADSL), they tried to reuse their existing authentication (and billing) systems. To meet these requirements, the IETF developed specifications to allow PPP frames to be transported over other networks than the point-to-point links for which PPP was designed. Nowadays, most ADSL deployments use PPP over either ATM **RFC 2364** or Ethernet **RFC 2516**.

3.16.2 Ethernet

Ethernet was designed in the 1970s at the Palo Alto Research Center [Metcalfe1976]. The first prototype [59] used a coaxial cable as the shared medium and 3 Mbps of bandwidth. Ethernet was improved during the late 1970s and in the 1980s, Digital Equipment, Intel and Xerox published the first official Ethernet specification [DIX]. This specification defines several important parameters for Ethernet networks. The first decision was to standardise the commercial Ethernet at 10 Mbps. The second decision was the duration of the *slot time*. In Ethernet, a long *slot time* enables networks to span a long distance but forces the host to use a larger minimum frame size. The compromise was a *slot time* of 51.2 microseconds, which corresponds to a minimum frame size of 64 bytes.

The third decision was the frame format. The experimental 3 Mbps Ethernet network built at Xerox used short frames containing 8 bit source and destination addresses fields, a 16 bit type indication, up to 554 bytes of payload and a 16 bit CRC. Using 8 bit addresses was suitable for an experimental network, but it was clearly too small for commercial deployments. Although the initial Ethernet specification [DIX] only allowed up to 1024 hosts on an Ethernet network, it also recommended three important changes compared to the networking technologies that were available at that time. The first change was to require each host attached to an Ethernet network to have a globally unique datalink layer address. Until then, datalink layer addresses were manually configured on each host. [DP1981] went against that state of the art and noted "*Suitable installation-specific administrative procedures are also needed for assigning numbers to hosts on a network. If a host is moved from one network to another it may be necessary to change its host number if its former number is in use on the new network. This is easier said than done, as each network must have an administrator who must record the continuously changing state of the system (often on a piece of paper tacked to the wall !). It is anticipated that in future office environments, hosts locations will change as often as telephones are changed in present-day offices.*" The second change introduced by Ethernet was to encode each address as a 48 bits field [DP1981]. 48 bit addresses were huge compared to the networking technologies available in the 1980s, but the huge address space had several advantages [DP1981] including the ability to allocate large blocks of addresses to manufacturers. Eventually, other LAN technologies opted for 48 bits addresses as well **[802]_** . The third change introduced by Ethernet was the definition of *broadcast* and *multicast*

[59] Additional information about the history of the Ethernet technology may be found at http://ethernethistory.typepad.com/

addresses. The need for *multicast* Ethernet was foreseen in [DP1981] and thanks to the size of the addressing space it was possible to reserve a large block of multicast addresses for each manufacturer.

The datalink layer addresses used in Ethernet networks are often called MAC addresses. They are structured as shown in the figure below. The first bit of the address indicates whether the address identifies a network adapter or a multicast group. The upper 24 bits are used to encode an Organisation Unique Identifier (OUI). This OUI identifies a block of addresses that has been allocated by the secretariat [60] that is responsible for the uniqueness of Ethernet addresses to a manufacturer. Once a manufacturer has received an OUI, it can build and sell products with one of the 16 million addresses in this block.

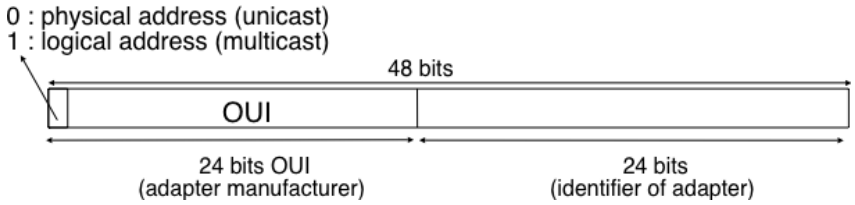


Figure 3.80: 48 bits Ethernet address format

The original 10 Mbps Ethernet specification [DIX] defined a simple frame format where each frame is composed of five fields. The Ethernet frame starts with a preamble (not shown in the figure below) that is used by the physical layer of the receiver to synchronise its clock with the sender's clock. The first field of the frame is the destination address. As this address is placed at the beginning of the frame, an Ethernet interface can quickly verify whether it is the frame recipient and if not, cancel the processing of the arriving frame. The second field is the source address. While the destination address can be either a unicast or a multicast/broadcast address, the source address must always be a unicast address. The third field is a 16 bits integer that indicates which type of network layer packet is carried inside the frame. This field is often called the *EtherType*. Frequently used *EtherType* values [61] include *0x0800* for IPv4, *0x86DD* for IPv6 [62] and *0x806* for the Address Resolution Protocol (ARP).

The fourth part of the Ethernet frame is the payload. The minimum length of the payload is 46 bytes to ensure a minimum frame size, including the header of 512 bits. The Ethernet payload cannot be longer than 1500 bytes. This size was found reasonable when the first Ethernet specification was written. At that time, Xerox had been using its experimental 3 Mbps Ethernet that offered 554 bytes of payload and **RFC 1122** required a minimum MTU of 572 bytes for IPv4. 1500 bytes was large enough to support these needs without forcing the network adapters to contain overly large memories. Furthermore, simulations and measurement studies performed in Ethernet networks revealed that CSMA/CD was able to achieve a very high utilization. This is illustrated in the figure below based on [SH1980], which shows the channel utilization achieved in Ethernet networks containing different numbers of hosts that are sending frames of different sizes.

The last field of the Ethernet frame is a 32 bit Cyclical Redundancy Check (CRC). This CRC is able to catch a much larger number of transmission errors than the Internet checksum used by IP, UDP and TCP [SGP98]. The format of the Ethernet frame is shown below.

Note: Where should the CRC be located in a frame ?

The transport and datalink layers usually chose different strategies to place their CRCs or checksums. Transport layer protocols usually place their CRCs or checksums in the segment header. Datalink layer protocols sometimes place their CRC in the frame header, but often in a trailer at the end of the frame. This choice reflects implementation assumptions, but also influences performance **RFC 893**. When the CRC is placed in the trailer, as in Ethernet, the datalink layer can compute it while transmitting the frame and insert it at the end of the transmission. All Ethernet interfaces use this optimisation today. When the checksum is placed in the header, as in a TCP segment, it is impossible for the network interface to compute it while transmitting the segment. Some network interfaces

[60] Initially, the OUIs were allocated by Xerox [DP1981]. However, once Ethernet became an IEEE and later an ISO standard, the allocation of the OUIs moved to IEEE. The list of all OUI allocations may be found at http://standards.ieee.org/regauth/oui/index.shtml

[61] The official list of all assigned Ethernet type values is available from http://standards.ieee.org/regauth/ethertype/eth.txt

[62] The attentive reader may question the need for different *EtherTypes* for IPv4 and IPv6 while the IP header already contains a version field that can be used to distinguish between IPv4 and IPv6 packets. Theoretically, IPv4 and IPv6 could have used the same *EtherType*. Unfortunately, developers of the early IPv6 implementations found that some devices did not check the version field of the IPv4 packets that they received and parsed frames whose *EtherType* was set to *0x0800* as IPv4 packets. Sending IPv6 packets to such devices would have caused disruptions. To avoid this problem, the IETF decided to apply for a distinct *EtherType* value for IPv6. Such a choice is now mandated by **RFC 6274** (section 3.1), although we can find a funny counter-example in **RFC 6214**.

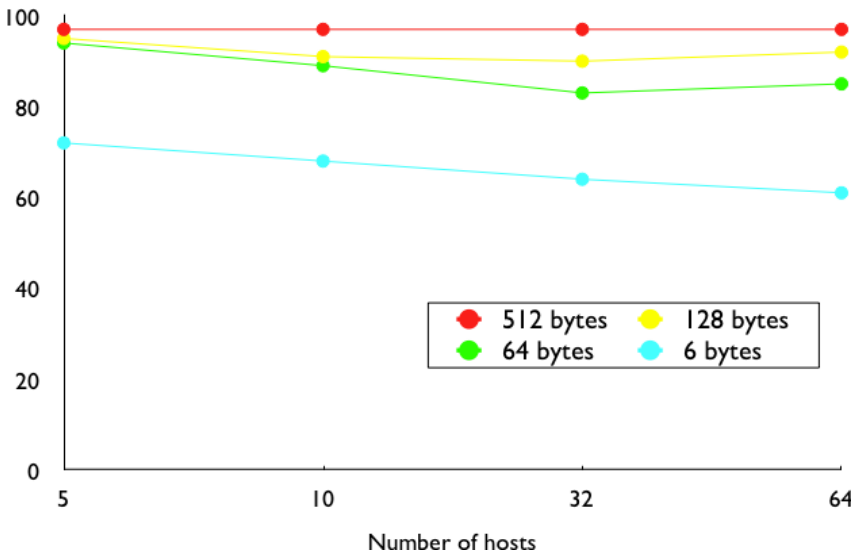


Figure 3.81: Impact of the frame length on the maximum channel utilisation [SH1980]

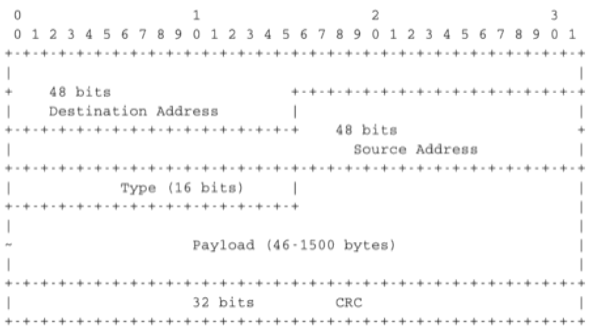


Figure 3.82: Ethernet DIX frame format

provide hardware assistance to compute the TCP checksum, but this is more complex than if the TCP checksum were placed in the trailer [63].

The Ethernet frame format shown above is specified in [DIX]. This is the format used to send both IPv4 **RFC 894** and IPv6 packets **RFC 2464**. After the publication of [DIX], the Institute of Electrical and Electronic Engineers (IEEE) began to standardise several Local Area Network technologies. IEEE worked on several LAN technologies, starting with Ethernet, Token Ring and Token Bus. These three technologies were completely different, but they all agreed to use the 48 bits MAC addresses specified initially for Ethernet **[802]**_ . While developing its Ethernet standard [802.3], the IEEE 802.3 working group was confronted with a problem. Ethernet mandated a minimum payload size of 46 bytes, while some companies were looking for a LAN technology that could transparently transport short frames containing only a few bytes of payload. Such a frame can be sent by an Ethernet host by padding it to ensure that the payload is at least 46 bytes long. However since the Ethernet header [DIX] does not contain a length field, it is impossible for the receiver to determine how many useful bytes were placed inside the payload field. To solve this problem, the IEEE decided to replace the *Type* field of the Ethernet [DIX] header with a length field [64]. This *Length* field contains the number of useful bytes in the frame payload. The payload must still contain at least 46 bytes, but padding bytes are added by the sender and removed by the receiver. In order to add the *Length* field without significantly changing the frame format, IEEE had to remove the *Type* field. Without this field, it is impossible for a receiving host to identify the type of network layer packet inside a received frame. To solve this new problem, IEEE developed a completely new sublayer called the Logical Link Control [802.2]. Several protocols were defined in this sublayer. One of them provided a slightly different version of the *Type* field of the original Ethernet frame format. Another contained acknowledgements and retransmissions to provide a reliable service... In practice, [802.2] is never used to support IP in Ethernet networks. The figure below shows the official [802.3] frame format.

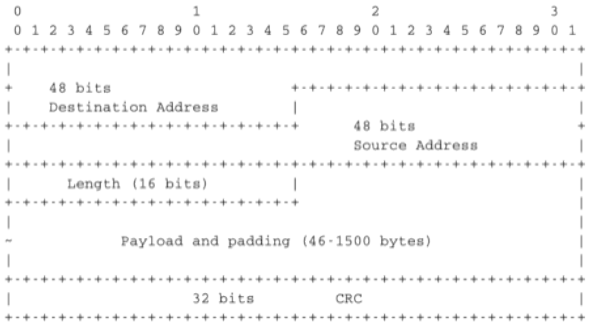


Figure 3.83: Ethernet 802.3 frame format

Note: What is the Ethernet service ?

An Ethernet network provides an unreliable connectionless service. It supports three different transmission modes [*unicast*, *multicast* and *broadcast*. While the Ethernet service is unreliable in theory, a good Ethernet network should, in practice, provide a service that :]

- delivers frames to their destination with a very high probability of successful delivery
- does not reorder the transmitted frames

The first property is a consequence of the utilisation of CSMA/CD. The second property is a consequence of the physical organisation of the Ethernet network as a shared bus. These two properties are important and all evolutions of the Ethernet technology have preserved them.

Several physical layers have been defined for Ethernet networks. The first physical layer, usually called 10Base5, provided 10 Mbps over a thick coaxial cable. The characteristics of the cable and the transceivers that were used

[63] These network interfaces compute the TCP checksum while a segment is transferred from the host memory to the network interface [SH2004].

[64] Fortunately, IEEE was able to define the [802.3] frame format while maintaining backward compatibility with the Ethernet [DIX] frame format. The trick was to only assign values above 1500 as *EtherType* values. When a host receives a frame, it can determine whether the frame's format by checking its *EtherType/Length* field. A value lower smaller than *1501* is clearly a length indicator and thus an [802.3] frame. A value larger than *1501* can only be type and thus a [DIX] frame.

then enabled the utilisation of 500 meter long segments. A 10Base5 network can also include repeaters between segments.

The second physical layer was 10Base2. This physical layer used a thin coaxial cable that was easier to install than the 10Base5 cable, but could not be longer than 185 meters. A 10BaseF physical layer was also defined to transport Ethernet over point-to-point optical links. The major change to the physical layer was the support of twisted pairs in the 10BaseT specification. Twisted pair cables are traditionally used to support the telephone service in office buildings. Most office buildings today are equipped with structured cabling. Several twisted pair cables are installed between any room and a central telecom closet per building or per floor in large buildings. These telecom closets act as concentration points for the telephone service but also for LANs.

The introduction of the twisted pairs led to two major changes to Ethernet. The first change concerns the physical topology of the network. 10Base2 and 10Base5 networks are shared buses, the coaxial cable typically passes through each room that contains a connected computer. A 10BaseT network is a star-shaped network. All the devices connected to the network are attached to a twisted pair cable that ends in the telecom closet. From a maintenance perspective, this is a major improvement. The cable is a weak point in 10Base2 and 10Base5 networks. Any physical damage on the cable broke the entire network and when such a failure occurred, the network administrator had to manually check the entire cable to detect where it was damaged. With 10BaseT, when one twisted pair is damaged, only the device connected to this twisted pair is affected and this does not affect the other devices. The second major change introduced by 10BaseT was that is was impossible to build a 10BaseT network by simply connecting all the twisted pairs together. All the twisted pairs must be connected to a relay that operates in the physical layer. This relay is called an *Ethernet hub*. A *hub* is thus a physical layer relay that receives an electrical signal on one of its interfaces, regenerates the signal and transmits it over all its other interfaces. Some *hubs* are also able to convert the electrical signal from one physical layer to another (e.g. 10BaseT to 10Base2 conversion).

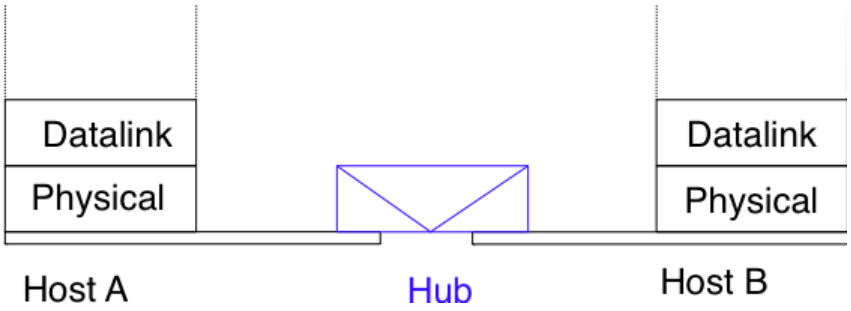


Figure 3.84: Ethernet hubs in the reference model

Computers can directly be attached to Ethernet hubs. Ethernet hubs themselves can be attached to other Ethernet hubs to build a larger network. However, some important guidelines must be followed when building a complex network with hubs. First, the network topology must be a tree. As hubs are relays in the physical layer, adding a link between *Hub2* and *Hub3* in the network below would create an electrical shortcut that would completely disrupt the network. This implies that there cannot be any redundancy in a hub-based network. A failure of a hub or of a link between two hubs would partition the network into two isolated networks. Second, as hubs are relays in the physical layer, collisions can happen and must be handled by CSMA/CD as in a 10Base5 network. This implies that the maximum delay between any pair of devices in the network cannot be longer than the 51.2 microseconds *slot time*. If the delay is longer, collisions between short frames may not be correctly detected. This constraint limits the geographical spread of 10BaseT networks containing hubs.

In the late 1980s, 10 Mbps became too slow for some applications and network manufacturers developed several LAN technologies that offered higher bandwidth, such as the 100 Mbps FDDI LAN that used optical fibers. As the development of 10Base5, 10Base2 and 10BaseT had shown that Ethernet could be adapted to different physical layers, several manufacturers started to work on 100 Mbps Ethernet and convinced IEEE to standardise this new technology that was initially called *Fast Ethernet*. *Fast Ethernet* was designed under two constraints. First, *Fast Ethernet* had to support twisted pairs. Although it was easier from a physical layer perspective to support higher bandwidth on coaxial cables than on twisted pairs, coaxial cables were a nightmare from deployment and maintenance perspectives. Second, *Fast Ethernet* had to be perfectly compatible with the existing 10 Mbps Ethernets to allow *Fast Ethernet* technology to be used initially as a backbone technology to interconnect 10 Mbps Ethernet networks. This forced *Fast Ethernet* to use exactly the same frame format as 10 Mbps Ethernet.

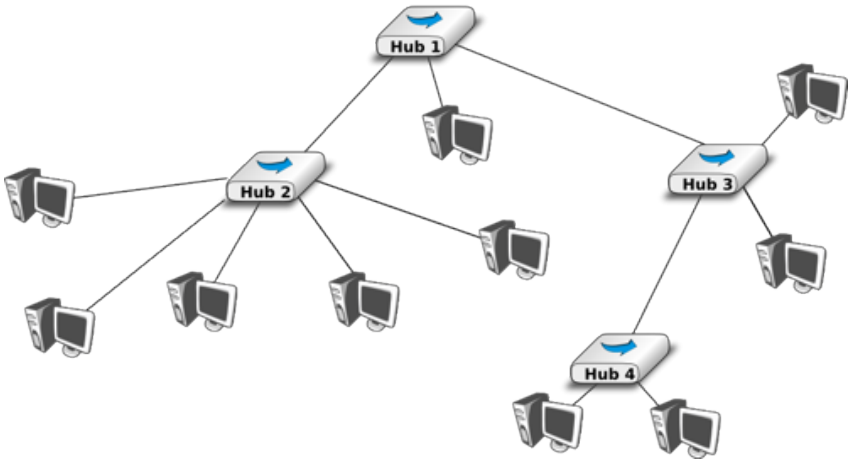


Figure 3.85: A hierarchical Ethernet network composed of hubs

This implied that the minimum *Fast Ethernet* frame size remained at 512 bits. To preserve CSMA/CD with this minimum frame size and 100 Mbps instead of 10 Mbps, the duration of the *slot time* was decreased to 5.12 microseconds.

The evolution of Ethernet did not stop. In 1998, the IEEE published a first standard to provide Gigabit Ethernet over optical fibers. Several other types of physical layers were added afterwards. The 10 Gigabit Ethernet standard appeared in 2002. Work is ongoing to develop standards for 40 Gigabit and 100 Gigabit Ethernet and some are thinking about Terabit Ethernet. The table below lists the main Ethernet standards. A more detailed list may be found at http://en.wikipedia.org/wiki/Ethernet_physical_layer

Standard	Comments
10Base5	Thick coaxial cable, 500m
10Base2	Thin coaxial cable, 185m
10BaseT	Two pairs of category 3+ UTP
10Base-F	10 Mb/s over optical fiber
100Base-Tx	Category 5 UTP or STP, 100 m maximum
100Base-FX	Two multimode optical fiber, 2 km maximum
1000Base-CX	Two pairs shielded twisted pair, 25m maximum
1000Base-SX	Two multimode or single mode optical fibers with lasers
10 Gbps	Optical fiber but also Category 6 UTP
40-100 Gbps	Optical fiber (experiences are performed with copper)

Ethernet Switches

Increasing the physical layer bandwidth as in *Fast Ethernet* was only one of the solutions to improve the performance of Ethernet LANs. A second solution was to replace the hubs with more intelligent devices. As *Ethernet hubs* operate in the physical layer, they can only regenerate the electrical signal to extend the geographical reach of the network. From a performance perspective, it would be more interesting to have devices that operate in the datalink layer and can analyse the destination address of each frame and forward the frames selectively on the link that leads to the destination. Such devices are usually called *Ethernet switches* [65]. An *Ethernet switch* is a relay that operates in the datalink layer as is illustrated in the figure below.

An *Ethernet switch* understands the format of the Ethernet frames and can selectively forward frames over each interface. For this, each *Ethernet switch* maintains a *MAC address table*. This table contains, for each MAC address known by the switch, the identifier of the switch's port over which a frame sent towards this address must be forwarded to reach its destination. This is illustrated below with the *MAC address table* of the bottom switch. When the switch receives a frame destined to address *B*, it forwards the frame on its South port. If it receives a frame destined to address *D*, it forwards it only on its North port.

[65] The first Ethernet relays that operated in the datalink layers were called *bridges*. In practice, the main difference between switches and bridges is that bridges were usually implemented in software while switches are hardware-based devices. Throughout this text, we always use

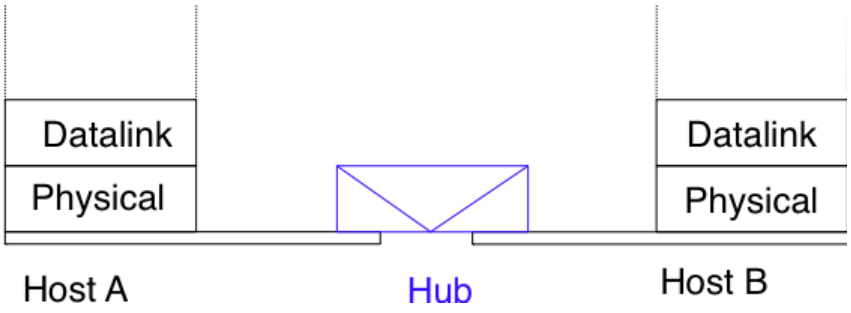


Figure 3.86: Ethernet switches and the reference model

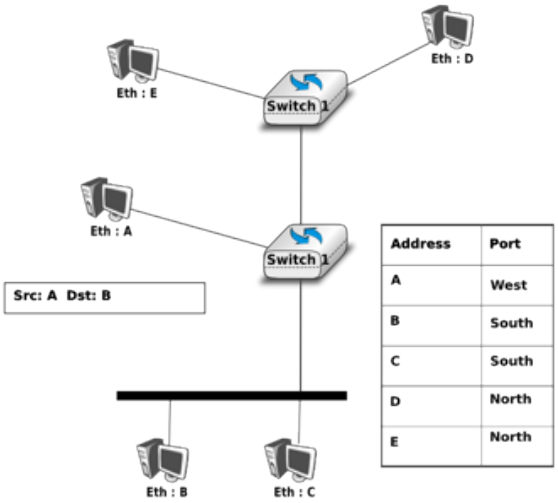


Figure 3.87: Operation of Ethernet switches

One of the selling points of Ethernet networks is that, thanks to the utilisation of 48 bits MAC addresses, an Ethernet LAN is plug and play at the datalink layer. When two hosts are attached to the same Ethernet segment or hub, they can immediately exchange Ethernet frames without requiring any configuration. It is important to retain this plug and play capability for Ethernet switches as well. This implies that Ethernet switches must be able to build their MAC address table automatically without requiring any manual configuration. This automatic configuration is performed by the *MAC address learning* algorithm that runs on each Ethernet switch. This algorithm extracts the source address of the received frames and remembers the port over which a frame from each source Ethernet address has been received. This information is inserted into the MAC address table that the switch uses to forward frames. This allows the switch to automatically learn the ports that it can use to reach each destination address, provided that this host has previously sent at least one frame. This is not a problem since most upper layer protocols use acknowledgements at some layer and thus even an Ethernet printer sends Ethernet frames as well.

The pseudo-code below details how an Ethernet switch forwards Ethernet frames. It first updates its *MAC address table* with the source address of the frame. The *MAC address table* used by some switches also contains a timestamp that is updated each time a frame is received from each known source address. This timestamp is used to remove from the *MAC address table* entries that have not been active during the last *n* minutes. This limits the growth of the *MAC address table*, but also allows hosts to move from one port to another. The switch uses its *MAC address table* to forward the received unicast frame. If there is an entry for the frame's destination address in the *MAC address table*, the frame is forwarded selectively on the port listed in this entry. Otherwise, the switch does not know how to reach the destination address and it must forward the frame on all its ports except the port from which the frame has been received. This ensures that the frame will reach its destination, at the expense of some unnecessary transmissions. These unnecessary transmissions will only last until the destination has sent its first frame. Multicast and Broadcast frames are also forwarded in a similar way.

```
# Arrival of frame F on port P
# Table : MAC address table dictionary : addr->port
# Ports : list of all ports on the switch
src=F.SourceAddress
dst=F.DestinationAddress
Table[src]=P  #src heard on port P
if isUnicast(dst) :
   if dst in Table:
     ForwardFrame(F,Table[dst])
   else:
     for o in Ports :
         if o!= P :  ForwardFrame(F,o)
else:
  # multicast or broadcast destination
  for o in Ports :
      if o!= P :  ForwardFrame(F,o)
```

Note: Security issues with Ethernet hubs and switches

From a security perspective, Ethernet hubs have the same drawbacks as the older coaxial cable. A host attached to a hub will be able to capture all the frames exchanged between any pair of hosts attached to the same hub. Ethernet switches are much better from this perspective thanks to the selective forwarding, a host will usually only receive the frames destined to itself as well as the multicast, broadcast and unknown frames. However, this does not imply that switches are completely secure. There are, unfortunately, attacks against Ethernet switches. From a security perspective, the *MAC address table* is one of the fragile elements of an Ethernet switch. This table has a fixed size. Some low-end switches can store a few tens or a few hundreds of addresses while higher-end switches can store tens of thousands of addresses or more. From a security point of view, a limited resource can be the target of Denial of Service attacks. Unfortunately, such attacks are also possible on Ethernet switches. A malicious host could overflow the *MAC address table* of the switch by generating thousands of frames with random source addresses. Once the *MAC address table* is full, the switch needs to broadcast all the frames that it receives. At this point, an attacker will receive unicast frames that are not destined to its address. The ARP attack discussed in the previous chapter could also occur with Ethernet switches [Vyncke2007]. Recent switches implement several types of defences against these attacks, but they need to be carefully configured by the network administrator. See [Vyncke2007] for a detailed discussion on security issues with Ethernet switches.

switch when referring to a relay in the datalink layer, but you might still see the word *bridge*.

The *MAC address learning* algorithm combined with the forwarding algorithm work well in a tree-shaped network such as the one shown above. However, to deal with link and switch failures, network administrators often add redundant links to ensure that their network remains connected even after a failure. Let us consider what happens in the Ethernet network shown in the figure below.

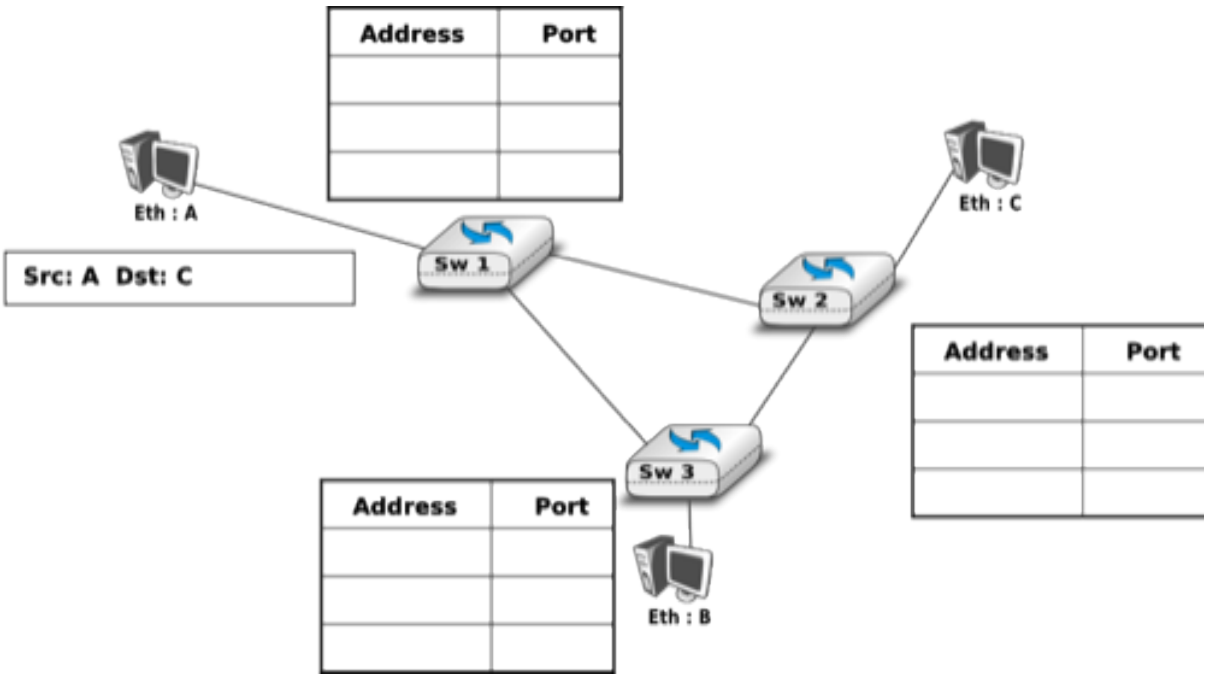


Figure 3.88: Ethernet switches in a loop

When all switches boot, their *MAC address table* is empty. Assume that host *A* sends a frame towards host *C*. Upon reception of this frame, switch1 updates its *MAC address table* to remember that address *A* is reachable via its West port. As there is no entry for address *C* in switch1's *MAC address table*, the frame is forwarded to both switch2 and switch3. When switch2 receives the frame, its updates its *MAC address table* for address *A* and forwards the frame to host *C* as well as to switch3. switch3 has thus received two copies of the same frame. As switch3 does not know how to reach the destination address, it forwards the frame received from switch1 to switch2 and the frame received from switch2 to switch1... The single frame sent by host *A* will be continuously duplicated by the switches until their *MAC address table* contains an entry for address *C*. Quickly, all the available link bandwidth will be used to forward all the copies of this frame. As Ethernet does not contain any *TTL* or *HopLimit*, this loop will never stop.

The *MAC address learning* algorithm allows switches to be plug-and-play. Unfortunately, the loops that arise when the network topology is not a tree are a severe problem. Forcing the switches to only be used in tree-shaped networks as hubs would be a severe limitation. To solve this problem, the inventors of Ethernet switches have developed the *Spanning Tree Protocol*. This protocol allows switches to automatically disable ports on Ethernet switches to ensure that the network does not contain any cycle that could cause frames to loop forever.

The Spanning Tree Protocol (802.1d)

The *Spanning Tree Protocol* (STP), proposed in [Perlman1985], is a distributed protocol that is used by switches to reduce the network topology to a spanning tree, so that there are no cycles in the topology. For example, consider the network shown in the figure below. In this figure, each bold line corresponds to an Ethernet to which two Ethernet switches are attached. This network contains several cycles that must be broken to allow Ethernet switches that are using the MAC address learning algorithm to exchange frames.

In this network, the STP will compute the following spanning tree. *Switch1* will be the root of the tree. All the interfaces of *Switch1*, *Switch2* and *Switch7* are part of the spanning tree. Only the interface connected to *LANB* will be active on *Switch9*. *LANH* will only be served by *Switch7* and the port of *Switch44* on *LANG* will be

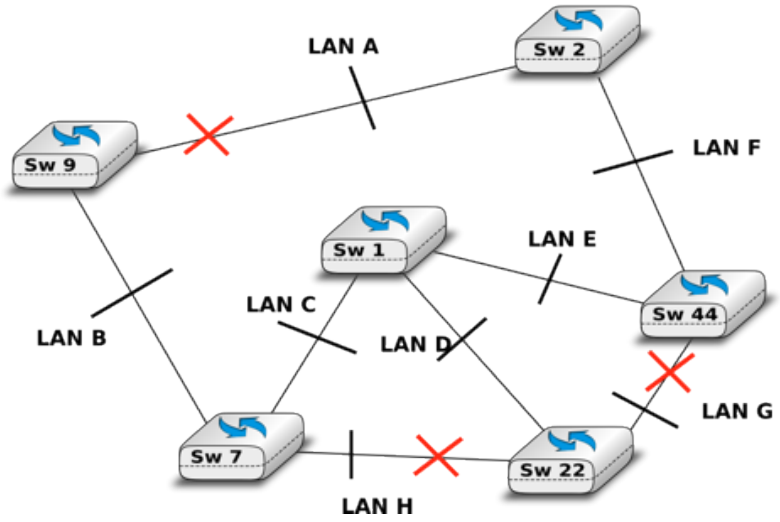


Figure 3.89: Spanning tree computed in a switched Ethernet network

disabled. A frame originating on *LANB* and destined for *LANA* will be forwarded by *Switch7* on *LANC*, then by *Switch1* on *LANE*, then by *Switch44* on *LANF* and eventually by *Switch2* on *LANA*.

Switches running the *Spanning Tree Protocol* exchange *BPDUs*. These *BPDUs* are always sent as frames with destination MAC address as the *ALL_BRIDGES* reserved multicast MAC address. Each switch has a unique 64 bit *identifier*. To ensure uniqueness, the lower 48 bits of the identifier are set to the unique MAC address allocated to the switch by its manufacturer. The high order 16 bits of the switch identifier can be configured by the network administrator to influence the topology of the spanning tree. The default value for these high order bits is 32768.

The switches exchange *BPDUs* to build the spanning tree. Intuitively, the spanning tree is built by first selecting the switch with the smallest *identifier* as the root of the tree. The branches of the spanning tree are then composed of the shortest paths that allow all of the switches that compose the network to be reached. The *BPDUs* exchanged by the switches contain the following information :

- the *identifier* of the root switch (*R*)
- the *cost* of the shortest path between the switch that sent the *BPDU* and the root switch (*c*)
- the *identifier* of the switch that sent the *BPDU* (*T*)
- the number of the switch port over which the *BPDU* was sent (*p*)

We will use the notation *<R,c,T,p>* to represent a *BPDU* whose *root identifier* is *R*, *cost* is *c* and that was sent on the port *p* of switch *T*. The construction of the spanning tree depends on an ordering relationship among the *BPDUs*. This ordering relationship could be implemented by the python function below.

```
# returns True if bpdu b1 is better than bpdu b2
def better( b1, b2) :
    return ( (b1.R < b2.R) or
             ( (b1.R==b2.R) and (b1.c<b2.c) ) or
             ( (b1.R==b2.R) and (b1.c==b2.c) and (b1.T<b2.T) ) or
             ( (b1.R==b2.R) and (b1.c==b2.c) and (b1.T==b2.T) and (b1.p<b2.p) ) )
```

In addition to the *identifier* discussed above, the network administrator can also configure a *cost* to be associated to each switch port. Usually, the *cost* of a port depends on its bandwidth and the [802.1d] standard recommends the values below. Of course, the network administrator may choose other values. We will use the notation *cost[p]* to indicate the cost associated to port *p* in this section.

Bandwidth	Cost
10 Mbps	2000000
100 Mbps	200000
1 Gbps	20000
10 Gbps	2000
100 Gbps	200

The *Spanning Tree Protocol* uses its own terminology that we illustrate in the figure above. A switch port can be in three different states : *Root*, *Designated* and *Blocked*. All the ports of the *root* switch are in the *Designated* state. The state of the ports on the other switches is determined based on the *BPDU* received on each port.

The *Spanning Tree Protocol* uses the ordering relationship to build the spanning tree. Each switch listens to *BPDUs* on its ports. When *BPDU=<R,c,T,p>* is received on port *q*, the switch computes the port's *priority vector*: *V[q]=<R,c+cost[q],T,p,q>* , where *cost[q]* is the cost associated to the port over which the *BPDU* was received. The switch stores in a table the last *priority vector* received on each port. The switch then compares its own *identifier* with the smallest *root identifier* stored in this table. If its own *identifier* is smaller, then the switch is the root of the spanning tree and is, by definition, at a distance *0* of the root. The *BPDU* of the switch is then *<R,0,R,p>*, where *R* is the switch *identifier* and *p* will be set to the port number over which the *BPDU* is sent. Otherwise, the switch chooses the best priority vector from its table, *bv=<R,c,T,p>*. The port over which this best priority vector was learned is the switch port that is closest to the *root* switch. This port becomes the *Root* port of the switch. There is only one *Root* port per switch. The switch can then compute its *BPDU* as *BPDU=<R,c,S,p>* , where *R* is the *root identifier*, *c* the cost of the best priority vector, *S* the identifier of the switch and *p* will be replaced by the number of the port over which the *BPDU* will be sent. The switch can then determine the state of all its ports by comparing its own *BPDU* with the priority vector received on each port. If the switch's *BPDU* is better than the priority vector of this port, the port becomes a *Designated* port. Otherwise, the port becomes a *Blocked* port.

The state of each port is important when considering the transmission of *BPDUs*. The root switch regularly sends its own *BPDU* over all of its (*Designated*) ports. This *BPDU* is received on the *Root* port of all the switches that are directly connected to the *root switch*. Each of these switches computes its own *BPDU* and sends this *BPDU* over all its *Designated* ports. These *BPDUs* are then received on the *Root* port of downstream switches, which then compute their own *BPDU*, etc. When the network topology is stable, switches send their own *BPDU* on all their *Designated* ports, once they receive a *BPDU* on their *Root* port. No *BPDU* is sent on a *Blocked* port. Switches listen for *BPDUs* on their *Blocked* and *Designated* ports, but no *BPDU* should be received over these ports when the topology is stable. The utilisation of the ports for both *BPDUs* and data frames is summarised in the table below.

Port state	Receives BPDUs	Sends BPDU	Handles data frames
Blocked	yes	no	no
Root	yes	no	yes
Designated	yes	yes	yes

To illustrate the operation of the *Spanning Tree Protocol*, let us consider the simple network topology in the figure below.

Assume that *Switch4* is the first to boot. It sends its own *BPDU=<4,0,4,?>* on its two ports. When *Switch1* boots, it sends *BPDU=<1,0,1,1>*. This *BPDU* is received by *Switch4*, which updates its table and computes a new *BPDU=<1,3,4,?>*. Port 1 of *Switch4* becomes the *Root* port while its second port is still in the *Designated* state.

Assume now that *Switch9* boots and immediately receives *Switch1* 's BPDU on port 1. *Switch9* computes its own *BPDU=<1,1,9,?>* and port 1 becomes the *Root* port of this switch. This *BPDU* is sent on port 2 of *Switch9* and reaches *Switch4*. *Switch4* compares the priority vector built from this *BPDU* (i.e. *<1,2,9,2>*) and notices that it is better than *Switch4* 's *BPDU=<1,3,4,2>*. Thus, port 2 becomes a *Blocked* port on *Switch4*.

During the computation of the spanning tree, switches discard all received data frames, as at that time the network topology is not guaranteed to be loop-free. Once that topology has been stable for some time, the switches again start to use the MAC learning algorithm to forward data frames. Only the *Root* and *Designated* ports are used to forward data frames. Switches discard all the data frames received on their *Blocked* ports and never forward frames on these ports.

Switches, ports and links can fail in a switched Ethernet network. When a failure occurs, the switches must be able to recompute the spanning tree to recover from the failure. The *Spanning Tree Protocol* relies on regular

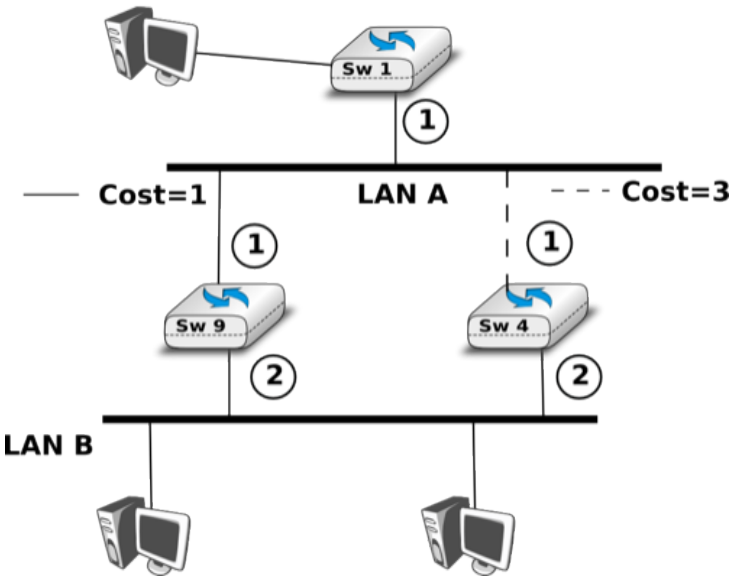


Figure 3.90: A simple Spanning tree computed in a switched Ethernet network

transmissions of the *BPDUs* to detect these failures. A *BPDU* contains two additional fields : the *Age* of the *BPDU* and the *Maximum Age*. The *Age* contains the amount of time that has passed since the root switch initially originated the *BPDU*. The root switch sends its *BPDU* with an *Age* of zero and each switch that computes its own *BPDU* increments its *Age* by one. The *Age* of the *BPDUs* stored on a switch's table is also incremented every second. A *BPDU* expires when its *Age* reaches the *Maximum Age*. When the network is stable, this does not happen as *BPDU* s are regularly sent by the *root* switch and downstream switches. However, if the *root* fails or the network becomes partitioned, *BPDU* will expire and switches will recompute their own *BPDU* and restart the *Spanning Tree Protocol*. Once a topology change has been detected, the forwarding of the data frames stops as the topology is not guaranteed to be loop-free. Additional details about the reaction to failures may be found in [802.1d]

Virtual LANs

Another important advantage of Ethernet switches is the ability to create Virtual Local Area Networks (VLANs). A virtual LAN can be defined as a *set of ports attached to one or more Ethernet switches*. A switch can support several VLANs and it runs one MAC learning algorithm for each Virtual LAN. When a switch receives a frame with an unknown or a multicast destination, it forwards it over all the ports that belong to the same Virtual LAN but not over the ports that belong to other Virtual LANs. Similarly, when a switch learns a source address on a port, it associates it to the Virtual LAN of this port and uses this information only when forwarding frames on this Virtual LAN.

The figure below illustrates a switched Ethernet network with three Virtual LANs. *VLAN2* and *VLAN3* only require a local configuration of switch *S1*. Host *C* can exchange frames with host *D*, but not with hosts that are outside of its VLAN. *VLAN1* is more complex as there are ports of this VLAN on several switches. To support such VLANs, local configuration is not sufficient anymore. When a switch receives a frame from another switch, it must be able to determine the VLAN in which the frame originated to use the correct MAC table to forward the frame. This is done by assigning an identifier to each Virtual LAN and placing this identifier inside the headers of the frames that are exchanged between switches.

IEEE defined in the [802.1q] standard a special header to encode the VLAN identifiers. This 32 bit header includes a 20 bit VLAN field that contains the VLAN identifier of each frame. The format of the [802.1q] header is described below.

The [802.1q] header is inserted immediately after the source MAC address in the Ethernet frame (i.e. before the EtherType field). The maximum frame size is increased by 4 bytes. It is encoded in 32 bits and contains four fields. The Tag Protocol Identifier is set to *0x8100* to allow the receiver to detect the presence of this additional header. The *Priority Code Point* (PCP) is a three bit field that is used to support different transmission priorities

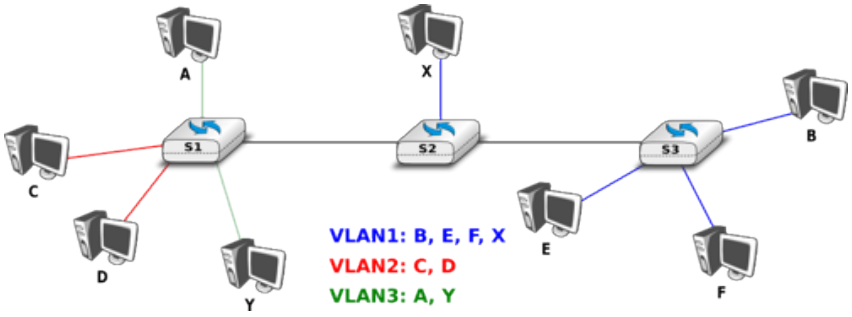


Figure 3.91: Virtual Local Area Networks in a switched Ethernet network

Figure 3.92: Format of the 802.1q header

for the frame. Value *0* is the lowest priority and value *7* the highest. Frames with a higher priority can expect to be forwarded earlier than frames having a lower priority. The *C* bit is used for compatibility between Ethernet and Token Ring networks. The last 12 bits of the 802.1q header contain the VLAN identifier. Value *0* indicates that the frame does not belong to any VLAN while value *0xFFF* is reserved. This implies that 4094 different VLAN identifiers can be used in an Ethernet network.

3.16.3 802.11 wireless networks

The radio spectrum is a limited resource that must be shared by everyone. During most of the twentieth century, governments and international organisations have regulated most of the radio spectrum. This regulation controls the utilisation of the radio spectrum, in order to ensure that there are no interferences between different users. A company that wants to use a frequency range in a given region must apply for a license from the regulator. Most regulators charge a fee for the utilisation of the radio spectrum and some governments have encouraged competition among companies bidding for the same frequency to increase the license fees.

In the 1970s, after the first experiments with ALOHANet, interest in wireless networks grew. Many experiments were done on and outside the ARPANet. One of these experiments was the first mobile phone , which was developed and tested in 1973. This experimental mobile phone was the starting point for the first generation analog mobile phones. Given the growing demand for mobile phones, it was clear that the analog mobile phone technology was not sufficient to support a large number of users. To support more users and new services, researchers in several countries worked on the development of digital mobile telephones. In 1987, several European countries decided to develop the standards for a common cellular telephone system across Europe : the *Global System for Mobile Communications* (GSM). Since then, the standards have evolved and more than three billion users are connected to GSM networks today.

While most of the frequency ranges of the radio spectrum are reserved for specific applications and require a special licence, there are a few exceptions. These exceptions are known as the Industrial, Scientific and Medical (ISM) radio bands. These bands can be used for industrial, scientific and medical applications without requiring a licence from the regulator. For example, some radio-controlled models use the 27 MHz ISM band and some cordless telephones operate in the 915 MHz ISM. In 1985, the 2.400-2.500 GHz band was added to the list of ISM bands. This frequency range corresponds to the frequencies that are emitted by microwave ovens. Sharing this band with licensed applications would have likely caused interferences, given the large number of microwave ovens that are used. Despite the risk of interferences with microwave ovens, the opening of the 2.400-2.500 GHz allowed the networking industry to develop several wireless network techniques to allow computers to exchange data without using cables. In this section, we discuss in more detail the most popular one, i.e. the WiFi [802.11] family of wireless networks. Other wireless networking techniques such as BlueTooth or HiperLAN use the same frequency range.

Today, WiFi is a very popular wireless networking technology. There are more than several hundreds of millions

of WiFi devices. The development of this technology started in the late 1980s with the WaveLAN proprietary wireless network. WaveLAN operated at 2 Mbps and used different frequency bands in different regions of the world. In the early 1990s, the IEEE created the 802.11 working group to standardise a family of wireless network technologies. This working group was very prolific and produced several wireless networking standards that use different frequency ranges and different physical layers. The table below provides a summary of the main 802.11 standards.

Standard	Frequency	Typical throughput	Max bandwidth	Range (m) indoor/outdoor
802.11	2.4 GHz	0.9 Mbps	2 Mbps	20/100
802.11a	5 GHz	23 Mbps	54 Mbps	35/120
802.11b	2.4 GHz	4.3 Mbps	11 Mbps	38/140
802.11g	2.4 GHz	19 Mbps	54 Mbps	38/140
802.11n	2.4/5 GHz	74 Mbps	150 Mbps	70/250

When developing its family of standards, the IEEE 802.11 working group took a similar approach as the IEEE 802.3 working group that developed various types of physical layers for Ethernet networks. 802.11 networks use the CSMA/CA Medium Access Control technique described earlier and they all assume the same architecture and use the same frame format.

The architecture of WiFi networks is slightly different from the Local Area Networks that we have discussed until now. There are, in practice, two main types of WiFi networks : *independent* or *adhoc* networks and *infrastructure* networks [66]. An *independent* or *adhoc* network is composed of a set of devices that communicate with each other. These devices play the same role and the *adhoc* network is usually not connected to the global Internet. *Adhoc* networks are used when for example a few laptops need to exchange information or to connect a computer with a WiFi printer.

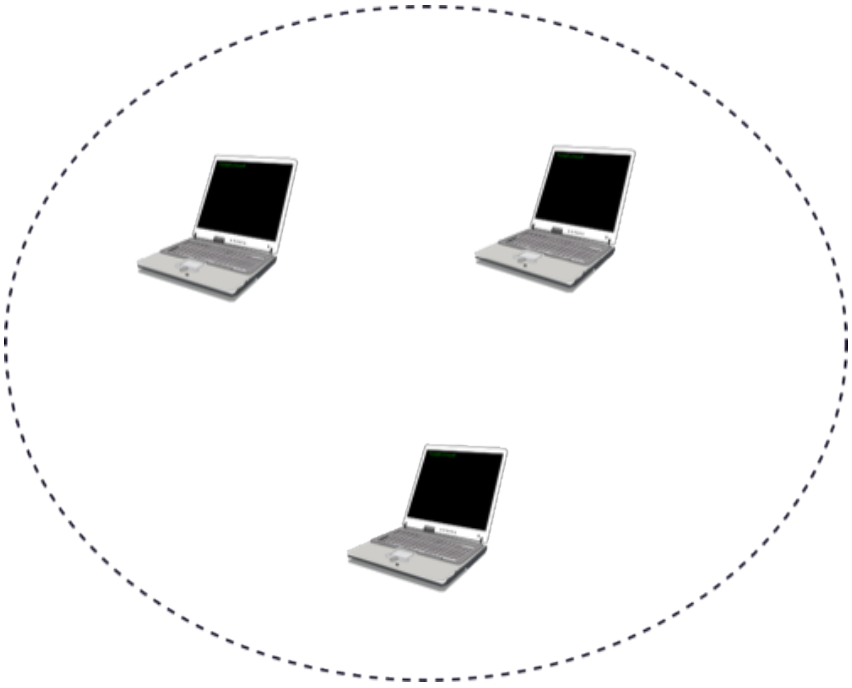


Figure 3.93: An 802.11 independent or adhoc network

Most WiFi networks are *infrastructure* networks. An *infrastructure* network contains one or more *access points* that are attached to a fixed Local Area Network (usually an Ethernet network) that is connected to other networks such as the Internet. The figure below shows such a network with two access points and four WiFi devices. Each WiFi device is associated to one access point and uses this access point as a relay to exchange frames with the devices that are associated to another access point or reachable through the LAN.

An 802.11 access point is a relay that operates in the datalink layer like switches. The figure below represents the

[66] The 802.11 working group defined the *basic service set (BSS)* as a group of devices that communicate with each other. We continue to use *network* when referring to a set of devices that communicate.

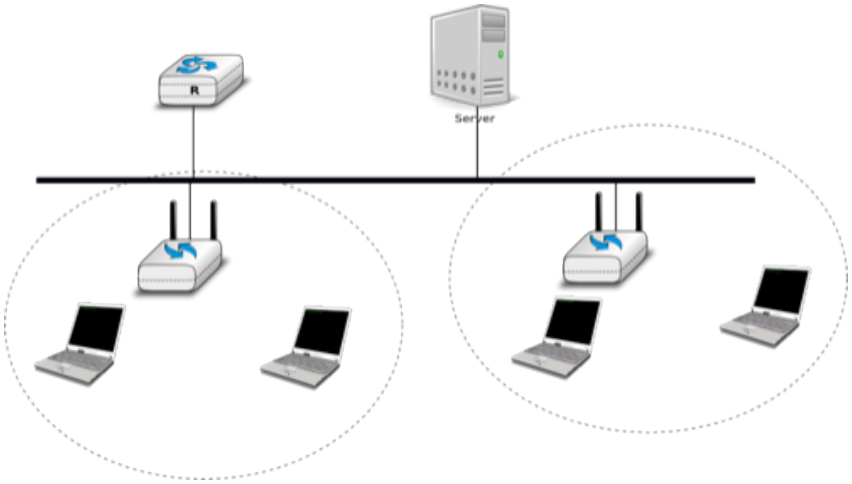


Figure 3.94: An 802.11 infrastructure network

layers of the reference model that are involved when a WiFi host communicates with a host attached to an Ethernet network through an access point.

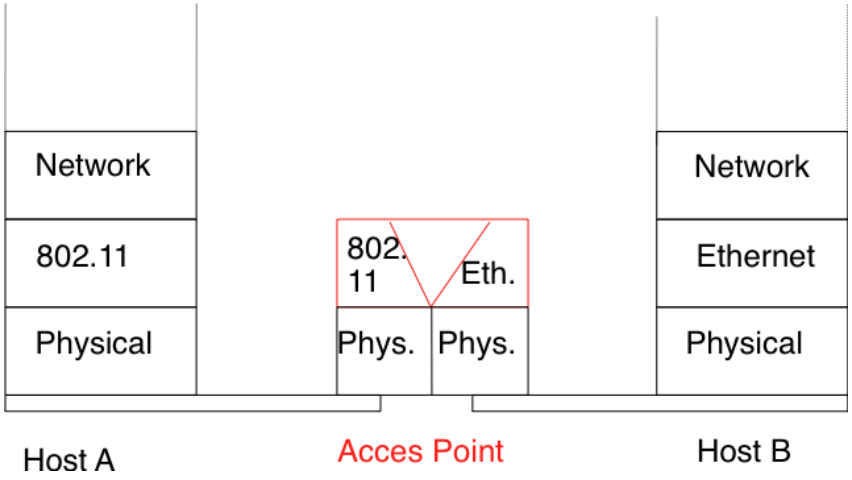


Figure 3.95: An 802.11 access point

802.11 devices exchange variable length frames, which have a slightly different structure than the simple frame format used in Ethernet LANs. We review the key parts of the 802.11 frames. Additional details may be found in [802.11] and [Gast2002] . An 802.11 frame contains a fixed length header, a variable length payload that may contain up 2324 bytes of user data and a 32 bits CRC. Although the payload can contain up to 2324 bytes, most 802.11 deployments use a maximum payload size of 1500 bytes as they are used in *infrastructure* networks attached to Ethernet LANs. An 802.11 data frame is shown below.

The first part of the 802.11 header is the 16 bit *Frame Control* field. This field contains flags that indicate the type of frame (data frame, RTS/CTS, acknowledgement, management frames, etc), whether the frame is sent to or from a fixed LAN, etc [802.11]. The *Duration* is a 16 bit field that is used to reserve the transmission channel. In data frames, the *Duration* field is usually set to the time required to transmit one acknowledgement frame after a SIFS delay. Note that the *Duration* field must be set to zero in multicast and broadcast frames. As these frames are not acknowledged, there is no need to reserve the transmission channel after their transmission. The *Sequence control* field contains a 12 bits sequence number that is incremented for each data frame.

The astute reader may have noticed that the 802.11 data frames contain three 48-bits address fields [67] . This is surprising compared to other protocols in the network and datalink layers whose headers only contain a source and

[67] In fact, the [802.11] frame format contains a fourth optional address field. This fourth address is only used when an 802.11 wireless network is used to interconnect bridges attached to two classical LAN networks.

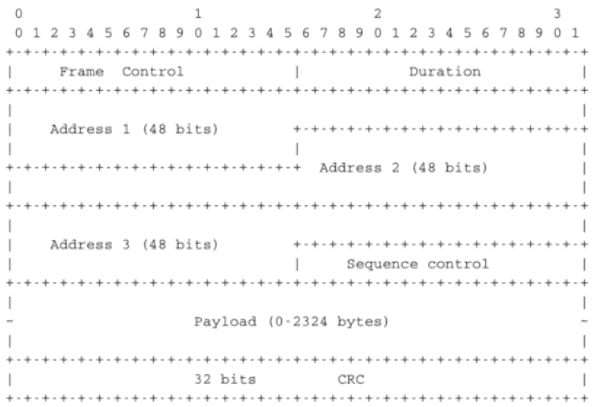


Figure 3.96: 802.11 data frame format

a destination address. The need for a third address in the 802.11 header comes from the *infrastructure* networks. In such a network, frames are usually exchanged between routers and servers attached to the LAN and WiFi devices attached to one of the access points. The role of the three address fields is specified by bit flags in the *Frame Control* field.

When a frame is sent from a WiFi device to a server attached to the same LAN as the access point, the first address of the frame is set to the MAC address of the access point, the second address is set to the MAC address of the source WiFi device and the third address is the address of the final destination on the LAN. When the server replies, it sends an Ethernet frame whose source address is its MAC address and the destination address is the MAC address of the WiFi device. This frame is captured by the access point that converts the Ethernet header into an 802.11 frame header. The 802.11 frame sent by the access point contains three addresses : the first address is the MAC address of the destination WiFi device, the second address is the MAC address of the access point and the third address the MAC address of the server that sent the frame.

802.11 control frames are simpler than data frames. They contain a *Frame Control*, a *Duration* field and one or two addresses. The acknowledgement frames are very small. They only contain the address of the destination of the acknowledgement. There is no source address and no *Sequence Control* field in the acknowledgement frames. This is because the acknowledgement frame can easily be associated to the previous frame that it acknowledges. Indeed, each unicast data frame contains a *Duration* field that is used to reserve the transmission channel to ensure that no collision will affect the acknowledgement frame. The *Sequence Control* field is mainly used by the receiver to remove duplicate frames. Duplicate frames are detected as follows. Each data frame contains a 12 bits *Sequence Control* field and the *Frame Control* field contains the *Retry* bit flag that is set when a frame is transmitted. Each 802.11 receiver stores the most recent sequence number received from each source address in frames whose *Retry* bit is reset. Upon reception of a frame with the *Retry* bit set, the receiver verifies its sequence number to determine whether it is a duplicated frame or not.

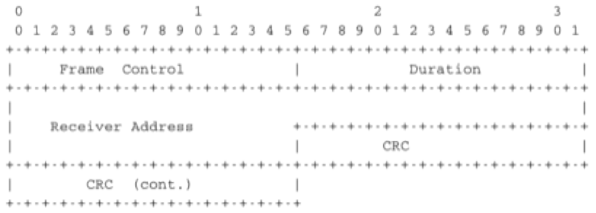


Figure 3.97: IEEE 802.11 ACK and CTS frames

802.11 RTS/CTS frames are used to reserve the transmission channel, in order to transmit one data frame and its acknowledgement. The RTS frames contain a *Duration* and the transmitter and receiver addresses. The *Duration* field of the RTS frame indicates the duration of the entire reservation (i.e. the time required to transmit the CTS, the data frame, the acknowledgements and the required SIFS delays). The CTS frame has the same format as the acknowledgement frame.

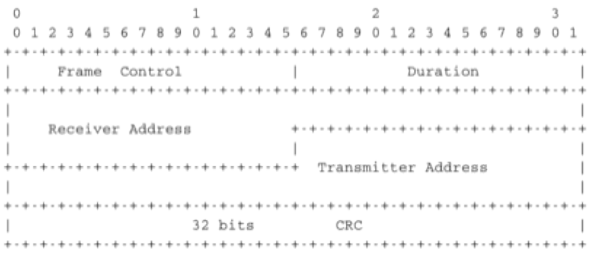


Figure 3.98: IEEE 802.11 RTS frame format

Note: The 802.11 service

Despite the utilization of acknowledgements, the 802.11 layer only provides an unreliable connectionless service like Ethernet networks that do not use acknowledgements. The 802.11 acknowledgements are used to minimize the probability of frame duplication. They do not guarantee that all frames will be correctly received by their recipients. Like Ethernet, 802.11 networks provide a high probability of successful delivery of the frames, not a guarantee. Furthermore, it should be noted that 802.11 networks do not use acknowledgements for multicast and broadcast frames. This implies that in practice such frames are more likely to suffer from transmission errors than unicast frames.

In addition to the data and control frames that we have briefly described above, 802.11 networks use several types of management frames. These management frames are used for various purposes. We briefly describe some of these frames below. A detailed discussion may be found in [802.11] and [Gast2002].

A first type of management frames are the *beacon* frames. These frames are broadcasted regularly by access points. Each *beacon frame* contains information about the capabilities of the access point (e.g. the supported 802.11 transmission rates) and a *Service Set Identity* (SSID). The SSID is a null-terminated ASCII string that can contain up to 32 characters. An access point may support several SSIDs and announce them in beacon frames. An access point may also choose to remain silent and not advertise beacon frames. In this case, WiFi stations may send *Probe request* frames to force the available access points to return a *Probe response* frame.

Note: IP over 802.11

Two types of encapsulation schemes were defined to support IP in Ethernet networks : the original encapsulation scheme, built above the Ethernet DIX format is defined in **RFC 894** and a second encapsulation **RFC 1042** scheme, built above the LLC/SNAP protocol [802.2]. In 802.11 networks, the situation is simpler and only the **RFC 1042** encapsulation is used. In practice, this encapsulation adds 6 bytes to the 802.11 header. The first four bytes correspond to the LLC/SNAP header. They are followed by the two bytes Ethernet Type field (*0x800* for IP and *0x806* for ARP). The figure below shows an IP packet encapsulated in an 802.11 frame.

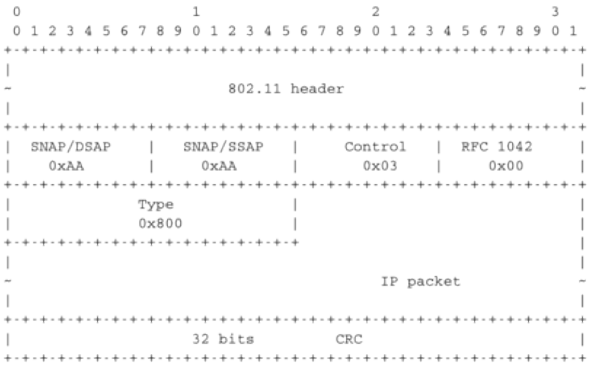


Figure 3.99: IP over IEEE 802.11

The second important utilisation of the management frames is to allow a WiFi station to be associated with an

access point. When a WiFi station starts, it listens to beacon frames to find the available SSIDs. To be allowed to send and receive frames via an access point, a WiFi station must be associated to this access point. If the access point does not use any security mechanism to secure the wireless transmission, the WiFi station simply sends an *Association request* frame to its preferred access point (usually the access point that it receives with the strongest radio signal). This frame contains some parameters chosen by the WiFi station and the SSID that it requests to join. The access point replies with an *Association response frame* if it accepts the WiFI station.

Part 3: Practice

4.1 Reliable transfer

Multiple Choice Questions

4.1.1 Alternating Bit Protocol

The Alternating Bit Protocol is the simplest reliable protocol. The following questions should allow you to get a better understanding of the operation of this protocol.

1. Consider the Alternating Bit Protocol as described in the first chapter.

- How does the protocol recover from the loss of a data segment ?
- How does the protocol recovers from the loss of an acknowledgement ?

2. How would you set the retransmission timer if you were implementing the Alternating Bit Protocol to exchange files with a server such as the one that you measured above ?
3. What are the factors that affect the performance of the Alternating Bit Protocol ?
4. A version of the Alternating Bit Protocol supporting variable length frames uses a header that contains the following fields :
    - a *number* (0 or 1)
    - a *length* field that indicates the length of the data
    - a *CRC*

    To speedup the transmission of the frames, a student proposes to compute the CRC over the data part of the segment but not over the header. What do you think of this optimisation ?
5. A student proposed to optimise the Alternating Bit Protocol by adding a negative acknowledgment, i.e. the receiver sends a *NAK* control segment when it receives a corrupted data segment. What kind of information should be placed in this control segment and how should the sender react when receiving such a *NAK* ?
6. Derive a mathematical expression that provides the *goodput*, i.e. the amount of payload bytes that have been transmitted during a period of time, achieved by the Alternating Bit Protocol assuming that :

- Each frame contains D bytes of data and c bytes of control information
- Each acknowledgement contains c bytes of control information
- The bandwidth of the two directions of the link is set to B bits per second
- The delay between the two hosts is s seconds in both directions
- there are no transmission errors

7. Links are often considered as symmetrical, i.e. they offer the same bandwidth in both directions. Symmetrical links are widely used in Local Area Networks and in the core of the Internet, but there are many asymmetrical link technologies. The most common example are the various types of ADSL and CATV technologies. Consider an implementation of the Alternating Bit Protocol that is used between two hosts that are directly connected by using an asymmetric link. Assume that a host is sending frames containing 10 bytes of control information and 90 bytes of data and that the acknowledgements are 10 bytes long. If the link delay is negligible, what is the minimum bandwidth required on the return link to ensure that the transmission of acknowledgements is not a bottleneck ?
8. Consider an Alternating Bit Protocol that is used over a link that suffers from deterministic errors. When the error ratio is set to $\frac{1}{p}$, this means that $p-1$ bits are transmitted correctly and the p^{th} bit is corrupted. Discuss the factors that affect the performance of the Alternating Bit Protocol over such a link.

4.1.2 Go-back-n and selective repeat

Most reliable protocols use a window and either go-back-n or selective repeat as retransmission strategy.

1. What are the techniques used by a go-back-n sender to recover from :

- transmission errors
- losses of data segments
- losses of acknowledgements

2. Consider a *b* bits per second link between two hosts that has a propagation delay of *t* seconds. Derive a formula that computes the time elapsed between the transmission of the first bit of a *d* bytes segment from a sending host and the reception of the last bit of this segment on the receiving host.
3. Consider a go-back-n sender and a go-back receiver that are directly connected with a 10 Mbps link that has a propagation delay of 100 milliseconds. Assume that the retransmission timer is set to three seconds. If the window has a length of 4 segments, draw a time-sequence diagram showing the transmission of 10 segments (each segment contains 10000 bits):

- when there are no losses
- when the third and seventh segments are lost
- when every second acknowledgement is discarded due to transmission errors

4. Same question when using selective repeat instead of go-back-n. Note that the answer is not necessarily the same.
5. Consider two high-end servers connected back-to-back by using a 10 Gbps interface. If the delay between the two servers is one millisecond, what is the throughput that can be achieved by a reliable protocol that is using 10,000 bits frames and a window of

- one frame
- ten frames
- hundred frames

6. Consider two servers are directly connected by using a *b* bits per second link with a round-trip-time of *r* seconds. The two servers are using a transport protocol that sends segments containing *s* bytes and acknowledgements composed of *a* bytes. Can you derive a formula that computes the smallest window (measured in segments) that is required to ensure that the servers will be able to completely utilise the link ?
7. Is it possible for a go-back-n receiver to inter-operate with a selective-repeat sender ? Justify your answer.
8. Is it possible for a selective-repeat receiver to inter-operate with a go-back-n sender ? Justify your answer.
9. The go-back-n and selective repeat mechanisms that are described in the book exclusively rely on cumulative acknowledgements. This implies that a receiver always returns to the sender information about the last frame that was received in-sequence. If there are frequent losses, a selective repeat receiver could return several times the same cumulative acknowledgment. Can you think of other types of acknowledgements that could be used by a selective repeat receiver to provide additional information about the out-of-sequence frames

that it has received. Design such acknowledgements and explain how the sender should react upon reception of this information.

10. A go-back-n receiver has sent a full window of data segments. All the segments have been received correctly and in-order by the receiver, but all the returned acknowledgements have been lost. Show by using a time sequence diagram (e.g. by considering a window of four segments) what happens in this case. Can you fix the problem on the go-back-n sender ?

11. Same question as above, but assume now that both the sender and the receiver implement selective repeat. Note that the answer will be different from the above question.

12. Consider a protocol that supports window of one hundred 1250 Bytes segments. What is the maximum bandwidth that this protocol can achieve if the round-trip-time is set to one second ? What happens if, instead of advertising a window of one hundred frames, the receiver decides to advertise a window of 10 frames ?

4.1.3 Error detection

Reliable protocols depend on error detection algorithms to detect transmission errors. The following questions will reinforce your understanding of these algorithms.

1. Reliable protocols rely on different types of checksums to verify whether frames have been affected by transmission errors. The most frequently used checksums are :

   - the Internet checksum used by UDP, TCP and other Internet protocols which is defined in **RFC 1071** and implemented in various modules, e.g. http://ilab.cs.byu.edu/cs460/code/ftp/ichecksum.py for a python implementation
   - the 16 bits or the 32 bits Cyclical Redundancy Checks (CRC) that are often used on disks, in zip archives and in datalink layer protocols. See http://docs.python.org/library/binascii.html for a python module that contains the 32 bits CRC
   - the Fletcher checksum [Fletcher1982], see http://drdobbs.com/database/184408761 for implementation details

   By using your knowledge of the Internet checksum, can you find a transmission error that will not be detected by this checksum ?

2. The CRCs are efficient error detection codes that are able to detect :

   - all errors that affect an odd number of bits
   - all errors that affect a sequence of bits which is shorter than the length of the CRC

   Carry experiments with one implementation of CRC-32 to verify that this is indeed the case.

3. Checksums and CRCs should not be confused with secure hash functions such as MD5 defined in **RFC 1321** or SHA-1 described in **RFC 4634**. Secure hash functions are used to ensure that files or sometimes packets/segments have not been modified. Secure hash functions aim at detecting malicious changes while checksums and CRCs only detect random transmission errors. Perform some experiments with hash functions such as those defined in the http://docs.python.org/library/hashlib.html python hashlib module to verify that this is indeed the case.

4.2 Building a network

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=2

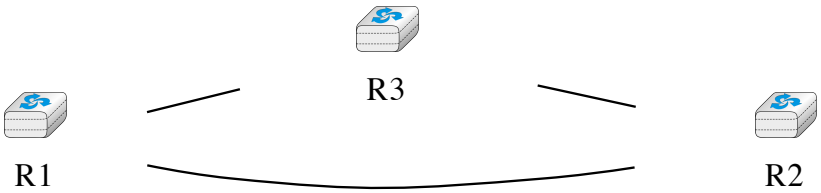
Multiple choices questions

4.2.1 Building forwarding tables

1. Consider the network shown in the figure below.

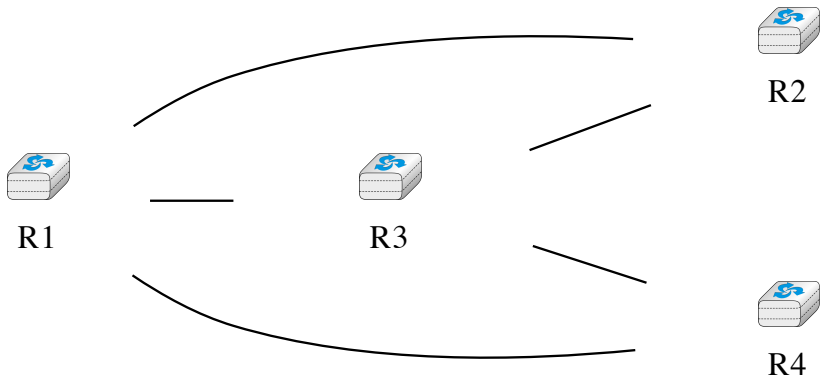
Assume that the tree nodes are using port-forwarding tables. List in details all the packets that are exchanged when *R3* sends a packet to *R2*. *R2* replies to this packet. Explain all the packets that are transmitted in this network for this exchange.

2. Consider the network shown in the figure below.



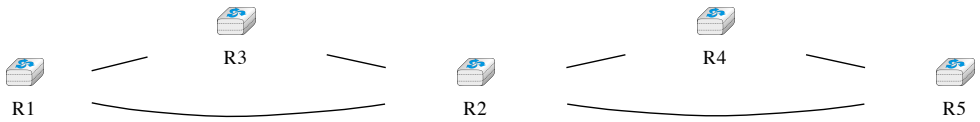
Assume that the tree nodes are using port-forwarding tables. List in details all the packets that are exchanged when *R3* sends a packet to *R2*. *R2* replies to this packet. Explain all the packets that are transmitted in this network for this exchange.

3. Consider the network shown in the figure below.



Assuming that the network uses source routing, what are the possible paths from *R1* to *R4* ?

4. Consider the network shown in the figure below.

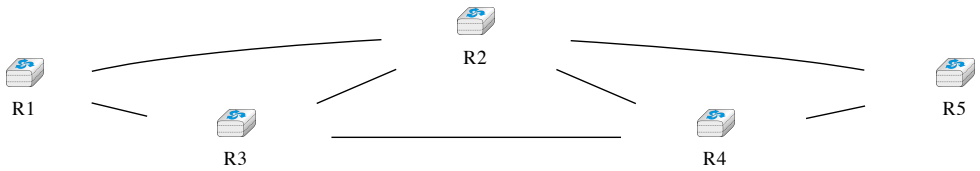


The network operator uses would like to have the following paths in this network :

- *R3->R2->R4->R5* and *R1->R2->R5*

Is it possible to achieve these paths and if so what are the required forwarding tables ?

5. Consider the network shown in the figure below.



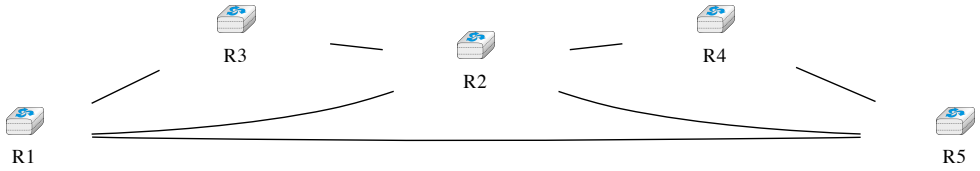
The network operator would like to use the following paths :

- *R1->R2->R4* and *R3->R2->R5->R4*

Are these paths possible with link-state or distance vector routing ? If yes, how do configure the link weights. If no, explain your answer.

Same question with label switching.

6. Consider the network shown in the figure below.



The network operator would like to use the following paths :

- *R1->R5->R4* and *R3->R2->R4*

Are these paths possible with link-state or distance vector routing ? If yes, how do configure the link weights. If no, explain your answer.

4.2.2 Routing protocols

1. Routing protocols used in data networks only use positive link weights. What would happen with a distance vector routing protocol in the network below that contains a negative link weight ?

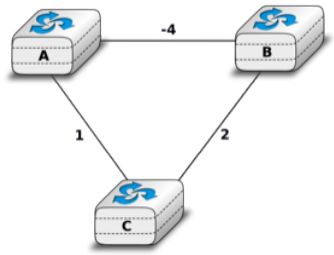


Figure 4.1: A simple network

2. When a network specialist designs a network, one of the problems that he needs to solve is to set the metrics the links in his network. In the USA, the Abilene network interconnects most of the research labs and universities. The figure below shows the topology of this network in 2009.

Figure 4.2: The Abilene network

In this network, assume that all the link weights are set to 1. What is the paths followed by a packet sent by the router located in *Los Angeles* to reach :

- the router located in *New York*
- the router located in *Washington* ?

Is it possible to configure the link metrics so that the packets sent by the router located in *Los Angeles* to the routers located in respectively *New York* and *Washington* do not follow the same path ?

Is it possible to configure the link weights so that the packets sent by the router located in *Los Angeles* to router located in *New York* follow one path while the packets sent by the router located in *New York* to the router located in *Los Angeles* follow a completely different path ?

Assume that the routers located in *Denver* and *Kansas City* need to exchange lots of packets. Can you configure the link metrics such that the link between these two routers does not carry any packet sent by another router in the network ?

3. In the five nodes network shown below, can you configure the link metrics so that the packets sent by router *E* to router *A* use link *B->A* while the packets sent by router *B* use links *B->D* and *D->A*?

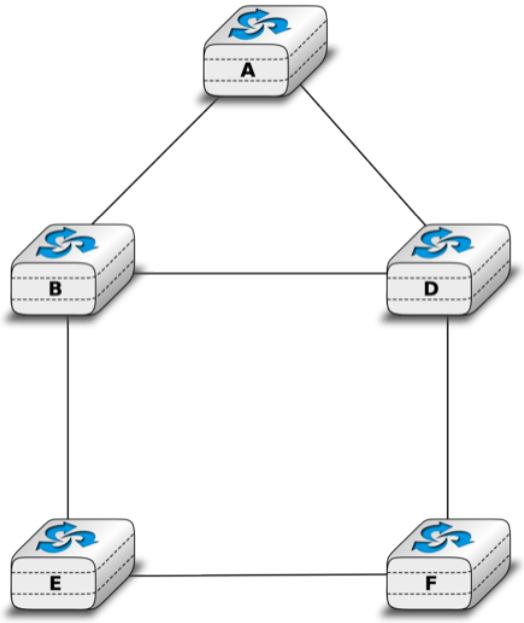


Figure 4.3: Simple five nodes network

4. In the five nodes network shown above, can you configure the link weights so that the packets sent by router *E* (resp. *F*) follow the *E->B->A* path (resp. *F->D->B->A*) ?
5. In the above questions, you have worked on the stable state of the routing tables computed by routing protocols. Let us now consider the transient problems that mainly happen when the network topology changes. For this, consider the network topology shown in the figure below and assume that all routers use a distance vector protocol that uses split horizon.

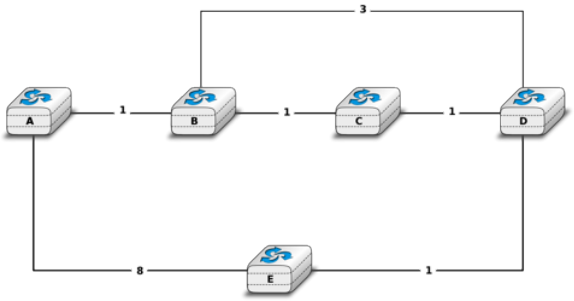


Figure 4.4: Simple network with redundant links

If you compute the routing tables of all routers in this network, you would obtain a table such as the table below :

Destination	Routes on A	Routes on B	Routes on C	Routes on D	Routes on E
A	0	1 via A	2 via B	3 via C	4 via D
B	1 via B	0	1 via B	2 via C	3 via D
C	2 via B	1 via C	0	1 via C	2 via D
D	3 via B	2 via C	1 via D	0	1 via D
E	4 via B	3 via C	2 via D	1 via E	0

Distance vector protocols can operate in two different modes : *periodic updates* and *triggered updates*. *Periodic updates* is the default mode for a distance vector protocol. For example, each router could advertise its distance vector every thirty seconds. With the *triggered updates* a router sends its distance vector when its routing table changes (and periodically when there are no changes).

- Consider a distance vector protocol using split horizon and *periodic updates*. Assume that the link *B*-*C* fails. *B* and *C* update their local routing table but they will only advertise it at the end of their period. Select one ordering for the *periodic updates* and every time a router sends its distance vector, indicate the vector sent to each neighbor and update the table above. How many periods are required to allow the network to converge to a stable state ?
- Consider the same distance vector protocol, but now with *triggered updates*. When link *B*-*C* fails, assume that *B* updates its routing table immediately and sends its distance vector to *A* and *D*. Assume that both *A* and *D* process the received distance vector and that *A* sends its own distance vector, ... Indicate all the distance vectors that are exchanged and update the table above each time a distance vector is sent by a router (and received by other routers) until all routers have learned a new route to each destination. How many distance vector messages must be exchanged until the network converges to a stable state ?

6. Consider the network shown below. In this network, the metric of each link is set to *1* except link *A*-*B* whose metric is set to *4* in both directions. In this network, there are two paths with the same cost between *D* and *C*. Old routers would randomly select one of these equal cost paths and install it in their forwarding table. Recent routers are able to use up to *N* equal cost paths towards the same destination.

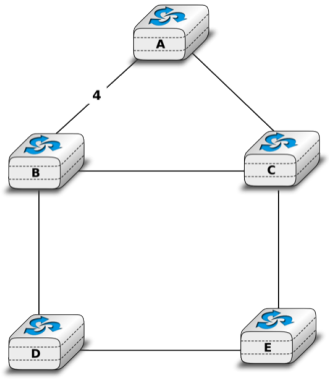


Figure 4.5: A simple network

On recent routers, a lookup in the forwarding table for a destination address returns a set of outgoing interfaces. How would you design an algorithm that selects the outgoing interface used for each packet, knowing that to avoid reordering, all segments of a given TCP connection should follow the same path ?

7. Consider again the network shown above. After some time, link state routing converges and all routers compute the following routing tables :

Destination	Routes on A	Routes on B	Routes on C	Routes on D	Routes on E
A	0	2 via C	1 via A	3 via B,E	2 via C
B	2 via C	0	1 via B	1 via B	2 via D,C
C	1 via C	1 via C	0	2 via B,E	1 via C
D	3 via C	1 via D	2 via B,E	0	1 via D
E	2 via C	2 via C,D	1 via E	1 via E	0

An important difference between OSPF and RIP is that OSPF routers flood link state packets that allow the other routers to recompute their own routing tables while RIP routers exchange distance vectors. Consider that link *B*-*C* fails and that router *B* is the first to detect the failure. At this point, *B* cannot reach anymore *A*, *C* and 50% of its paths towards *E* have failed. *C* cannot reach *B* anymore and half of its paths towards *D* have failed.

Router *B* will flood its updated link state packet through the entire network and all routers will recompute their forwarding table. Upon reception of a link state packet, routers usually first flood the received link-state packet and then recompute their forwarding table. Assume that *B* is the first to recompute its forwarding table, followed by *D*, *A*, *C* and finally *E*

8. After each update of a forwarding table, verify which pairs of routers are able to exchange packets. Provide your answer using a table similar to the one shown above.
9. Can you find an ordering of the updates of the forwarding tables that avoids all transient problems ?

4.3 Serving applications

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=3

4.3.1 Writing simple networked applications

Networked applications were usually implemented by using the *socket API*. This API was designed when TCP/IP was first implemented in the Unix BSD operating system [Sechrest] [LFJLMT], and has served as the model for many APIs between applications and the networking stack in an operating system. Although the socket API is very popular, other APIs have also been developed. For example, the STREAMS API has been added to several Unix System V variants [Rago1993]. The socket API is supported by most programming languages and several textbooks have been devoted to it. Users of the C language can consult [DC2009], [Stevens1998], [SFR2004] or [Kerrisk2010]. The Java implementation of the socket API is described in [CD2008] and in the Java tutorial. In this section, we will use the python implementation of the socket API to illustrate the key concepts. Additional information about this API may be found in the socket section of the python documentation .

The socket API is quite low-level and should be used only when you need a complete control of the network access. If your application simply needs, for instance, to retrieve data from a web server, there are much simpler and higher-level APIs.

A detailed discussion of the socket API is outside the scope of this section and the references cited above provide a detailed discussion of all the details of the socket API. As a starting point, it is interesting to compare the socket API with the service primitives that we have discussed in the previous chapter. Let us first consider the connectionless service that consists of the following two primitives :

- *DATA.request(destination,message)* is used to send a message to a specified destination. In this socket API, this corresponds to the `send` method.
- *DATA.indication(message)* is issued by the transport service to deliver a message to the application. In the socket API, this corresponds to the return of the `recv` method that is called by the application.

The *DATA* primitives are exchanged through a service access point. In the socket API, the equivalent to the service access point is the *socket*. A *socket* is a data structure which is maintained by the networking stack and is used by the application every time it needs to send or receive data through the networking stack. The *socket* method in the python API takes two main arguments :

- an *address family* that specifies the type of address family and thus the underlying networking stack that will be used with the socket. This parameter can be either `socket.AF_INET` or `socket.AF_INET6`. `socket.AF_INET`, which corresponds to the TCP/IPv4 protocol stack is the default. `socket.AF_INET6` corresponds to the TCP/IPv6 protocol stack.
- a *type* indicates the type of service which is expected from the networking stack. `socket.STREAM` (the default) corresponds to the reliable bytestream connection-oriented service. `socket.DGRAM` corresponds to the connectionless service.

A simple client that sends a request to a server is often written as follows in descriptions of the socket API.

```
import socket
import sys
HOSTIP=sys.argv[1]
PORT=int(sys.argv[2])
MSG="Hello, World!"
s = socket.socket( socket.AF_INET6, socket.SOCK_DGRAM )
s.sendto( MSG, (HOSTIP, PORT,0,0) )
```

A typical usage of this application would be

```
python client.py ::1 12345
```

where `::1` is the IPv6 address of the host (in this case the localhost) where the server is running and `12345` the port of the server.

The first operation is the creation of the `socket`. Two parameters must be specified while creating a `socket`. The first parameter indicates the address family and the second the socket type. The second operation is the transmission of the message by using `sendto` to the server. It should be noted that `sendto` takes as arguments the message to be transmitted and a tuple that contains the IPv6 address of the server and its port number.

The code shown above supports only the TCP/IPv6 protocol stack. To use the TCP/IPv4 protocol stack the `socket` must be created by using the `socket.AF_INET` address family. Forcing the application developer to select TCP/IPv4 or TCP/IPv6 when creating a `socket` is a major hurdle for the deployment and usage of TCP/IPv6 in the global Internet [Cheshire2010]. While most operating systems support both TCP/IPv4 and TCP/IPv6, many applications still only use TCP/IPv4 by default. In the long term, the `socket` API should be able to handle TCP/IPv4 and TCP/IPv6 transparently and should not force the application developer to always specify whether it uses TCP/IPv4 or TCP/IPv6.

Another important issue with the socket API as supported by python is that it forces the application to deal with IP addresses instead of dealing directly with domain names. This limitation dates from the early days of the `socket` API in Unix 4.2BSD. At that time, name resolution was not widely available and only IP addresses could be used. Most applications rely on DNS names to interact with servers and this utilisation of the DNS plays a very important role to scale web servers and content distribution networks. To use domain names, the application needs to perform the DNS resolution by using the `getaddrinfo` method. This method queries the DNS and builds the `sockaddr` data structure which is used by other methods of the socket API. In python, `getaddrinfo` takes several arguments :

- a *name* that is the domain name for which the DNS will be queried
- an optional *port number* which is the port number of the remote server
- an optional *address family* which indicates the address family used for the DNS request. `socket.AF_INET` (resp. `socket.AF_INET6`) indicates that an IPv4 (IPv6) address is expected. Furthermore, the python socket API allows an application to use `socket.AF_UNSPEC` to indicate that it is able to use either IPv4 or IPv6 addresses.
- an optional *socket type* which can be either `socket.SOCK_DGRAM` or `socket.SOCK_STREAM`

In today's Internet hosts that are capable of supporting both IPv4 and IPv6, all applications should be able to handle both IPv4 and IPv6 addresses. When used with the `socket.AF_UNSPEC` parameter, the `socket.getaddrinfo` method returns a list of tuples containing all the information to create a `socket`.

```
import socket
socket.getaddrinfo('www.example.net',80,socket.AF_UNSPEC,socket.SOCK_STREAM)
```

```
[ (30, 1, 6, '', ('2001:db8:3080:3::2', 80, 0, 0)),
  (2, 1, 6, '', ('203.0.113.225', 80))]
```

In the example above, `socket.getaddrinfo` returns two tuples. The first one corresponds to the `sockaddr` containing the IPv6 address of the remote server and the second corresponds to the IPv4 information. Due to some peculiarities of IPv6 and IPv4, the format of the two tuples is not exactly the same, but the key information in both cases are the network layer address (`2001:db8:3080:3::2` and `203.0.113.225`) and the port number (`80`). The other parameters are seldom used.

`socket.getaddrinfo` can be used to build a simple client that queries the DNS and contact the server by using either IPv4 or IPv6 depending on the addresses returned by the `socket.getaddrinfo` method. The client below iterates over the list of addresses returned by the DNS and sends its request to the first destination address for which it can create a `socket`. Other strategies are of course possible. For example, a host running in an IPv6 network might prefer to always use IPv6 when IPv6 is available [1].

```
import socket
import sys
HOSTNAME=sys.argv[1]
PORT=int(sys.argv[2])
MSG="Hello, World!"
for a in socket.getaddrinfo(HOSTNAME, PORT, socket.AF_UNSPEC,socket.SOCK_DGRAM,0, socket.AI_PASSI
    address_family,sock_type,protocol,canonicalname, sockaddr=a
    try:
        s = socket.socket(address_family, sock_type)
    except socket.error:
        s = None
        print "Could not create socket"
        continue
    if s is not None:
        s.sendto(MSG, sockaddr)
        break
```

Now that we have described the utilisation of the socket API to write a simple client using the connectionless transport service, let us have a closer look at the reliable byte stream transport service. As explained above, this service is invoked by creating a `socket` of type `socket.SOCK_STREAM`. Once a socket has been created, a client will typically connect to the remote server, send some data, wait for an answer and eventually close the connection. These operations are performed by calling the following methods :

- `socket.connect` : this method takes a `sockaddr` data structure, typically returned by `socket.getaddrinfo`, as argument. It may fail and raise an exception if the remote server cannot be reached.
- `socket.send` : this method takes a string as argument and returns the number of bytes that were actually sent. The string will be transmitted as a sequence of consecutive bytes to the remote server. Applications are expected to check the value returned by this method and should resend the bytes that were not send.
- `socket.recv` : this method takes an integer as argument that indicates the size of the buffer that has been allocated to receive the data. An important point to note about the utilisation of the `socket.recv` method is that as it runs above a bytestream service, it may return any amount of bytes (up to the size of the buffer provided by the application). The application needs to collect all the received data and there is no guarantee that some data sent by the remote host by using a single call to the `socket.send` method will be received by the destination with a single call to the `socket.recv` method.
- `socket.shutdown` : this method is used to release the underlying connection. On some platforms, it is possible to specify the direction of transfer to be released (e.g. `socket.SHUT_WR` to release the outgoing direction or `socket.SHUT_RDWR` to release both directions).
- `socket.close`: this method is used to close the socket. It calls `socket.shutdown` if the underlying connection is still open.

[1] Most operating systems today by default prefer to use IPv6 when the DNS returns both an IPv4 and an IPv6 address for a name. See http://ipv6int.net/systems/ for more detailed information.

With these methods, it is now possible to write a simple HTTP client. This client operates over both IPv6 and IPv4 and writes the main page of the remote server on the standard output. It also reports the number of `socket.recv` calls that were used to retrieve the homepage [2] .

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site

import socket, sys

if len(sys.argv)!=3 and len(sys.argv)!=2:
    print "Usage : ",sys.argv[0]," hostname [port]"

hostname = sys.argv[1]
if len(sys.argv)==3 :
    port=int(sys.argv[2])
else:
    port = 80

READBUF=16384   # size of data read from web server
s=None

for res in socket.getaddrinfo(hostname, port, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    # create socket
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error:
        s = None
        continue
    # connect to remote host
    try:
        print "Trying "+sa[0]
        s.connect(sa)
    except socket.error, msg:
        # socket failed
        s.close()
        s = None
        continue
    if s :
        print "Connected to "+sa[0]
        s.send('GET / HTTP/1.1\r\nHost:'+hostname+'\r\n\r\n')
        finished=False
        count=0
        while not finished:
            data=s.recv(READBUF)
            count=count+1
            if len(data)!=0:
                print repr(data)
            else:
                finished=True
        s.shutdown(socket.SHUT_WR)
        s.close()
        print "Data was received in ",count," recv calls"
        break
```

As mentioned above, the socket API is very low-level. This is the interface to the transport service. For a common and simple task, like retrieving a document from the Web, there are much simpler solutions. For example, the python standard library includes several high-level APIs to implementations of various application layer protocols including HTTP. For example, the httplib module can be used to easily access documents via HTTP.

[2] Experiments with the client indicate that the number of *socket.recv* calls can vary at each run. There are various factors that influence the number of such calls that are required to retrieve some information from a server. We'll discuss some of them after having explained the operation of the underlying transport protocol.

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site, using
# the standard httplib library

import httplib, sys

if len(sys.argv)!=3 and len(sys.argv)!=2:
 print "Usage : ",sys.argv[0]," hostname [port]"
 sys.exit(1)

path = '/'
hostname = sys.argv[1]
if len(sys.argv)==3 :
 port = int(sys.argv[2])
else:
 port = 80

conn = httplib.HTTPConnection(hostname, port)
conn.request("GET", path)
r = conn.getresponse()
print "Response is %i (%s)" % (r.status, r.reason)
print r.read()
```

Another module, urllib2 allows the programmer to directly use URLs. This is much more simpler than directly using sockets.

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site, using
# the standard urllib2 library

import urllib2, sys

if len(sys.argv)!=2:
 print "Usage : ",sys.argv[0]," url"
 sys.exit(1)

url = sys.argv[1]
conn = urllib2.urlopen(url)
print conn.read()
```

But simplicity is not the only advantage of using high-level libraries. They allow the programmer to manipulate higher-level concepts (e.g. *I want the content pointed by this URL*) but also include many features such as transparent support for the utilisation of *TLS* or IPv6.

The second type of applications that can be written by using the socket API are the servers. A server is typically runs forever waiting to process requests coming from remote clients. A server using the connectionless will typically start with the creation of a *socket* with the `socket.socket`. This socket can be created above the TCP/IPv4 networking stack (`socket.AF_INET`) or the TCP/IPv6 networking stack (`socket.AF_INET6`), but not both by default. If a server is willing to use the two networking stacks, it must create two threads, one to handle the TCP/IPv4 socket and the other to handle the TCP/IPv6 socket. It is unfortunately impossible to define a socket that can receive data from both networking stacks at the same time with the python socket API.

A server using the connectionless service will typically use two methods from the socket API in addition to those that we have already discussed.

- `socket.bind` is used to bind a socket to a port number and optionally an IP address. Most servers will bind their socket to all available interfaces on the servers, but there are some situations where the server may prefer to be bound only to specific IP addresses. For example, a server running on a smartphone might want to be bound to the IP address of the WiFi interface but not on the 3G interface that is more expensive.
- `socket.recvfrom` is used to receive data from the underlying networking stack. This method returns both the sender's address and the received data.

The code below illustrates a very simple server running above the connectionless transport service that simply

prints on the standard output all the received messages. This server uses the TCP/IPv6 networking stack.

```
import socket, sys

PORT=int(sys.argv[1])
BUFF_LEN=8192

s=socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
s.bind(('',PORT,0,0))
while True:
    data, addr = s.recvfrom( BUFF_LEN )
    if data=="STOP" :
        print "Stopping server"
        sys.exit(0)
    print "received from ", addr, " message:", data
```

A server that uses the reliable byte stream service can also be built above the socket API. Such a server starts by creating a socket that is bound to the port that has been chosen for the server. Then the server calls the `socket.listen` method. This informs the underlying networking stack of the number of transport connection attempts that can be queued in the underlying networking stack waiting to be accepted and processed by the server. The server typically has a thread waiting on the `socket.accept` method. This method returns as soon as a connection attempt is received by the underlying stack. It returns a socket that is bound to the established connection and the address of the remote host. With these methods, it is possible to write a very simple web server that always returns a *404* error to all *GET* requests and a *501* errors to all other requests.

```
# An extremely simple HTTP server

import socket, sys, time

# Server runs on all IP addresses by default
HOST=''
# 8080 can be used without root priviledges
PORT=8080
BUFLEN=8192 # buffer size

s = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
try:
    print "Starting HTTP server on port ", PORT
    s.bind((HOST,PORT,0,0))
except socket.error :
    print "Cannot bind to port :",PORT
    sys.exit(-1)

s.listen(10) # maximum 10 queued connections

while True:
    # a real server would be multithreaded and would catch exceptions
    conn, addr = s.accept()
    print "Connection from ", addr
    data=''
    while not '\n' in data :  # wait until first line has been received
        data = data+conn.recv(BUFLEN)
    if data.startswith('GET'):
        # GET request
        conn.send('HTTP/1.0 404 Not Found\r\n')
        # a real server should serve files
    else:
        # other type of HTTP request
        conn.send('HTTP/1.0 501 Not implemented\r\n')

    now = time.strftime("%a, %d %b %Y %H:%M:%S", time.localtime())
    conn.send('Date: ' + now +'\r\n')
    conn.send('Server: Dummy-HTTP-Server\r\n')
```

```
conn.send('\r\n')
conn.shutdown(socket.SHUT_RDWR)
conn.close()
```

This server is far from a production-quality web server. A real web server would use multiple threads and/or non-blocking IO to process a large number of concurrent requests [3] . Furthermore, it would also need to handle all the errors that could happen while receiving data over a transport connection. These are outside the scope of this section and additional information on more complex networked applications may be found elsewhere. For example, [RG2010] provides an in-depth discussion of the utilisation of the socket API with python while [SFR2004] remains an excellent source of information on the socket API in C.

4.3.2 Exercises

1. Amazon provides the S3 storage service where companies and researchers can store lots of information and perform computations on the stored information. Amazon allows users to send files through the Internet, but also by sending hard-disks. Assume that a 1 Terabyte hard-disk can be delivered within 24 hours to Amazon by courier service. What is the minimum bandwidth required to match the bandwidth of this courier service ?

2. Several large data centers operators (e.g. Microsoft and google) have announced that they install servers as containers with each container hosting up to 2000 servers. Assuming a container with 2000 servers and each storing 500 GBytes of data, what is the time required to move all the data stored in one container over one 10 Gbps link ? What is the bandwidth of a truck that needs 10 hours to move one container from one data center to another.

3. The socket interface allows you to use the UDP protocol that provides the connectionless service on a Unix host. UDP, in theory, allows you to send SDUs of up to 64 KBytes.

- Implement a small UDP client and a small UDP server in C
- run the client and the servers on different workstations to determine experimentally the largest SDU that is supported by your language and OS. If possible, use different languages and Operating Systems in each group.

4. The time protocol, defined in **RFC 868** allows to read the current time on a remote host. Implement this very simple protocol on top of the UDP and TCP sockets. Compared the time required to retrieve this time information over UDP and TCP.

5. By using the socket interface, implement on top of the connectionless unreliable service provided by UDP a simple client that sends the message shown in the figure below.

   In this message, the bit flags should be set to *01010011b*, the value of the 16 bits field must be the square root of the value contained in the 32 bits field, the character string must be an ASCII representation (without any trailing *0*) of the number contained in the 32 bits character field. The last 16 bits of the message contain an Internet checksum that has been computed over the entire message.

   Upon reception of a message, the server verifies that :

   - the flag has the correct value
   - the 32 bits integer is the square of the 16 bits integer
   - the character string is an ASCII representation of the 32 bits integer
   - the Internet checksum is correct

   If the verification succeeds, the server returns a SDU containing *11111111b*. Otherwise it returns *01010101b*

   Your implementation must be able to run on both low endian and big endian machines. If you have access to different types of machines (e.g. x86 laptops and SPARC servers), try to run your implementation on both types of machines.

[3] There are many production quality web servers software available. apache is a very complex but widely used one. thttpd and lighttpd are less complex and their source code is probably easier to understand.

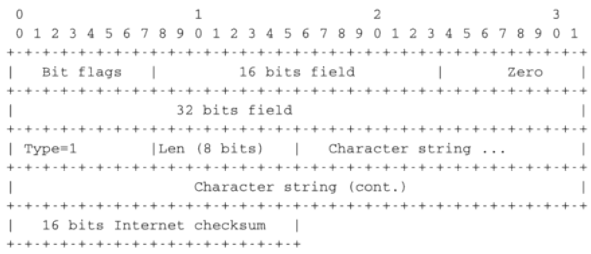


Figure 4.6: Simple SDU format

6. The socket library is also used to develop applications above the reliable bytestream service provided by TCP. We have implemented in C language a small tool to send information to a server on port *62141*. This server should operate as follows :

   - the server listens on port *62141* for a TCP connection
   - upon the establishment of a TCP connection, the server sends an integer by using the following TLV format :
     - the first two bits indicate the type of information (01 for ASCII, 10 for boolean)
     - the next six bits indicate the length of the information (in bytes)
     - An ASCII TLV has a variable length and the next bytes contain one ASCII character per byte. A boolean TLV has a length of one byte. The byte is set to *00000000b* for *true* and *00000001b* for false.
   - the client replies by sending the received integer encoded as a 32 bits integer in *network byte order*
   - the server returns a TLV containing *true* if the integer was correct and a TLV containing *false* otherwise and closes the TCP connection

   Implement a server in C that interacts with our client `/exercises/c/tlv_ex.c`

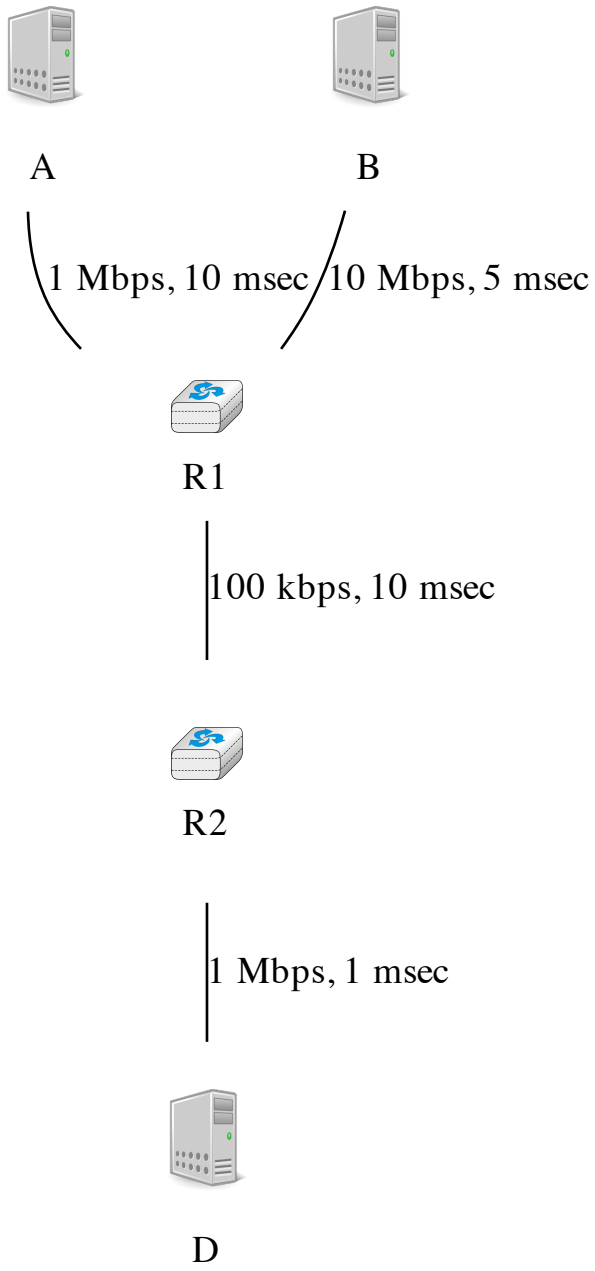
7. The Trivial File Transfer Protocol (TFTP), defined in **RFC 1350** is a simple file transfer protocol that runs on top of UDP. Implement a client for this protocol that allows to retrieve a file on a remote server.

4.4 Sharing resources

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=4

4.4.1 Exercises

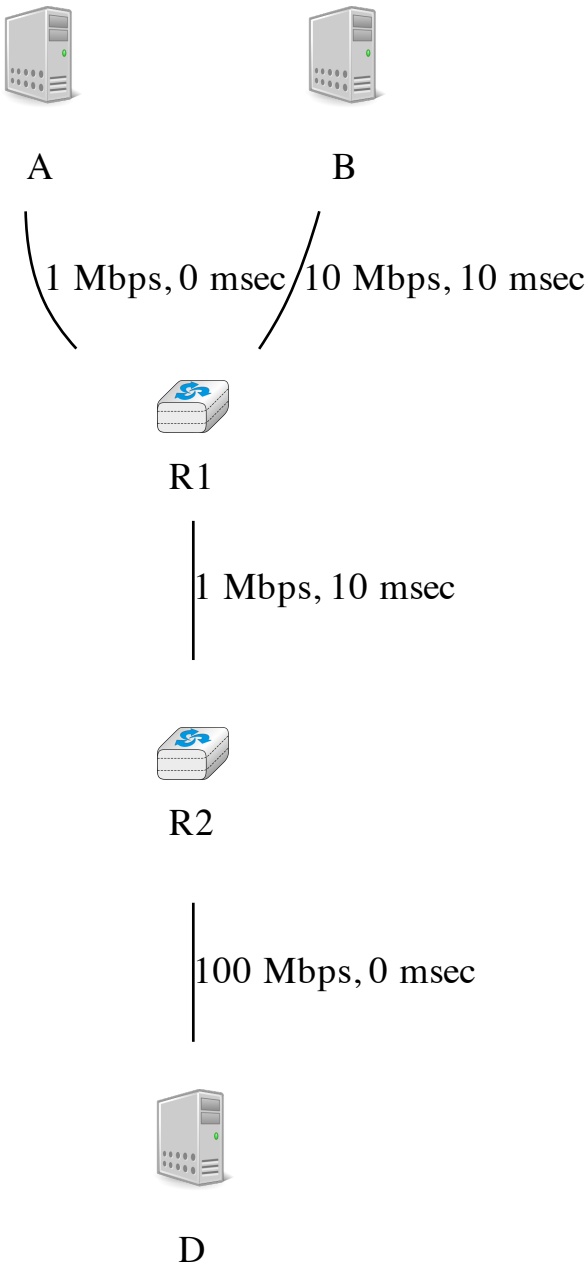
1. Consider the network depicted in the figure below.



In this network, compute the minimum round-trip-time between *A* (resp. *B*) and *D*. Perform the computation if the hosts send segments containing 100 bits, 1000 bits and 10,000 bits.

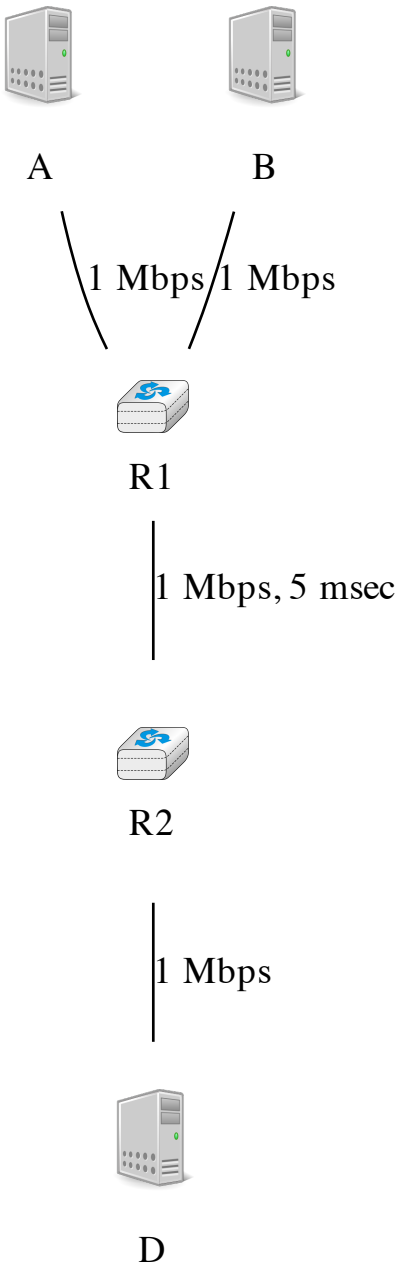
How is the maximum round-trip-time influenced if the buffers of router *R1* store 10 or 100 packets ?

2. Consider a slightly different variant of the above network.



If hosts *A* and *B* transmit 1000 bits segments and use a sending window of four segments, what is the maximum throughput that they can achieve ?

3. Consider the simple network depicted in the figure below.

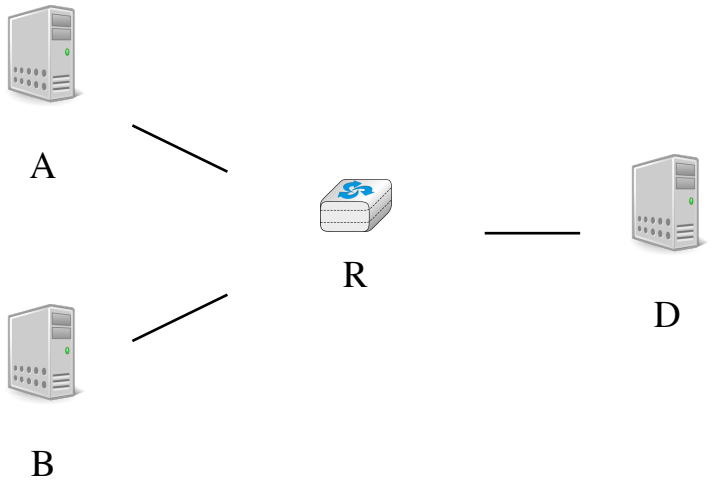


In the above network, all links have a 1 Mbps bandwidth and a negligible delay, except the *R1-R2* link. Host *A* uses a sending window of 10 segments. If there is no congestion control, what will be the delay experienced by host *B* if it sends a single segment when the network is in steady state (i.e. host *A* has been transmitting segments during the last seconds).

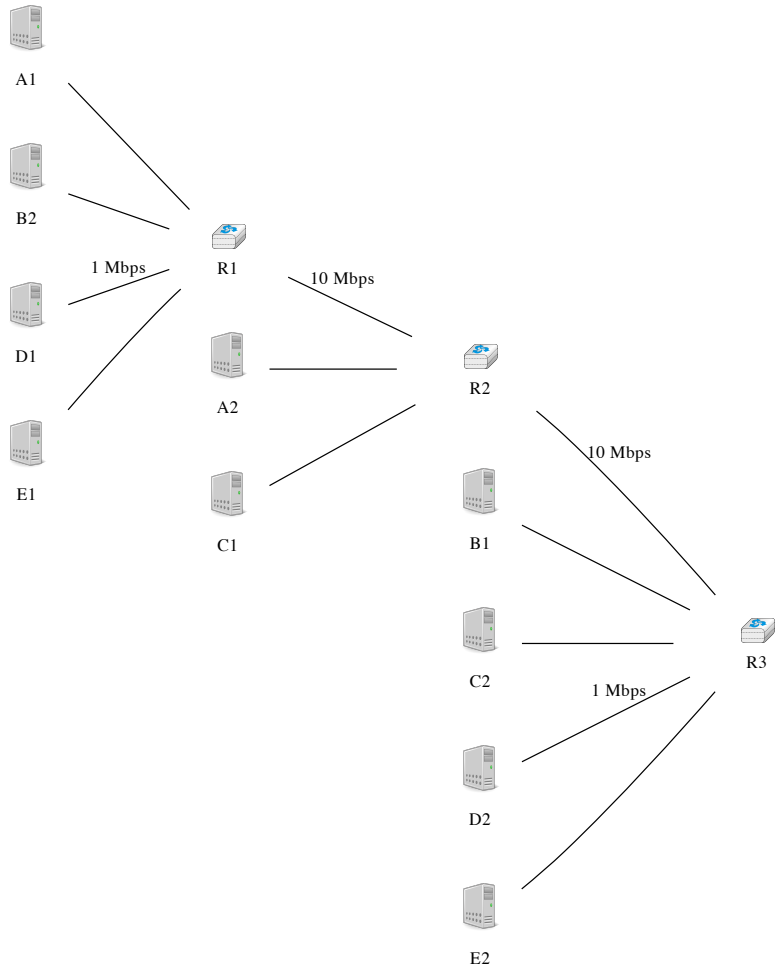
4. Same question as above, except that router *R1* uses a round-robin scheduler and two queues. The first queue contains all packets sent by host *A* and the second all packets sent by host *B*.

5. When analyzing the reaction of a network using round-robin schedulers, it is sometimes useful to consider that the packets sent by each source are equivalent to a fluid and that each scheduler acts as a tap. Using

this analogy, consider the network below. In this network, all the links are 100 Mbps and host *B* is sending packets at 100 Mbps. If A sends at 1, 5, 10, 20, 30, 40, 50, 60, 80 and 100 Mbps, what is the throughput that destination *D* will receive from *A*. Use this data to plot a graph that shows the portion of the traffic sent by host *A* which is received by host *D*.



6. Max-min fairness is the fairness objective for most congestion control schemes that operate in networks. Compute the max-min fair bandwidth allocation in the network below.



In this network, all hosts are attached with a 100 Mbps link, except hosts *D1* and *D2*. Data always flows from the host named *X1* to the host named *X2*.

7. Compute the max-min fair bandwidth allocation in the network below.

8. Consider the simple network depicted in the figure below.

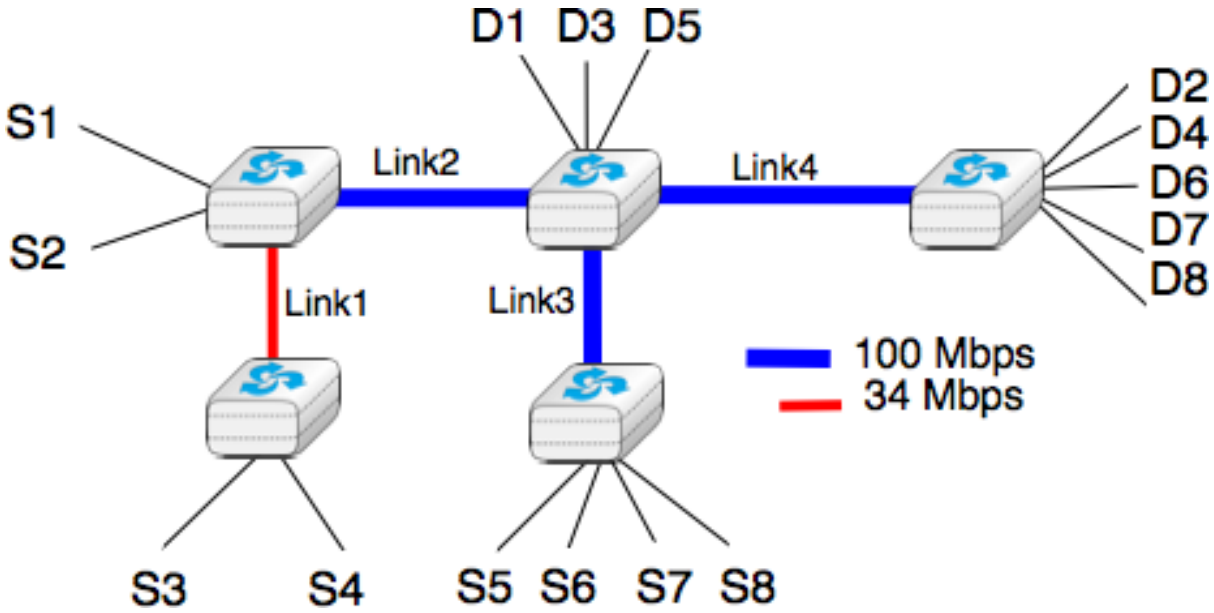
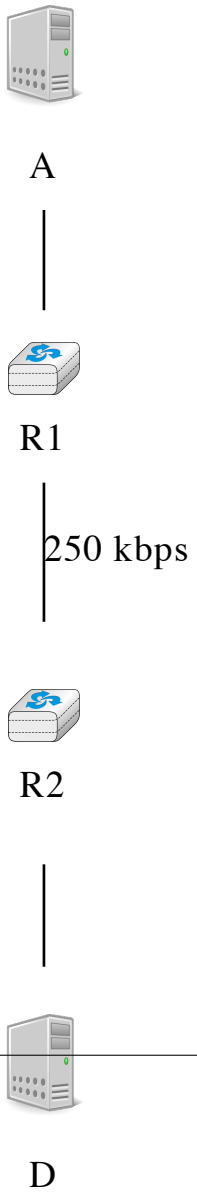


Figure 4.7: Simple network topology

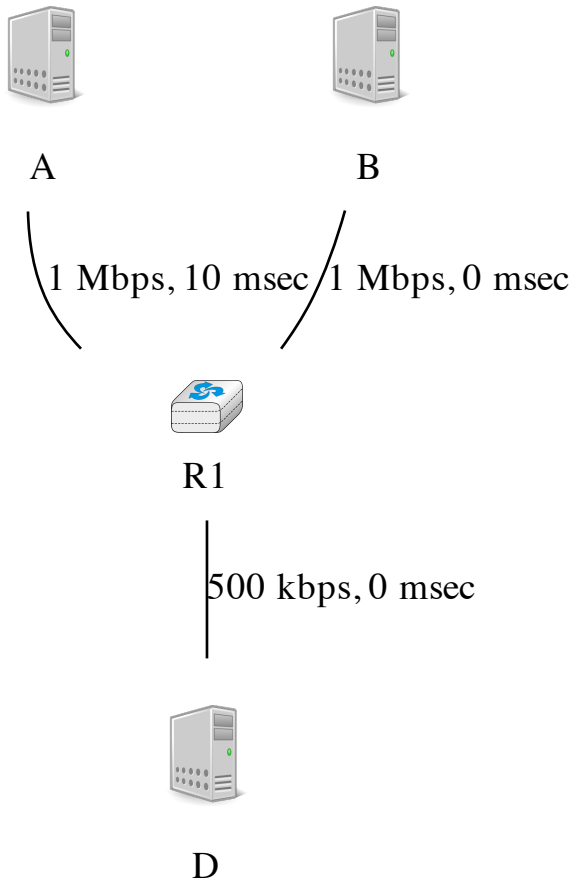


In this network, hosts are attached via 1 Mbps links while a 250 Kbps link is used between the routers. The propagation delays in the network are negligible. Host *A* sends 1000 bits long segments so that it takes one msec to transmit one segment on the *A-R1* link. Neglecting the transmission delays for the acknowledgements, what is the minimum round-trip time measured on host *A* with such segments ?

9. In the network above, represent by using a table as in the book the transmission of five 1000 bits segment by host *A*. How long does this transmission lasts if there is no congestion control and host *A* uses a sending window of two segments.

10. Same question as above, but now host *A* uses the simple DECBIT congestion control mechanism and a maximum window size of four segments.

11. Consider the network depicted in the figure below.



Hosts *A* and *B* use the simple congestion control scheme described in the book and router *R1* uses the DECBIT mechanism to mark packets as soon as its buffers contain one packet. Hosts *A* and *B* need to send five segments and start exactly at the same time. How long does each hosts needs to wait to receive the acknowledgement for its fifth segment ?

12. CSMA uses acknowledgements but not CSMA/CD. In networks using CSMA/CD how can a host verify that its frame has been received correctly ?

13. In CSMA/CD, would it be possible to increase or decrease the duration of the slottime ? Justify your answer

14. Consider a CSMA/CD network that contains hosts that generate frames at a regular rate. When the transmission rate increases, the amount of collisions increases. For a given network load, measured in bits/sec, would the number of collisions be smaller, equal or larger with short frames than with long frames ?
15. Compare two CSMA sources in a network with some load. When the channel becomes free, the first source is able to transmit its frame within less than one microsecond. The second source is slower and takes half a slotTime to transmit its frame. Compare the collisions that will affect the two sources.
16. What is the capture effect in a network using CSMA/CD ?
17. What is the hidden station problem in a network using CSMA/CA ?

4.5 Application layer

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=5

4.5.1 The DNS

The Domain Name System (DNS) plays a key role in the Internet today as it allows applications to use fully qualified domain names (FQDN) instead of IPv4 or IPv6 addresses. Many tools enable queries through DNS servers. For this exercise, we will use dig which is installed on most Unix systems.

A typical usage of dig is as follows

```
dig @server -t type fqdn
```

where

- *server* is the IP address or the name of a DNS server or resolver
- *type* is the type of DNS record that is requested by the query such as *NS* for a nameserver, *A* for an IPv4 address, *AAAA* for an IPv6 address, *MX* for a mail relay, ...
- *fqdn* is the fully qualified domain name being queried

dig also contains some additional parameters and flags that are described in the manpage. Among these, the *+trace* flag allows to trace all requests that are sent when recursing through DNS servers.

1. What are the IP addresses of the resolvers that the *dig* implementation you are using relies on [4] ?
2. What is the IPv6 address that corresponds to *inl.info.ucl.ac.be* ? Which type of DNS query does *dig* send to obtain this information ?
3. Which type of DNS request do you need to send to obtain the nameservers that are responsible for a given domain ?
4. What are the nameservers that are responsible for the *be* top-level domain ? Where are they located ? Is it possible to use IPv6 to query them ?
5. When run without any parameter, *dig* queries one of the root DNS servers and retrieves the list of the names of all root DNS servers. For technical reasons, there are only 13 different root DNS servers. This information is also available as a text file from http://www.internic.net/zones/named.root What are the IPv6 addresses of all these servers.
6. Assume now that you are residing in a network where there is no DNS resolver and that you need to start your query from the DNS root.
    - Use *dig* to send a query to one of these root servers to find the IPv6 address of the DNS server(s) (NS record) responsible for the *org* top-level domain

[4] On a Linux machine, the *Description* section of the *dig* manpage tells you where *dig* finds the list of nameservers to query.

- Use *dig* to send a query to one of these DNS servers to find the IP address of the DNS server(s) (NS record) responsible for *root-servers.org*
- Continue until you find the server responsible for *www.root-servers.org*
- What is the lifetime associated to this IPv6 address ?

7. Perform the same analysis for a popular website such as *www.google.com*. What is the lifetime associated to the corresponding IPv6 address ? If you perform the same request several times, do you always receive the same answer ? Can you explain why a lifetime is associated to the DNS replies ?
8. Use *dig* to find the mail relays used by the *uclouvain.be* and *gmail.com* domains. What is the *TTL* of these records ? Can you explain the preferences used by the *MX* records. You can find more information about the MX records in **RFC 5321**
9. When *dig* is run, the header section in its output indicates the *id* the DNS identifier used to send the query. Does your implementation of *dig* generates random identifiers ?

```
dig -t MX gmail.com

; <<>> DiG 9.4.3-P3 <<>> -t MX gmail.com
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25718
```

10. A DNS implementation such as *dig* and more importantly a name resolver such as bind or unbound, always checks that the received DNS reply contains the same identifier as the DNS request that it sent. Why is this so important ?

- Imagine an attacker who is able to send forged DNS replies to, for example, associate *www.bigbank.com* to his own IP address. How could he attack a DNS implementation that
  - sends DNS requests containing always the same identifier
  - sends DNS requests containing identifiers that are incremented by one after each request
  - sends DNS requests containing random identifiers

11. The DNS protocol can run over UDP and over TCP. Most DNS servers prefer to use UDP because it consumes fewer resources on the server. However, TCP is useful when a large answer is expected or when a large answer is expected. Use *time dig +tcp* to query a root DNS server. Is it faster to receive an answer via TCP or via UDP ?

4.5.2 Internet email protocols

Many Internet protocols are ASCII-based protocols where the client sends requests as one line of ASCII text terminated by *CRLF* and the server replies with one of more lines of ASCII text. Using such ASCII messages has several advantages compared to protocols that rely on binary encoded messages

- the messages exchanged by the client and the server can be easily understood by a developer or network engineer by simply reading the messages
- it is often easy to write a small prototype that implements a part of the protocol
- it is possible to test a server manually by using telnet Telnet is a protocol that allows to obtain a terminal on a remote server. For this, telnet opens a TCP connection with the remote server on port 23. However, most *telnet* implementations allow the user to specify an alternate port as *telnet hosts port* When used with a port number as parameter, *telnet* opens a TCP connection to the remote host on the specified port. *telnet* can thus be used to test any server using an ASCII-based protocol on top of TCP. Note that if you need to stop a running *telnet* session, `Ctrl-C` will not work as it will be sent by *telnet* to the remote host over the TCP connection. On many *telnet* implementations you can type `Ctrl-]` to freeze the TCP connection and return to the telnet interface.

1. Use your preferred email tool to send an email message to yourself containing a single line of text. Most email tools have the ability to show the *source* of the message, use this function to look at the message that

you sent and the message that you received. Can you find an explanation for all the lines that have been added to your single line email ?

2. The TCP protocol supports 65536 different ports numbers. Many of these port numbers have been reserved for some applications. The official repository of the reserved port numbers is maintained by the Internet Assigned Numbers Authority (IANA) on http://www.iana.org/assignments/port-numbers [5] Using this information, what is the default port number for the POP3 protocol ? Does it run on top of UDP or TCP ?

3. The Post Office Protocol (POP) is a rather simple protocol described in **RFC 1939**. POP operates in three phases. The first phase is the authorization phase where the client provides a username and a password. The second phase is the transaction phase where the client can retrieve emails. The last phase is the update phase where the client finalises the transaction. What are the main POP commands and their parameters ? When a POP server returns an answer, how can you easily determine whether the answer is positive or negative ?

4. On smartphones, users often want to avoid downloading large emails over a slow wireless connection. How could a POP client only download emails that are smaller than 5 KBytes ?

4.5.3 The HyperText Transfer Protocol

1. System administrators who are responsible for web servers often want to monitor these servers and check that they are running correctly. As a HTTP server uses TCP on port 80, the simplest solution is to open a TCP connection on port 80 and check that the TCP connection is accepted by the remote host. However, as HTTP is an ASCII-based protocol, it is also very easy to write a small script that downloads a web page on the server and compares its content with the expected one. Use *telnet* to verify that a web server is running on host *cnp3bis.info.ucl.ac.be* [6]

2. Instead of using *telnet* on port 80, it is also possible to use a command-line tool such as curl Use curl with the *–trace-ascii tracefile* option to store in *tracefile* all the information exchanged by curl when accessing the server.

   - what is the version of HTTP used by curl ?
   - can you explain the different headers placed by curl in the request ?
   - can you explain the different headers found in the response ?

3. HTTP 1.1, specified in **RFC 2616** forces the client to use the *Host:* in all its requests. HTTP 1.0 does not define the *Host:* header, by most implementations support it. By using *telnet* and *curl* retrieve the first page of the http://cnp3bis.info.ucl.ac.be webserver by sending http requests with and without the *Host:* header. Explain the difference between the two .

4. The headers sent in a HTTP request allow the client to provide additional information to the server. One of these headers is the *Accept-Language* header that allows to indicate the preferred language of the client [7]. For example, *curl -HAccept-Language:en http://www.google.be' will send to 'http://www.google.be* a HTTP request indicating English (en) as the preferred language. Does google provide a different page in French (fr) and Walloon (wa) ? Same question for *http://www.uclouvain.be* (given the size of the homepage, use `diff` to compare the different pages retrieved from *www.uclouvain.be*)

5. Compare the size of the http://www.yahoo.com and http://www.google.com web pages by downloading them with curl

6. What is a http cookie ? List some advantages and drawbacks of using cookies on web servers.

7. Assume that you are now responsible for the *http://www.belgium.be*. The government has built two datacenters containing 1000 servers each in Antwerp and Namur. This website contains static information and your objective is to balance the load between the different servers and ensures that the service remains up even if one of the datacenters is disconnected from the Internet due to flooding or other natural disasters. What are the techniques that you can use to achieve this goal ?

[5] On Unix hosts, a subset of the port assignments is often placed in */etc/services*

[6] The minimum command sent to a HTTP server is *GET / HTTP/1.0* followed by CRLF and a blank line

[7] The list of available language tags can be found at http://www.iana.org/assignments/language-subtag-registry Versions in other formats are available at http://www.langtag.net/registries.html Additional information about the support of multiple languages in Internet protocols may be found in rfc5646

8. The ipvfoo extension on google chrome allows to visually detect whether a website is using IPv6 and IPv4, but also to see which web sites have been contacted when rendering a given webpage. Some websites are distributed over several dozens of different servers. Can you find one ?
9. Recent HTTP browsers have started to replace HTTP with the SPDY protocol. What are the characteristics of this protocol and why is it faster than HTTP ?

4.6 Configuring DNS and HTTP servers

Configuring DNS and HTTP servers can be complex on real hosts. To allow you to learn network configurations without risking breaking anything, we will use Netkit. Netkit is network emulator based on User Mode Linux. It allows to easily set up an emulated network composed of virtual Linux machines, that can act as end-host or routers.

Note: Where can I find Netkit?

Netkit is available at http://www.netkit.org. For the labs, we have built a custom netkit image which is installed on the INGI servers [8] .

There are two ways to use Netkit : The manual way, and by using pre-configured labs. In the first case, you boot and control each machine individually, using the commands starting with a "v" (for virtual machine). In the second case, you can start a whole network in a single operation. The commands for controlling the lab start with a "l". The man pages of those commands is available from http://wiki.netkit.org/man/man7/netkit.7.html

A netkit lab is simply a directory containing at least a configuration file called *lab.conf*, and one directory for each virtual machine.

The directory of each device is initially empty, but will be used by Netkit to store their filesystem.

The lab directory can contain optional files. Some labs will include configuration files while others will use scripts that are executed when the virtual machines boot.

Starting a lab consists thus simply in unpacking the provided archive, going into the lab directory and typing *lstart* to start the network.

Note: netkit tools

As the virtual machines run Linux, standard networking tools such as `ping(8)`, tcpdump, netstat etc. are available.

Another useful hint is that is it possible to share files between the Netkit virtual machines and the local host. Virtual machines can access to the directory of the lab they belong to. This directory is mounted in their filesystem at the path */hostlab*.

4.6.1 Starting Netkit in the lab

Netkit has been installed in the INGI labs. In order to run the Netkit network emulator, launch the following commands:

```
ssh -Y <ingilogin>@permeke.info.ucl.ac.be
export PATH=$PATH:/etinfo/applications/netkit/bin
```

To launch a single host instance, use the command `vstart`:

```
vstart hostname
```

To launch the DNS lab[#fdnslab]_, use the following commands:

[8] The image that we use is a custom Netkit filesystem with a recent 64 bits Linux kernel. If you wish to test it on your own Linux machine, you can install Netkit as explained on http://wiki.netkit.org and download the filesystem and kernel images from http://cnp3bis.info.ucl.ac.be/netkit/netkit-images.zip Note that this archive has a compressed size of more than 1 GByte !

```
cp -r /etinfo/applications/netkit/dnslab/ $HOME/    # do not forget the trailing /'s
lstart -f -d $HOME/dnslab
```

To stop the lab, please stop all the involved instances by using the command `halt` inside each virtual machine.

Do not forget to cleanup the virtual disks when you are finished:

```
rm -f $HOME/dnslab/\*.disk
```

4.6.2 Exploring DNS

In this lab, you will experiment with the Domain Name system. Several DNS servers and resolvers are preconfigured in the Netkit which is provided.

Below, you can find a graph where the DNS topology we will use is depicted.

To begin experimentation, start the lab by using the commands explained above. In this lab, the DNS servers are correctly configured. We ask you to find the IP address of the following fully qualified domain names (FQDN):

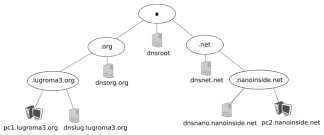
- `pc2.nanoinside.net`
- `dnsorg.org`
- `dnsroot`

For this, you should use the `dig` command whose syntax is :

```
dig @server -t type FQDN
```

If no server is specified, `dig` uses the default resolver that you can find in the configuration file `/etc/resolv.conf`.

While doing these requests, observe the packets that are exchanged between the differents DNS server with the `+trace` option. Is this what you expected? Sketch the Questions/responses on the figure below.



You have learned that DNS can work in two ways: Forward and Reverse. We will now resolve IPv6 addresses into their corresponding DNS names. Find the FQDN domain name of the following IPv6 addresses :

- `2001:db8:ba1:b0a::22`
- `2001:db8:ba1:b0a::2`

Again, you should use the `dig` command but with the `-x` option.

```
dig @server -x ipv6
```

with as parameter the IPv6 address you want to resolve.

4.6.3 Using DNS to access a website

Now that you have played a bit with deployed DNS servers and resolvers, we will now try to add a DNS entry that will point to some IP address and setup a website that can be reached through the added DNS entry.

We will create the website on `pc2` and we will call it `helloworld.nanoinside.net`. You thus have to add a DNS entry so that `helloworld.nanoinside.net` points to the IP address of `pc2`. See https://help.ubuntu.com/community/BIND9ServerHowto for a tutorial on how to configure `bind9`.

Once the DNS entry is set up, it is time to configure the web server. `Apache2` is installed. See http://tuxtweaks.com/2009/07/how-to-configure-apache-linux/ for a tutorial. The final goal is to see "Hello world !" when accessing the website:

```
$ curl -s helloworld.nanoinside.net
Hello world !
```

The configuration files of apache are located in `/etc/apache2/`

Enjoy !

4.7 Experimenting with Internet transport protocols

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=6

Transport protocols play a very important role on today's Internet. Most applications interact directly with them to exchange data. In the previous lab, you have performed DNS queries using dig. Each DNS request that you sent or reply that you received were placed inside UDP segments.

4.7.1 Packet trace analysis

When debugging networking problems or to analyse performance problems, it is sometimes useful to capture the segments that are exchanged between two hosts and to analyse them.

Several packet trace analysis tools are available, either as commercial or open-source tools. These tools are able to capture all the packets exchanged on a link. Of course, capturing packets require administrator privileges. They can also analyse the content of the captured packets and display information about them. The captured packets can be stored in a file for offline analysis.

tcpdump is probably one of the most well known packet capture software. It is able to both capture packets and display their content. tcpdump is a text-based tool that can display the value of the most important fields of the captured packets. Additional information about tcpdump may be found in *`tcpdump(1)`*.

The text below is an example of the output of tcpdump for all the TCP segments exchanged during the download of a short webpage over HTTP.

```
11:58:54.207193 IP6 2001:db8:b0:15:da:b055:0:2.52305 > 2001:db8:be:600d::2.80: Flags [S], seq 5445
11:58:54.207628 IP6 2001:db8:be:600d::2.80 > 2001:db8:b0:15:da:b055:0:2.52305: Flags [S.], seq 190
11:58:54.208344 IP6 2001:db8:b0:15:da:b055:0:2.52305 > 2001:db8:be:600d::2.80: Flags [.], ack 1, w
11:58:54.208360 IP6 2001:db8:b0:15:da:b055:0:2.52305 > 2001:db8:be:600d::2.80: Flags [P.], seq 1:1
11:58:54.208750 IP6 2001:db8:be:600d::2.80 > 2001:db8:b0:15:da:b055:0:2.52305: Flags [.], ack 110
11:58:54.227126 IP6 2001:db8:be:600d::2.80 > 2001:db8:b0:15:da:b055:0:2.52305: Flags [P.], seq 1:4
11:58:54.227526 IP6 2001:db8:b0:15:da:b055:0:2.52305 > 2001:db8:be:600d::2.80: Flags [.], ack 491
11:58:54.234242 IP6 2001:db8:b0:15:da:b055:0:2.52305 > 2001:db8:be:600d::2.80: Flags [F.], seq 110
11:58:54.234921 IP6 2001:db8:be:600d::2.80 > 2001:db8:b0:15:da:b055:0:2.52305: Flags [F.], seq 491
11:58:54.235245 IP6 2001:db8:b0:15:da:b055:0:2.52305 > 2001:db8:be:600d::2.80: Flags [.], ack 492
```

You can easily recognize in the output above the *SYN* segment containing the *MSS* option, the *SYN+ACK* segment returned by the server and then the few data segments exchanged on the connection.

wireshark is more recent than tcpdump. It evolved from the ethereal packet trace analysis software. It can be used as a text tool like tcpdump. For a TCP connection, wireshark would provide almost the same output as tcpdump. The main advantage of wireshark is that it also includes a graphical user interface that allows to perform various types of analysis on a packet trace.

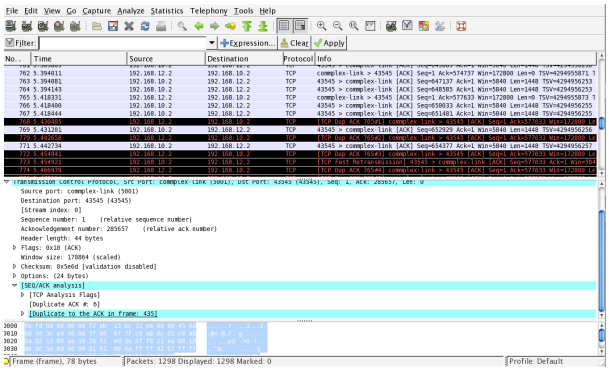


Figure 4.8: Wireshark : default window

The wireshark window is divided in three parts. The top part of the window is a summary of the first packets from the trace. By clicking on one of the lines, you can show the detailed content of this packet in the middle part of the window. The middle of the window allows you to inspect all the fields of the captured packet. The bottom part of the window is the hexadecimal representation of the packet, with the field selected in the middle window being highlighted.

wireshark is very good at displaying packets, but it also contains several analysis tools that can be very useful. The first tool is *Follow TCP stream*. It is part of the *Analyze* menu and allows you to reassemble and display all the payload exchanged during a TCP connection. This tool can be useful if you need to analyse for example the commands exchanged during a SMTP session.

The second tool is the flow graph that is part of the *Statistics* menu. It provides a time sequence diagram of the packets exchanged with some comments about the packet contents. See blow for an example.

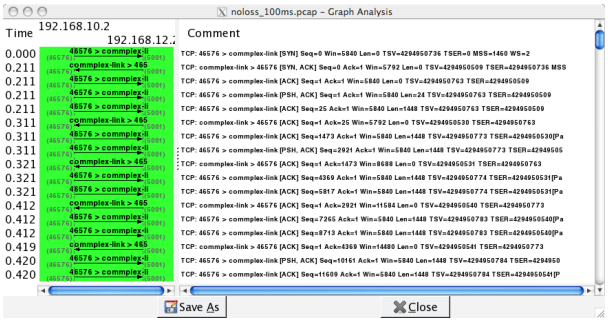


Figure 4.9: Wireshark : flow graph

The third set of tools are the *TCP stream graph* tools that are part of the *Statistics menu*. These tools allow you to plot various types of information extracted from the segments exchanged during a TCP connection. A first interesting graph is the *sequence number graph* that shows the evolution of the sequence number field of the captured segments with time. This graph can be used to detect graphically retransmissions.

A second interesting graph is the *round-trip-time* graph that shows the evolution of the round-trip-time in function of time. This graph can be used to check whether the round-trip-time remains stable or not. Note that from a packet trace, wireshark can plot two *round-trip-time* graphs, One for the flow from the client to the server and the other one. wireshark will plot the *round-trip-time* graph that corresponds to the selected packet in the top wireshark window.

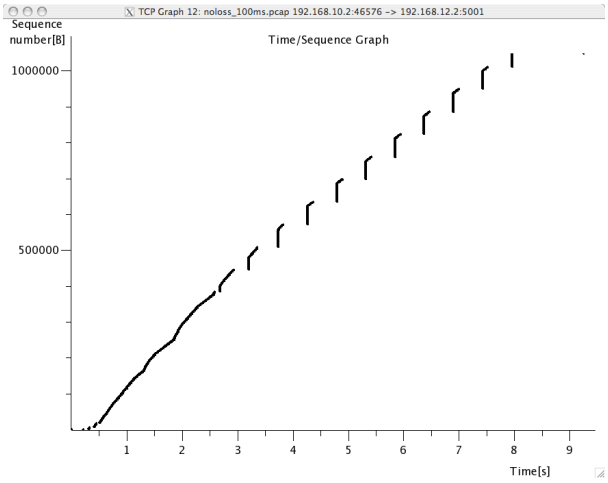


Figure 4.10: Wireshark : sequence number graph

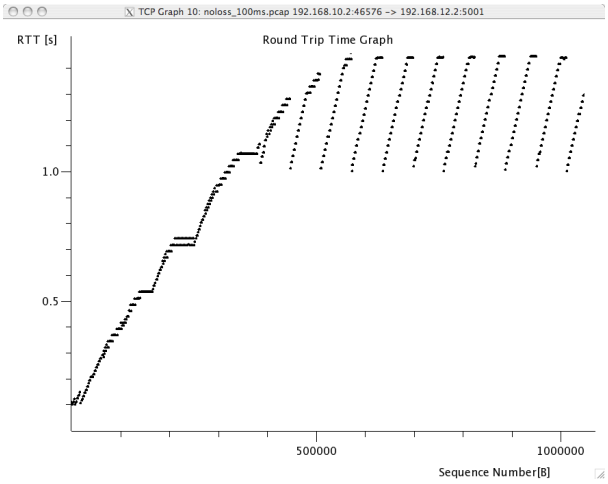
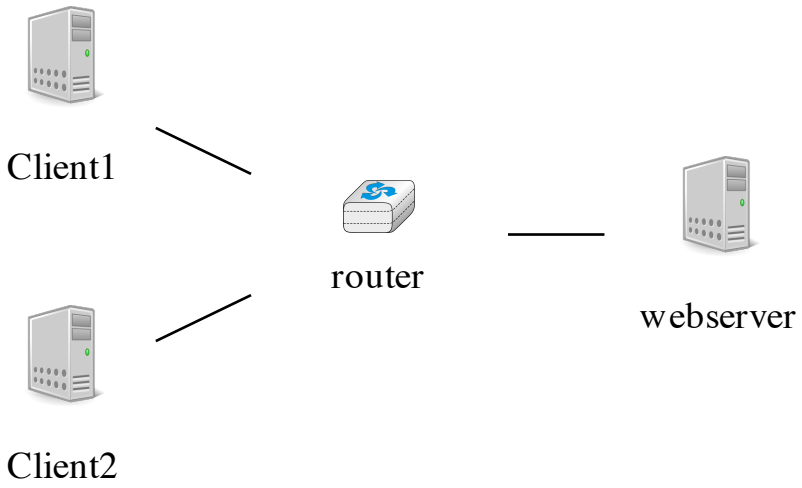


Figure 4.11: Wireshark : round-trip-time graph

4.7.2 Experimenting with UDP

For these UDP experiments, you will use a simple lab that contains four hosts.



This lab can be downloaded from `/netkit/netkit-lab_tcpudp.zip`

Two hosts, *Client1* and *Client2* are clients that you will use to send information to the *webserver* host through an intermediate *router*. Thanks to the *router*, you will be able to easily observe the packets that are exchanged and delay or discard some of them to see how the protocol reacts to these events.

Note: Discovering IP addresses

On a netkit lab such as this one, it is sometimes necessary to discover the IP addresses of the different hosts. On Linux, the IP addresses associated to an interface are configured by using *ifconfig(8)*. The output below shows the configuration of the interfaces of the *router*

```
#ifconfig
 eth0      Link encap:Ethernet  HWaddr 02:bb:88:79:c7:fb
           inet6 addr: fe80::bb:88ff:fe79:c7fb/64 Scope:Link
           inet6 addr: 2001:db8:b0:15:da:b055:0:1/96 Scope:Global
           UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
           RX packets:15 errors:0 dropped:0 overruns:0 frame:0
           TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:1052 (1.0 KiB)  TX bytes:636 (636.0 B)
           Interrupt:5

 eth1      Link encap:Ethernet  HWaddr 5a:f7:20:7e:4e:9d
           inet6 addr: 2001:db8:be:600d::1/64 Scope:Global
           inet6 addr: fe80::58f7:20ff:fe7e:4e9d/64 Scope:Link
           UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
           RX packets:7 errors:0 dropped:0 overruns:0 frame:0
           TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:488 (488.0 B)  TX bytes:636 (636.0 B)
           Interrupt:5
```

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:200 (200.0 B)  TX bytes:200 (200.0 B)
```

The output of *ifconfig(8)* shows that this *router* has three interfaces. The loopback interface (`lo` on Linux) is the default software-based interface of all hosts. The `eth0` interface is connected to the two clients while the `eth1` interface is connected to the webserver. The IP address of the *router* on `eth0` (resp. `eth1`) is `2001:db8:b0:15:da:b055:0::1` (resp. `2001:db8:be:600d::1`).

The *webserver* has been configured as a server that supports the following services :

- `http` over TCP on port `80`
- `echo` over both UDP and TCP on port `7`
- `discard` over both UDP and TCP on port `9`
- `daytime` over both UDP and TCP on port `13`
- `telnet` over TCP on port `23`

The last three services were popular services installed on all TCP/IP hosts. However, some of them caused security problems and nowadays they are rarely enabled on real servers.

`echo` is a very simple service. When a server receives some information, over UDP or TCP, it simply returns it to the client. `discard` is a kind of blackhole. All the information, sent over UDP or TCP, to a `discard` server is simply discarded upon reception. `daytime` is a very simple protocol that allows to query the current time on the server. The format of the response is described in **RFC 867**.

Several tools allow to send information over UDP and TCP. *telnet* is very useful to interact with TCP servers. *nc* (or `netcat`) is another tool which can be very useful when debugging network problems. It allows to easily contact servers over UDP or TCP, but can also be used to create simple but powerful servers from the command line. Several versions of `nc`/`netcat` have been written. See http://en.wikipedia.org/wiki/Netcat for additional details.

Start `tcpdump` on `router` to capture all UDP segments. The `tcpdump` manpage will show you how to only capture UDP segments.

1. Using `nc` on *Client1*, send data to the `discard` server running on *webserver*. Observe the segments that are exchanged. How does the client select its source port number ?
2. Using `nc` on *Client2*, send data to the `echo` server running on *webserver*. Use `tcpdump` to verify whether the data returned by the server is the same as the one sent by the client.

Note: Some useful tcpdump options

`tcpdump` contains many options as described in the *tcpdump*. Among these, the following ones could prove useful :

- `-n` : instructs `tcpdump` to print the addresses of the captured packets and do not try to resolve their names. To resolve names, `tcpdump` needs to query the reverse DNS servers and this may interfere with the packet capture or introduce delays.
- `-w filename` : instructs `tcpdump` to save the captured packets into a file for further postprocessing. The packet trace can then be read by using `tcpdump -r filename` or with Wireshark.
- `-v`, or `-vv` or even `-vvv` : use different levels of verbosity when printing information extracted from the packet
- `-S` forces TCP print the exact sequence/acknowledgements numbers found in the captured segments. By default, tcpdump prints sequence numbers that are relative to the beginning of the connection

- `-s snaplen` indicates the default size for the captured packets. Some versions of tcpdump use a default snaplength of 64 or 96 bytes, i.e. they only capture the beginning of the packets. This usually includes all useful headers. You might want to increase this value to capture long data segments.

4.7.3 Experiments with TCP

nc can also be used to interact with TCP servers. TCP is a complex protocol and a TCP implementation such the Linux kernel contains a large number of configuration parameters. To ease your understanding of the basic mechanisms of TCP, we have disabled most of the TCP options on *Client1* and *Client2*.

Start by using *tcpdump* on *router* to capture all the packets sent on the interface attached to *webserver*

1. Using *nc*, try to open a TCP connection to a port on *webserver* where there is no listening server, e.g. port `5`. How does *tcpdump* shows the first segment of the three way handshake. How does the TCP stack on *webserver* answers to this segment ? What are the TCP options used
2. Using *nc*, open a TCP connection to the `echo` port on *webserver* and send some information. From the *tcpdump* output, how does the server closes the TCP connection ?
3. Perform the same experiment as above, but now by using the `daytime` server.
4. Perform the same experiment as above, but now by using `wget` to retrieve the homepage on *webserver*. Which version of HTTP is used ? How is the TCP connection closed ?
5. The MSS option is the first option that was specified in TCP. It is used during the three-way-handshake to announce the Maximum Segment Size supported by a host. On Linux, the MSS value is computed from the maximum packet size of the underlying network. You can change the maximum packet size of the underlying network (or Maximum Transmission Unit - MTU) by using the command *ifconfig(8)*

   ```
   ifconfig eth0 mtu 1300
   ```

   This command reduces the MTU of interface `eth0` to 1300 bytes. Use *tcpdump(8)* to observe whether this change affects the segments sent by the client or by the server when *nc(1)* is used with the `echo` service.
6. The TCP stack on *Client1* was configured to disable all recent TCP options, including Window Scale, Timestamp and Selective acknowledgements. Enable the Timestamp option using the correct `sysctl` and verify with *tcpdump(8)* that this extension is actually used
7. The main benefit of TCP is that it can react to delays, losses and packet duplications. In a netkit lab, there are usually no delay and no losses or duplications. Fortunately, various tools can be used on the Linux kernel to emulate various network properties. Netem is one of these tools. It can be used on a router to add delay, losses or duplications when a router sends packets. Using the commands described in http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, configure the interface between *router* and *websever* with :

   1. A fixed delay of 100 milliseconds
   2. Packet losses of 10%, 50% and 95%
   3. Packet corruption
   4. Packet reordering

   Using *nc(1)* with the `discard` or `echo` service, observe by using *tcpdump(8)* how TCP reacts to these events during :

   - the three-way handshake
   - the data transmission phase
   - the connection release phase
8. Perform the same experiment with the `discard` service, but this time introduce errors on the link between *router* and *Client1*. Is TCP more affected from errors on the data segments or on the acknowledgements ?

9. Using a configuration with netem that includes a non-zero delay, packet losses and reordering, observe the benefits of using Selective Acknowledgements. For this, configure netem on the link between *router* and *webserver* and enable the selective acknowledgements with the `tcp_sack sysctl` on *Client2*. Observe the difference between *Client1*, which does not use the selective acknowledgements and *Client2*.

4.7.4 Injecting TCP segments

Packet capture tools like tcpdump and Wireshark are very useful to observe the segments that transport protocols exchange. They are also very useful to understand and debug network problems as we'll discuss in subsequent labs. TCP is a complex protocol that has evolved a lot since its first specification **RFC 793**. TCP includes a large number of heuristics that influence the reaction of a TCP implementation to various types of events. A TCP implementation interacts with the application through the `socket` API. Recently, several researchers from Google proposed packetdrill [CCB+2013]. packetdrill is a TCP test suite that was designed to develop unit tests to verify the correct operation of a TCP implementation. packetdrill interacts with the Linux TCP implementation in two ways :

- packetdrill can issue any system call through the socket interface
- packetdrill can inject any segment in the TCP stack as if it was received from a remote host

A detailed description of packetdrill and one example can be found in [CCB+2013]. packetdrill uses a syntax which is a mix between the C language and the tcpdump_syntax. The following example illustrates the three-way handshake with packetdrill

```
// create socket and listen for incoming connections
0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
0.000 bind(3, ..., ...) = 0
0.000 listen(3, 1) = 0

// inject the first SYN segment in the TCP stack at time 0.200
// MSS is set to 1000 to simplify computation

0.200 < S 0:0(0) win 4096 <mss 1000>
// We expect a SYN+ACK with the default Linux MSS
0.200 > S. 0:0(0) ack 1 <mss 1460>
// We reply by injecting a valid ack at time 0.300
0.300 < . 1:1(0) ack 1 win 4096
// At that time, the connection is established
0.300 accept(3, ..., ...) = 4

// Receive first segment
0.510 < . 1:1001(1000) ack 1 win 4096

// Expects one ack in response
0.510 > . 1:1(0) ack 1001

// Application reads received data
0.600 read(4, ..., 1000) = 1000

// Application writes 1000 bytes
0.650 write(4, ..., 1000) = 1000
// Expects that it will send one segment
0.650 > P. 1:1001(1000) ack 1001

// We reply with an acknowledgement after 50 msec
0.700 < . 1001:1001(0) ack 1001 win 257

// We inject a RST to close connection
0.701 < R. 1001:1001(0) ack 1001 win 4096
```

1. To show your understanding of the TCP state machine, use packetdrill to develop one script that demon-

strates the operation of TCP. Inside each group, ensure that there is at least one script that demonstrates :

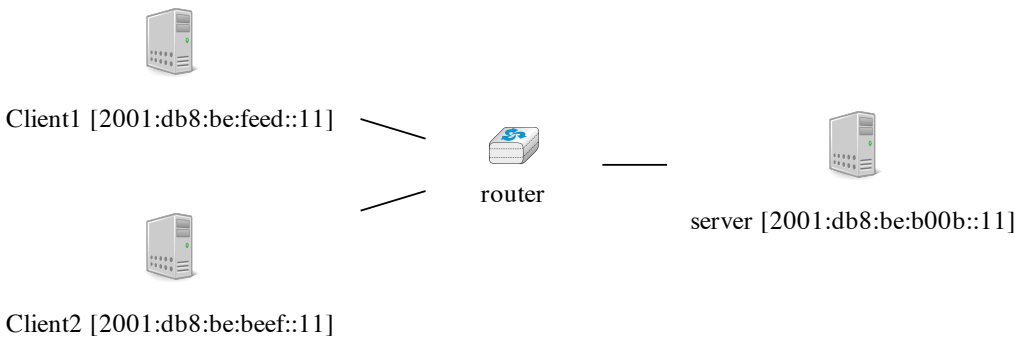
- the simultaneous establishment of a TCP connection when the client and the server generate a SYN almost at the same time
- the ability of TCP to accept out-of-order segments
- the Nagle algorithm
- the operation of TCP when the server announces a very small window
- the support of delayed acknowledgements (i.e. an acknowledgement is sent for every second segment or after 50 msec of delay when there is no reordering)
- the TCP fast retransmit
- different paths through the TCP state machine when closing a TCP connection (e.g. client sends FIN first, server sends FIN first, client and server send FIN at the same time, ...)
- the negotiation of the MSS during the three-way handshake
- the support of the TCP timestamp option
- the reaction of TCP when out-of-window data is received
- the reaction of TCP upon reception of a SYN+ACK segment when a connection has not yet been established

4.8 Experimenting with Internet congestion control

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=7

For this lab, you will reuse a slightly modified version of the TCP/UDP lab. The filesystem used for this lab contains several new software packages. Make sure to start the lab by using `lstart` from the lab's directory. The archive containing the lab directory can be downloaded from `/netkit/netkit-lab_congestion.zip`

Your objective with this lab is to better understand the TCP congestion control scheme and how it influences the performance of TCP.



During this lab, you will use three software packages that are very useful to understand TCP performance in real IP networks. To simulate network congestion, the bandwidth of the link between the router and the server has been shaped at 1 Mbps.

iperf (version 3) is a frequently used network performance testing tool. It is often used by network administrators who need to test the performance of a network between two hosts. To use iperf, you first need to start the server

process by using `iperf -s` on host `server` on the lab. The server listens on `5201` for measurements requests coming from the server. When a measurement starts, the client sends data to the server.

The iperf manpage lists all the options of the server and the client. The most interesting ones are :

- `-6` forces the utilisation of IPv6
- `--udp` uses UDP for the measurements instead of TCP which is the default
- `--set-mss x` sets the TCP MSS at x bytes
- `--bandwidth b` iperf tries to send data at b bits/sec (default for UDP is 1 Mbps, unlimited for TCP)
- `--verbose` provides much more detailed output, including TCP_INFO statistics with TCP on Linux
- `--window x` specifies the TCP window size in bytes on both the client and the server
- `--reverse` forces the measurement to be done from the server to the client
- `--parallel n` iperf uses n parallel flows to perform the measurement

4.8.1 Experiments

The experiments below will allow you to verify experimentally the key factors that influence the performance of the TCP congestion control scheme. We use a TCP implementation that supports the TCP congestion control scheme, but not the window scale, timestamp and selective acknowledgement options. You can, of course, enable these options if you want to experiment with them.

1. The round-trip-time is a key factor that influences the performance of TCP in a network. TCP maintains its own estimate of the round-trip-time over a connection. Other tools like *ping6(8)* can be used to measure the round-trip-time. Start a lab and use *ping6(8)* to measure the round-trip-time. Then, start iperf sessions from `client1` and `client2` and capture the TCP segments on `router` with *tcpdump(8)*. Analyze the collected trace with tcptrace or wireshark and observe the evolution of the measured round-trip-time.
2. An important factor to achieve a high goodput with TCP is the window size. Using the `--window` parameter of iperf, compare the performance achieved by a client with a window of 4 KBytes, 16 KBytes and 32 KBytes. Compare this with the bandwidth delay product in the emulated network.
3. The first factor that influences the performance of the TCP congestion control scheme is the round-trip-time. A TCP connection with a longer round-trip-time will react slower than a connection with a shorter round-trip time. Start an iperf server and use **netem_** to add a delay of
   - `10 msec` on the link between the `router` and `client1`
   - `200 msec` on the link between the `router` and `client2`

   Using iperf, verify experimentally that the `client1` obtains a higher goodput [9] than `client2`

1. The TCP congestion control scheme operates on a per TCP connection basis. This implies that a client that uses several parallel TCP connections should be favored compared to a client that uses a single TCP connection. Using the `--parallel `` parameter of iperf_, verify that this is indeed the case in a lab where ``client1` and `client2` have the same round-trip-time.
2. Another factor that influences the performance of TCP is the size of the transmitted segments. Using the `-mss` parameter of iperf, change the MSS on `client1` and verify whether this reduces its performance compared to `client2` (assuming, of course, that the round-trip-times are the same for the two clients)
3. In the book, we have explained that the TCP goodput was inversely proportional to the square root of the packet loss ratio. Using **netem_**, simulate different packet loss ratios and verify this formula.
4. In some cases, the goodput obtained by an application also depends on the performance of the application itself. For example, iperf provides the `--file` option that reads the data to be sent from a file instead of

[9] The goodput is defined as the total number of bytes transmitted by an application divided by the duration of the transfer. This is the measurement reported by iperf. It should not be confused with the *throughput* which is measured on the network interfaces and usually includes all the overheads of the different layers.

from memory to verify whether the disk is the bottleneck [10]. In our emulated lab, this option cannot be used. However, you can emulate a bottleneck on the client/server by using the `--bandwidth` parameter of iperf. Use this option and analyze the captured packet trace to see how you can identify this behavior from the trace.

4.8.2 Packet traces

Network administrators often need to debug performance problems in a network by looking only at packet traces. The three packet traces below were collected in an emulated network that is similar to the one we used in the lab. Can you analyze the packet traces and :

- order them by increasing TCP goodput
- identify the performance bottleneck inside each trace

The three traces can be downloaded from :

- `/exercises/traces/congestion1.pcap`
- `/exercises/traces/congestion2.pcap`
- `/exercises/traces/congestion3.pcap`

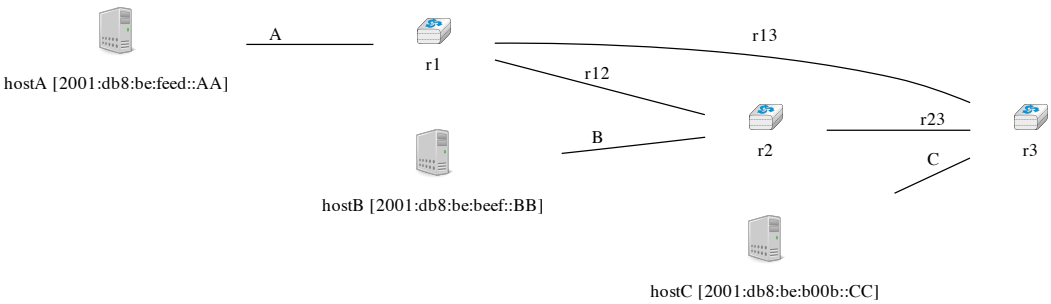
wireshark and tcptrace should help you to analyze these three traces.

4.9 Configuring IPv6

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=6

During the previous labs, you have mainly used existing netkit labs to understand transport layer protocols such as TCP and UDP. This is one usage of netkit, but not the most interesting one. An advantage of netkit compared to the utilisation of real networking devices is that it allows you to build quickly a network, configure it and test its operation. The netkit manpages, available from http://wiki.netkit.org/man/man7/netkit.7.html provide many details on the operation of netkit

netkit is composed of a set of scripts that read configuration files and allow you to run scripts and copy files on the virtual machines that represent the routers and the switches. All the information about a netkit lab resides in a directory. The first file in such a directory is the lab.conf file. This file is a text file that describes the network topology that you want to emulate.



The network above, with three hosts and three routers can be represented by the following lab.conf file.

[10] See http://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf-and-iperf3/disk-testing-using-iperf/ for additional information and an example.

```
# Header
LAB_DESCRIPTION="Lab to understand the basics of IPv6"
LAB_VERSION=1
LAB_AUTHOR="you"
# List of virtual machines in the lab
machines="hostA hostB hostC r1 r2 r3"
# First host, 64 MBytes of RAM
hostA[M]=64
# The link between hostA and r1 ends on interface 0 on hostA and r1
hostA[0]=A
r1[0]=A
# The link between hostB and r2
hostB[0]=B
r2[0]=B
# The link between hostC and r3
hostC[0]=C
r3[0]=C
# The link between r1 and r2
r1[1]=r12
r2[1]=r12
# The link between r1 and r3
r1[2]=r13
r3[2]=r13
# The link between r2 and r3
r2[2]=r23
r3[1]=r23
```

To build your own lab, you need to first define the subnetworks that compose the lab. Each subnetwork is identified by a label (*A* for the subnetwork between `hostA` and `r1`). Then, you connect each subnetwork to an interface on a host or router. Each interface is identified by a unique integer. The first interface has number `0`, the second number `1`, ... In the virtual machine, interface `0` corresponds to interface `eth0`. By default, a virtual machine has 32 Mbytes of virtual memory. If you need more memory, for example to run tcpdump, you can extend it with the `[M]` parameter. Other parameters are described in the netkit man pages.

Note: Big brother in your emulated network

In some scenarios, it might be interesting to have a virtual machine that is present on all links inside the emulated network. This virtual machine could allow you to easily collect packets on any link in the network. For example, if you would like to add such a monitoring machine in the network above, you could add the following configuration.

```
# Add nsa to list of machines
machines="hostA hostB hostC r1 r2 r3 nsa"
# NSA can capture all packets
nsa[0]=A
nsa[1]=B
nsa[2]=C
nsa[3]=r12
nsa[4]=r23
nsa[5]=r13
# To analyze all these packets, NSA needs memory
nsa[M]=128
```

The above `lab.conf` file defines the network topology and the configuration of the virtual machines. To allow lstart to start the lab, you need to create a directory for each virtual machine and a startup script. The directory is named `machine` where `machine` is the name chosen for the virtual machine. The startup script is specific for each virtual machine and is called `machine.startup`. For example, to create these directories and files, you could run the following bash script in the directory where the `lab.conf` file is stored.

```
for vm in hostA hostB hostC r1 r2 r3
do
  mkdir $vm
  touch $vm.startup
```

```
done
```

With these files, you can start the lab, but the virtual machines need to be configured before you can exchange packets. For this, you need first to manually assign IPv6 addresses. On Linux, IP addresses are configured by using the *ifconfig(8)* command [11]. This command takes a lot of parameters. A typical usage is the following :

```
# enable interface eth0
ifconfig eth0 up
# add one IPv6 address on this interface
ifconfig eth0 add 2001:db8:be:feed::AA/64
```

The first command above activates interface `eth0`. This command is mandatory before the interface can send/receive packets. The second command configures the IPv6 address associated to this interface. An IPv6 address is always composed of two parts :

- a subnet identifier
- a host identifier

The subnet identifier are the high order bits of the IPv6 address. They identify the subnet to which the interface is connected. In the example above, the subnet is 64 bits long and is `2001:db8:be:feed`. All IPv6 addresses that belong to this subnet can be reached by using the attached subnetwork (i.e. through the datalink layer) without passing through an intermediate router. The low order bits of the address represent the host identifier (`::AA` in the above configuration) inside the subnet.

Your first objective is to configure the IPv6 addresses of the three endhosts based on the information in the figure above. For this lab, we use only 64 bits subnets. Once an IPv6 address has been configured, you can verify that it is reachable from the host where it has been configured by issuing a *ping6(8)* command towards this address on this host.

Once IPv6 has been configured on the endhosts, you need to configure the IPv6 addresses on the three routers. Start by configuring the IPv6 addresses on the interfaces `eth0` of these routers. For this, select one identifier in the subnetwork attached to each host and assign one host identifier to the attached router.

The next step is to configure the IPv6 addresses on the links between the routers. Select one subnetwork identifier starting from `2001:db8:` for each inter-router subnetwork and configure the IPv6 addresses on the two attached routers.

At this point, all IPv6 addresses should have been configured. Make sure that you recorded in a text file all the commands that you typed. They will be necessary later on to automate the creation of the lab. A good idea would be to create an `/etc/hosts` file that contains the mapping between names and all assigned IPv6 addresses. This file is a text file such as the one below.

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
2001:db8:be:beef::AA      hostA
2001:db8:be:feed::BB      hostB
2001:db8:be:b00b::CC      hostC
```

Add to the file above the IPv6 addresses that you have configured for the routers. Make sure that a different name is used for each address chosen for a router. This file has the format of the `/etc/hosts` that provides name to address mappings when DNS is not in operation (as in this lab since we did not configure any DNS server). You can copy this file on all virtual hosts to have the list of all adresses on each host.

[11] You can use the `ip` command instead of *ifconfig(8)* or *route(8)*. See the Linux IPv6 Howto for additional information.

At this point, you have configured all IPv6 addresses, but there are still no routes. Without routes, packets will not be forwarded in the network. You need to manually configure the forwarding tables on the hosts and the routers.

On the hosts, configuring the routing table is simple. You simply need to add a default route towards the router that is directly connected to the host. This can be done by using the *route(8)* command [1].

```
#route -A inet6 add default gw 2001:db8:be:feed::11
```

The commande above adds a *default* route, i.e. a route towards `::/0` in the IPv6 routing table (parameter `-A inet6`) that points to router (*gw* or gateway is an old synonym for router) `2001:db8:be:feed::11`. This command can be issued on `hostA` and router `r1` must have been configured with address `2001:db8:be:feed::11`. You can now add a default route on all virtual hosts.

You can verify that these routes have been inserted in the routing tables by inspecting them with the *ip(8)* or *route(8)* command.

The next step is to configure the routes on the three routers. For this, you need to first decide the paths that you want to use and make sure that all routers have a route towards at least the subnets attached to the endhosts.

A first approach is to use shortest path routing. In this case, you need to make sure that :

- `r1` has a route to `2001:db8:be:beef/64` via `r2`
- `r3` has a route to `2001:db8:be:beef/64` via `r2`
- `r2` has a route to `2001:db8:be:feed/64` via `r1`
- `r3` has a route to `2001:db8:be:feed/64` via `r1`
- `r1` has a route to `2001:db8:be:b00b/64` via `r3`
- `r2` has a route to `2001:db8:be:b00b/64` via `r3`

When you configure such a route, make sure that you use the correct IPv6 address of the gateway. For example, to configure the first route above on `r1`, you might issue a command such as :

```
/sbin/route -A inet6 add 2001:db8:be:feed/64 gw 2001:db8:bad:1212::22
```

Assuming that subnet `2001:db8:bad:1212/64` has been used on the link between `r1` and `r2` and that the address of `r2` on this subnet is `2001:db8:bad:1212::22`.

Note: Asymmetric paths

Note that when manually configuring the routes as above, nothing forces you to use symmetric routes. For example, the following paths could be configured in the network above.

- `r1` has a route to `2001:db8:be:beef/64` via `r2`
- `r3` has a route to `2001:db8:be:beef/64` via `r1`
- `r2` has a route to `2001:db8:be:feed/64` via `r3`
- `r3` has a route to `2001:db8:be:feed/64` via `r1`
- `r1` has a route to `2001:db8:be:b00b/64` via `r2`
- `r2` has a route to `2001:db8:be:b00b/64` via `r3`

Feel free to configure such paths in a different lab.

Before testing the lab, make sure that the three routers are configured to forward IPv6 packets, i.e. act as routers. By default, Linux virtual machines are configured as endhosts and do not forward IPv6 packets. You can change this configuration by issuing the following `sysctl` :

```
sysctl -w net.ipv6.conf.all.forwarding=1
```

Now, you are ready to test the correct operation of your emulated network. For this, you can use the two most common network debugging tools :

- *ping6(8)*

- *traceroute6(8)*

ping6(8) sends an ICMP Echo request to a given destination address. If *ping6(8)* succeeds, this indicates that both the forward and backward paths operate correctly. *traceroute6(8)* sends UDP segments with different HopLimit values to discover the routers on a path towards a given destination address. If *ping6(8)* and *traceroute6(8)* succeed between all pairs of hosts, your network is correctly configured. Otherwise, try to find the missing or buggy configuration and correct it.

Configuring a network manually takes some time and requires many commands. You probably do not want to issue all these commands each time you start a netkit lab. netkit helps you to automate the configuration of the virtual machines with two simple tools.

The first way to automate a lab is the startup script. A startup script is a simple shell script that is launched automatically by netkit once a virtual machine has booted. This script is named `machine.startup` where `machine` is the name of the virtual machine. It is placed in the directory that contains the `lab.conf` file.

The second way to automate a lab is by automatically copying files on the virtual machines. For each virtual machine, you can provide a hierarchy of directories and files that will be copied by netkit when the virtual machine starts. For example, if you want to place the `hosts` file automatically as `/etc/hosts` on virtual machine `hostA`, issue the following commands from the directory that contains `lab.conf`.

```
mkdir hostA
cd hostA
mkdir etc
cd etc
cp ../../hosts .
```

These commands create the hostA/etc directory and copy the hosts file that we created earlier at its final location. This file will be copied as `/etc/hosts` in the filesystem used by virtual machine `hostA`.

4.9.1 Assignment

ICMPv6 **RFC 4443**, the Internet Control Message Protocol, is a key companion to IPv6. ICMPv6 can report to the sender various types of errors that can occur during the transmission of a packet. *traceroute6(8)* exploits one of these messages to determine the path followed by packets towards a given destination.

To demonstrate your understanding of ICMPv6, prepare a lab with a few hosts and routers, prepare and test a scenario that uses a few commands that would cause a host or router to generate one of the following ICMPv6 error messages :

- *Destination Unreachable* (but not from a router directly connected to the source of the packet)
  - Code `0` : No route to destination
  - Code `3` : Address unreachable
  - Code `4` : Port unreachable
- *Packet Too Big* (with UDP segments and TCP segments)
- *Time Exceeded* message (but only Code `1` - Fragment reassembly time exceeded)

Provide in your report a short text that explains why the ICMPv6 error message is generated and show a tcpdump output containing this message.

4.10 IP Address Assignment Methods and Intradomain Routing

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues/new

This lab allows you to experiment with the main IPv6 address techniques and illustrates the operation of two intradomain routing protocols (OSPFv3 and RIP). For this, you will use 4

netkit labs: /netkit/netkit-lab_slaac.zip, /netkit/netkit-lab_dhcpv6.zip, /netkit/netkit-lab_ospfv3.zip and /netkit/netkit-lab_rip.zip.

You will need to observe into the packets exchanged in the emulated network to understand the operation of the involved protocols and mechanisms. For this, you can use packet capture tools (such as tcpdump or wireshark).

Some tcpdump options (for more details and options, check man tcpdump):

-i allows to choose a specific interface (any for all interfaces).

-v (or -vv) provides a verbose more with more details on the contents of each packet

-s allows to capture the entire packets (not only the first 68 bytes).

Wireshark also allows to capture packets but provides a graphical user interface that is useful to analyse long packet traces. Wireshark is installed on the Linux machines in the lab and can be downloaded from http://www.wireshark.org

---

Note: If you have root access to the machine, you can also use wireshark to visualize traffic captured within a netkit lab :

When you have launched the lab, you can access to your $HOME directory or the lab directory (in the host machine) from a netkit virtual machine. These directories are located in /hosthome and /hostlab in netkit . Go into /hostlab :

```
cd /hostlab
```

Now you can launch a tcpdump capture and save the captured traffic on a file, in the hostlab (or hosthome) directory (option -w). This allows you to start a capture from this file with wireshark.

```
tcpdump -n -i IF -w aaa.pcap &
```

where aaa.out is the output file, IF the interface we want to listen on (use any for all interfaces) and we add the & symbol to run the sniffer in the background, so we can continue to work in the netkit shell.

You can then launch wireshark on your computer with the input file aaa.out :

```
wireshark -k -i<(tail -f aaa.pcap)&
```

Note that this is *not* possible if you are a non-privileged user in the machine. Therefore, if you are running netkit in the INGI lab you should use tcpdump to look at on-the-fly traffic.

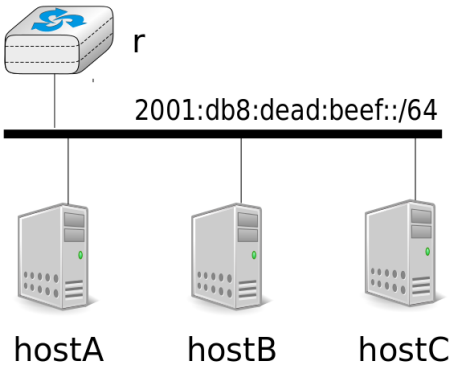
4.10.1 IPv6 Address Assignment Methods

The SLAAC lab

The first lab (/netkit/netkit-lab_slaac.zip) illustrates the operation of the Stateless Address Autoconfiguration (SLAAC) in IPv6. This mechanism allows hosts connecting to an IPv6 link to dynamically acquire a link-local address and, in case a router is present, to be assigned a global-scope IPv6. The link-local address enables a host to exchange packets with any other host in the same IPv6 subnet. Global-scope IPv6 address enables to communicate outside the local subnet.

Start the /netkit/netkit-lab_slaac.zip lab in netkit. This lab consists of 4 virtual machines: 3 hosts (hostA, hostB and hostC) and router r, connected as shown in the figure below

Router r uses IPv6 address 2001:db8:dead:beef::11/64 and is configured to send periodically Router Advertisements and reply to Router Sollicitations, via the radvd daemon. You can stop/restart this daemon by using the /etc/init.d/radvd script. The configuration of this daemon is detailed in file /etc/radvdv.conf. The configuration used for the lab is provided below.




```
interface eth0 {
    AdvSendAdvert on;
    MinRtrAdvInterval 3;
    MaxRtrAdvInterval 10;
    prefix 2001:db8:dead:beef::/64
    {
        AdvOnLink on;
        AdvAutonomous on;
        AdvRouterAddr on;
    };
};
```

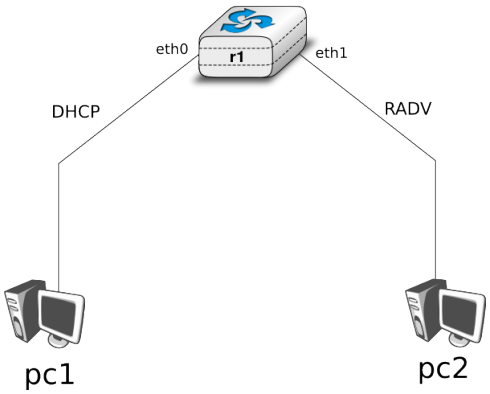
Unlike previous labs, IPv6 addresses are not configured with *ifconfig(8)* on the virtual hosts. These addresses will be obtained dynamically. All host interfaces are `down` at the startup, so you need to activate them with `ifconfig eth0 up` when you are ready to monitor the address assignment process. Use `tcpdump` on a host (for instance, `hostA`) to capture the packets in the link (or `wireshark` if you have root access to your machine). Look at the exchanged packets when the interfaces are activated.

1. How are global-scope addresses assigned ? Describe the observed router discovery process.
2. How do hosts acquire their link-local addresses ? How do these link-local addresses look like ? For this question, you can set an interface down and up to observe the acquisition of a new link-local address without restarting the lab.

The DHCPv6 lab

The goal of the `/netkit/netkit-lab_dhcpv6.zip` lab is to observe the operation of DHCPv6. In this lab, you will work on an emulated network with one router (`r1`) running a DHCPv6 server and 2 emulated hosts (`pc1` and `pc2`) running DHCPv6 clients.

Here is the topology of the network:



To use DHCPv6, these virtual machines uses the dibbler daemon.

When the lab is launched, run the dibbler server daemon on the router, then run the client daemon on the hosts (`pc1` and `pc2`). Startup scripts are provided for each of these daemons.

```
/etc/init.d/dibbler-server start
/etc/init.d/dibbler-client start
```

Based on the observed packets, try to figure out:

1. Why the server sends periodically multicast packets ?
2. What kind of packets are exchanged between the server and the clients? (You should identify 4 different types)

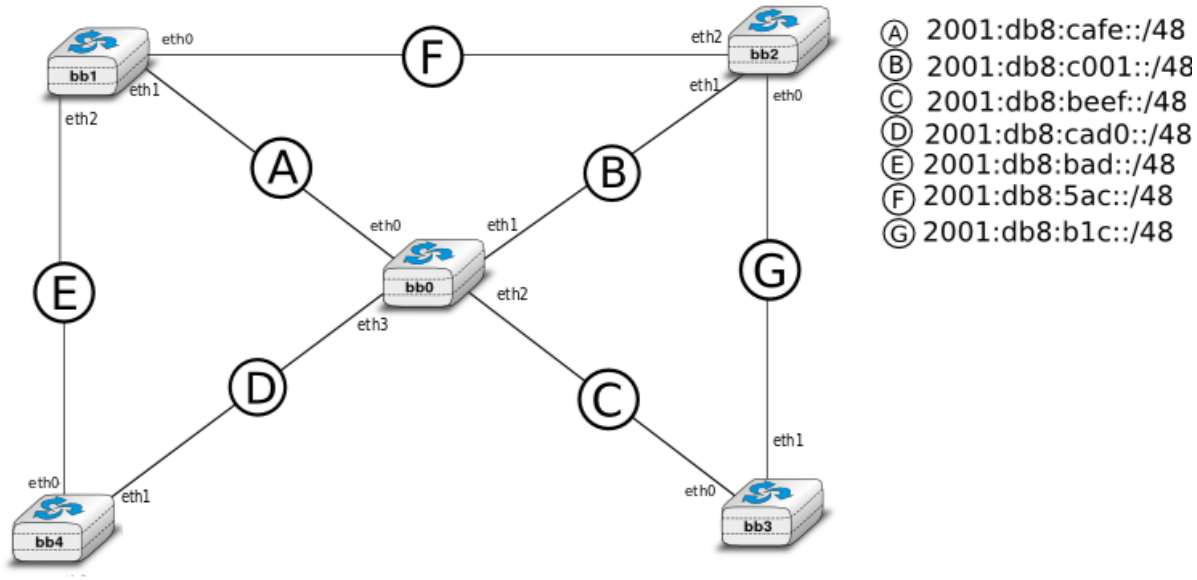
4.10.2 Intradomain routing

We focus on two intradomain routing protocols : RIP and OSPFv3.

The OSPFv3 lab

The Open Shortest Path First (OSPF) protocol is a link-state routing protocol, widely used in today's Internet for intra-domain routing. OSPFv3 (OSPF for IPv6) is specified in **RFC 5340**. OSPF control packets are sent directly over the IP layer, with protocol number 89, so they can be filtered with `tcpdump` with the command `tcpdump ip[9]==89`.

In the `/netkit/netkit-lab_ospfv3.zip` lab, you will work on a network with routers that use OSPFv3 to compute their routing tables. Here is the topology of the network:



To use OSPF, these routers uses daemons called `zebra` and `ospf6d` . Start the lab. Note that, if you try to `ping6` from a router to a non-adjacent one, you will see a destination unreachable. This is because the OSPF deamon is not launched yet. It is interesting to run `tcpdump` (in the background) on at least one machine, to capture the exchanged packets.

You should launch the `ospf6d` daemon on every router, looking at how every new OSPF-enabled router impacts in the monitored traffic. Launch first the daemon on `bb1`. To do that enter the following command line in the `bb1` terminal :

```
/etc/init.d/zebra start
```

Then launch the daemon on `bb2`, ...

1. Which packets have you observed in the packet traces ?
2. Were the routing tables modified ? If so, how ?

Launch now the deamon on the others routers while looking at the packets captured.

3. Perform traceroutes from/to different interfaces. Think about the path the traceroute is expected to take, and the path ICMP replies are expected to take. Does the *traceroute6(8)* confirm your expectations?

Now we will access the `ospf6d` daemon. This will help us to see the OSPF link-state database (LSDB), neighbors and routes.

In netkit, type :

```
telnet ::1 ospf6d
```

Note: Reminder: `::1` is the IPv6 address for localhost. `ospf6d` is the name of the daemon[#fservices]_ that runs OSPF in our router.

The daemon asks for a password. Use the default one, `zebra`.

Now you can interact with the OSPF daemon and observe its current state and the datastructures that it maintains. Some useful commands are :

```
show ipv6 ospf6 database
show ipv6 ospf6 neighbor
show ipv6 ospf6 route
show ipv6 ospf6 interface
exit
```

4. What is the information returned by each of the above commands ?
5. Is the LSDB the same for all routers ? Should it be ?

Now it is time to play with the topology. You can request the shortest path tree computed by a router (and monitor how it changes) with the command `show ipv6 ospf6 spf tree` when connected via `telnet` to the `ospf6d` daemon.

5. Try to disable some link sand observe what is happening. You can disable a link with the *ifconfig(8)* command :

   ```
   ifconfig IF down
   ```

   where `IF` is the name of your interface. (If you want to set up an interface down, remember that manually assigned static IP addresses vanish when the interface is down, so you need to assign them manually when you set it up again)

6. When you are in the daemon (`telnet ::1 ospf6d`) , change the link cost and try some `traceroute`. Below, the line you should enter in your console:

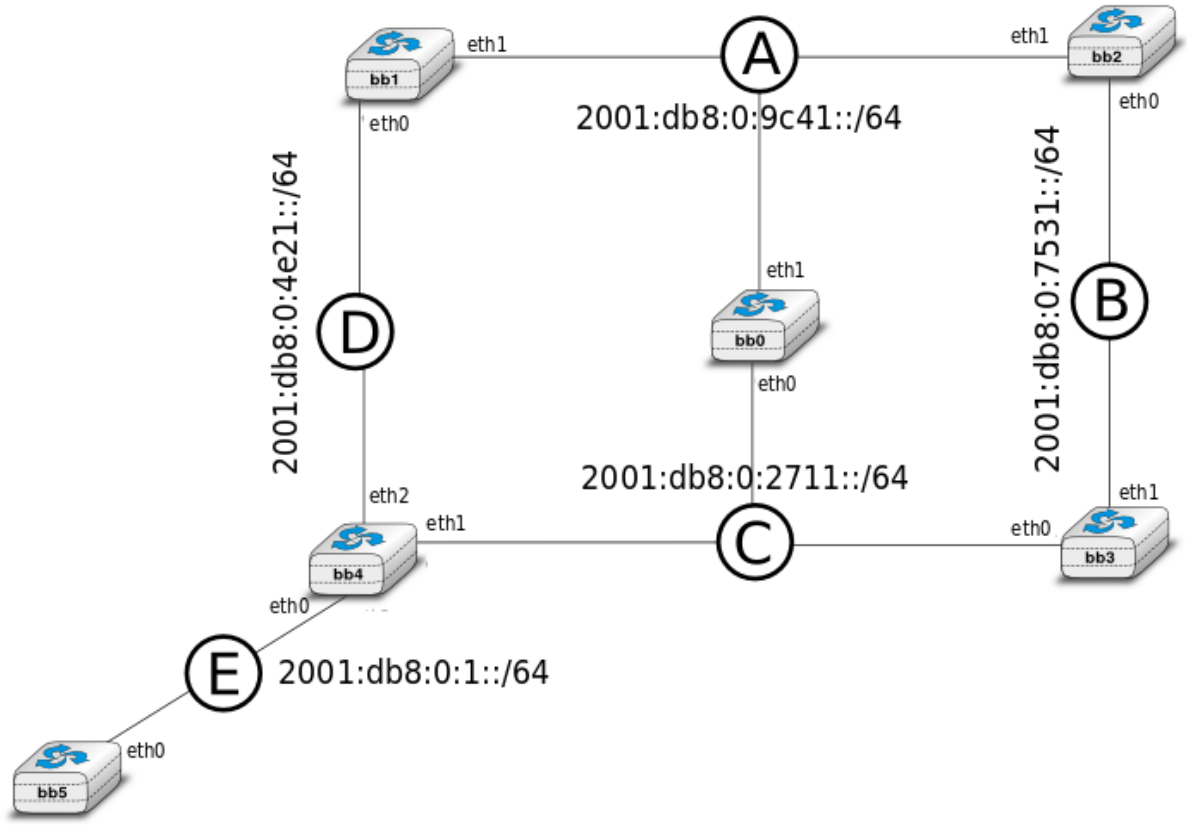
   ```
   telnet ::1 ospf6d
   zebra
   enable
   configure terminal
   router ospf6
   interface IF
   ipv6 ospf6 cost X
   ```

   where `IF` is the interface and `X` the new cost.

The RIP lab

The Routing Information Protocol (RIP) is a popular distance-vector protocol. It was widely used for intra-domain routing in the Internet, before being replaced by link-state protocols such as OSPF or IS-IS. RIPng (for IPv6) is specified in **RFC 2080**. RIPng packets are UDP packets and are sent to port 521, so you can filter RIPng packets with this command: `tcpdump udp and port 521`.

In the `/netkit/netkit-lab_rip.zip` lab, you will work on a network composed of routers that use RIPng to build their routing tables. Here is the topology of the network:



To use RIP, these routers use daemons called `zebra` and `ripngd` .

After launching the lab, use `tcpdump` at the machine `sniffer`. This machine has 5 interfaces, each of them connected to a different network link (see the topology description in the file `lab.conf` and the interfaces configured in the `sniffer.startup` file).

First of all, launch the `ripngd` and `zebra` daemons. To do that, type on each router the command :

```
/etc/init.d/zebra start
```

Observe the evolution of the routing table of one router. After a while, all destinations are available. Why it is not instantaneous?

1. Check routing tables. Are they updated ?
2. Sniff the RIP packets using `tcpdump` and observe them. Is this consistent with what you expected ?

Now it is time to modify the topology.

3. Try to make some links fail and observe what is happening. You can do that by stoping one interface on a router :

```
ifconfig IF down
```

where `IF` is the name of your interface.

4. Observe what is happening. Is the network recovering fast ? Why ?

4.10.3 Assignment

Some networks still rely on manually configured static routes. Static routing provides some flexibility compared to distance vector routing, but suffers from two important problems. First it does not react to failure. Second, configuration errors can cause forwarding loops and blackholes.

`/netkit/netkit-lab_err1.zip`, `/netkit/netkit-lab_err2.zip` and `/netkit/netkit-lab_err3.zip` are three netkit labs describing networks with four routers (`r1` to `r4`). The routes in each network are configured with static routes, but routing is not correct. You can extract the netkit labs from the corresponding ZIP files (`laberr1.zip` for `laberr1`, and so on).

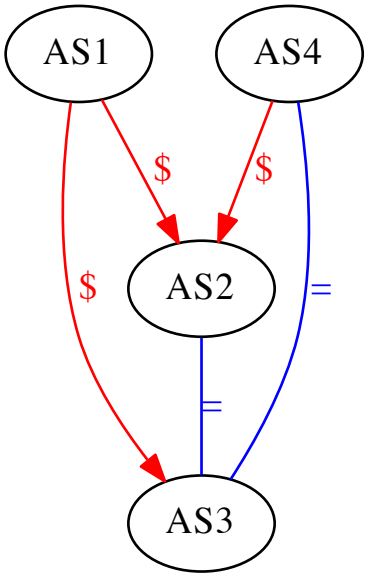
For each lab, find the errors by using *`traceroute6(8)`* or observing the routing tables of the routers. Then, propose a fix for each problem so that every router can reach successfully (e.g. with `ping6`) every other router in the network.

4.11 Inter-domain routing and BGP

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues/new

4.11.1 Exercises

1. Consider the interdomain topology shown in the figure below.

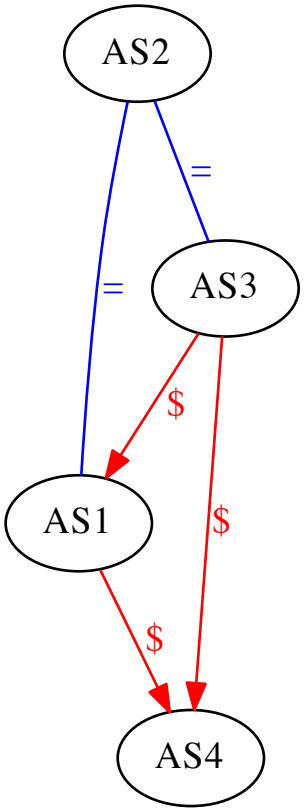


In this network, what are the paths :

- from *AS1* to *AS4*

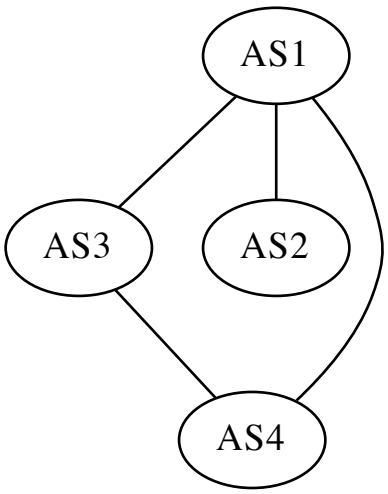
- from *AS4* to *AS2*
- from *AS4* to *AS1*

2. Consider the interdomain topology shown in the figure below. Assuming, that *AS1* advertises prefix `2001:db8:1::/48`, *AS2* prefix `2001:db8:2::/48`, ... compute the routing tables of the different ASes.



Are all ASes capable of reaching all the other ASes in this simple Internet ?

3. The interdomain topology below is composed of four domains. For this exercise, we assume that there are no routing policies, i.e. each domain advertise all its best paths to its peers. We focus on the prefix *p* advertised by *AS1*.



Assume that the BGP sessions are activated in the following order :

- *AS1-AS2*
- *AS2-AS1*
- *AS3-AS4*
- *AS1-AS3*
- *AS1-AS4*

At each step of this activation sequence, show the BGP messages for prefix *p* that are exchanged and provide the BGP routing table of the different ASes. Assume that BGP always prefers the short AS-Path.

Once the interdomain network has fully converged, analyze the consequence of a failure of the following BGP sessions on the routes towards prefix *p* :

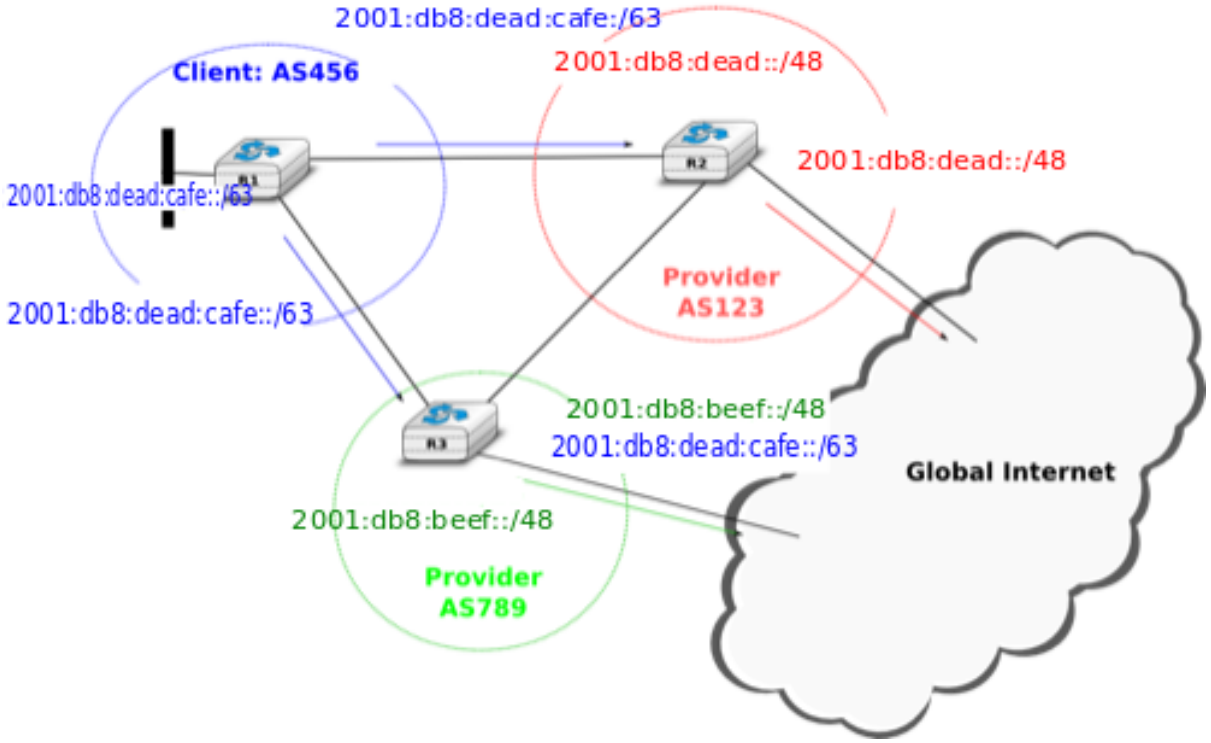
- *AS1-AS2*
- *AS1-AS4*

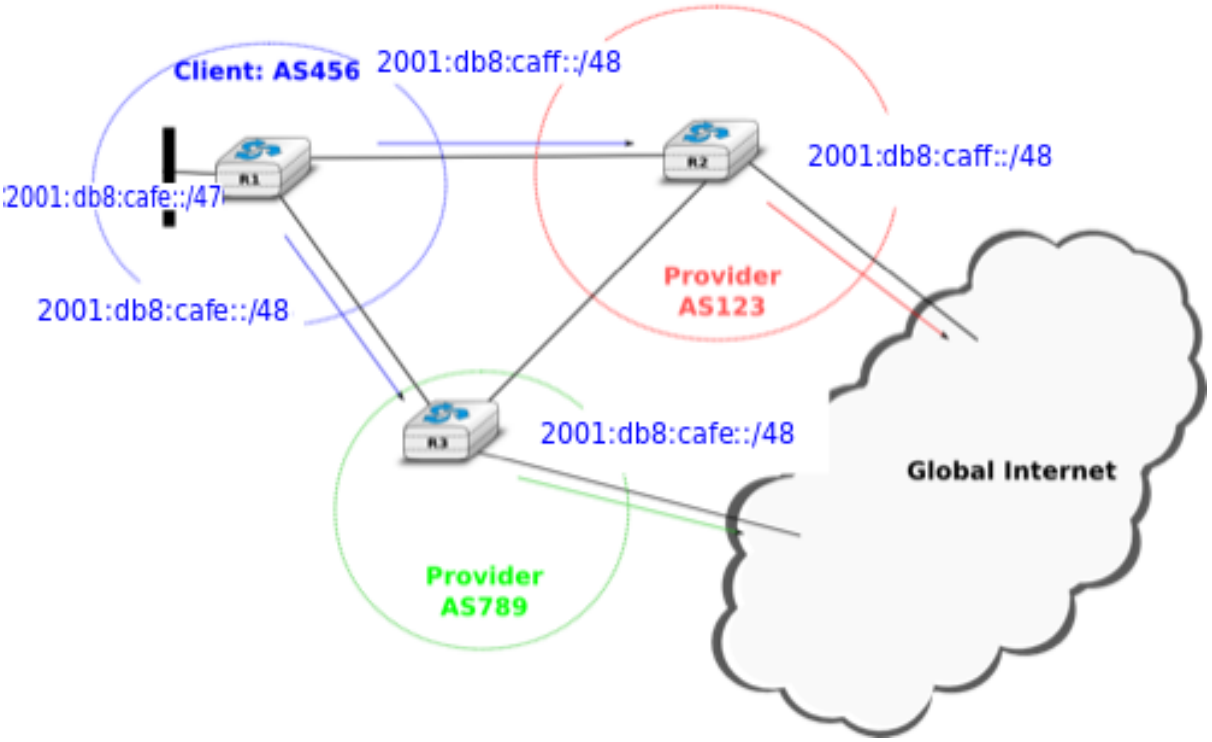
4. Consider the network below in which a stub domain, *AS456*, is connected to two providers *AS123* and *AS789*. *AS456* advertises its prefix to both its providers. On the other hand, *AS123* advertises *2001:db8:dead://48* while *AS789* advertises *2001:db8:beef::/48* and *2001:db8:dead:cafe::/63*. Via which provider will the packets destined to `2001:db8:dead:cafe::1` will be received by *AS456* ?

   Should *AS123* change its configuration ?

5. Consider that the AS stub (*AS456*) shown in the figure below decides to advertise two `/48` prefixes instead of its allocated `/47` prefix.

   - Via which provider does *AS456* receive the packets destined to `2001:db8:caff::bb` and `2001:db8:cafe::aa` ?
   - How is the reachability of these addresses affected when link *R1-R3* fails ?
   - Propose a configuration on R1 that achieves the same objective as the one shown in the figure but also preserves the reachability of all IP addresses inside *AS456* if one of *AS456*'s interdomain links fails.


2001:db8:dead:cafe::/63
Client: AS456
2001:db8:dead::/48
R1
R2
2001:db8:dead:cafe::/63
2001:db8:dead::/48
Provider
AS123
2001:db8:dead:cafe::/63
R3
2001:db8:beef::/48
2001:db8:dead:cafe::/63
Global Internet
2001:db8:beef::/48
Provider
AS789


Client: AS456
2001:db8:caff::/48
R1
R2
2001:db8:caff::/48
2001:db8:cafe::/47
Provider
AS123
2001:db8:cafe::/48
R3
2001:db8:cafe::/48
Global Internet
Provider
AS789

6. Consider the network shown in the figure below. In this network, each AS contains a single BGP router. Assume that *R1* advertises a single prefix. *R1* receives a lot of packets from *R9*. Without any help from *R2*, *R9* or *R4*, how could *R1* configure its BGP advertisement such that it receives the packets from *R9* via *R3* ? What happens when a link fails ?

7. Consider the network shown in the figure below where R1 advertises a single prefix. In this network, the link between R1 and R2 is considered as a backup link. It should only be used only when the primary link (R1-R4) fails.
   - Can you implement this in *R2* ? How ?
   - Assuming that *R1-R2* is a backup link, what are the paths used by all routers to reach *R1* ?
   - Assume now that the link *R1-R4* fails. Which BGP messages are exchanged and what are now the paths used to reach *R1* ?
   - Link *R1 - R4* comes back. Which BGP messages are exchanged and what do the paths used to reach *R1* become ?

4.11.2 Netkit BGP lab

This lab, `/netkit/netkit-lab_bgp.zip`, allows you to experiment with the BGP operation. The simulated internetwork consists of 8 BGP routers, each one corresponding to a different Autonomous System (AS). They have the peering relations shown in the figure:

To run BGP, these routers uses daemons called `zebra` and `bgpd` .

You can launch the lab using `lstart` in the lab folder. For monitoring traffic, you can use `tcpdump` and `wireshark`.

In this lab, router `r9` announces via BGP an IPv6 prefix throughout the network. You can observe in the routing tables of the other routers that there are entries for the local prefixes and for the prefix announced by `r9`, in which the nexthop is indicated. But routers have no addional information about prefixes from other routers.

You can have more advanced informations about how BGP runs on the routers by accessing to the `bgpd` daemon via `telnet`:

```
telnet localhost bgpd
```

The password is `zebra`, as usual.

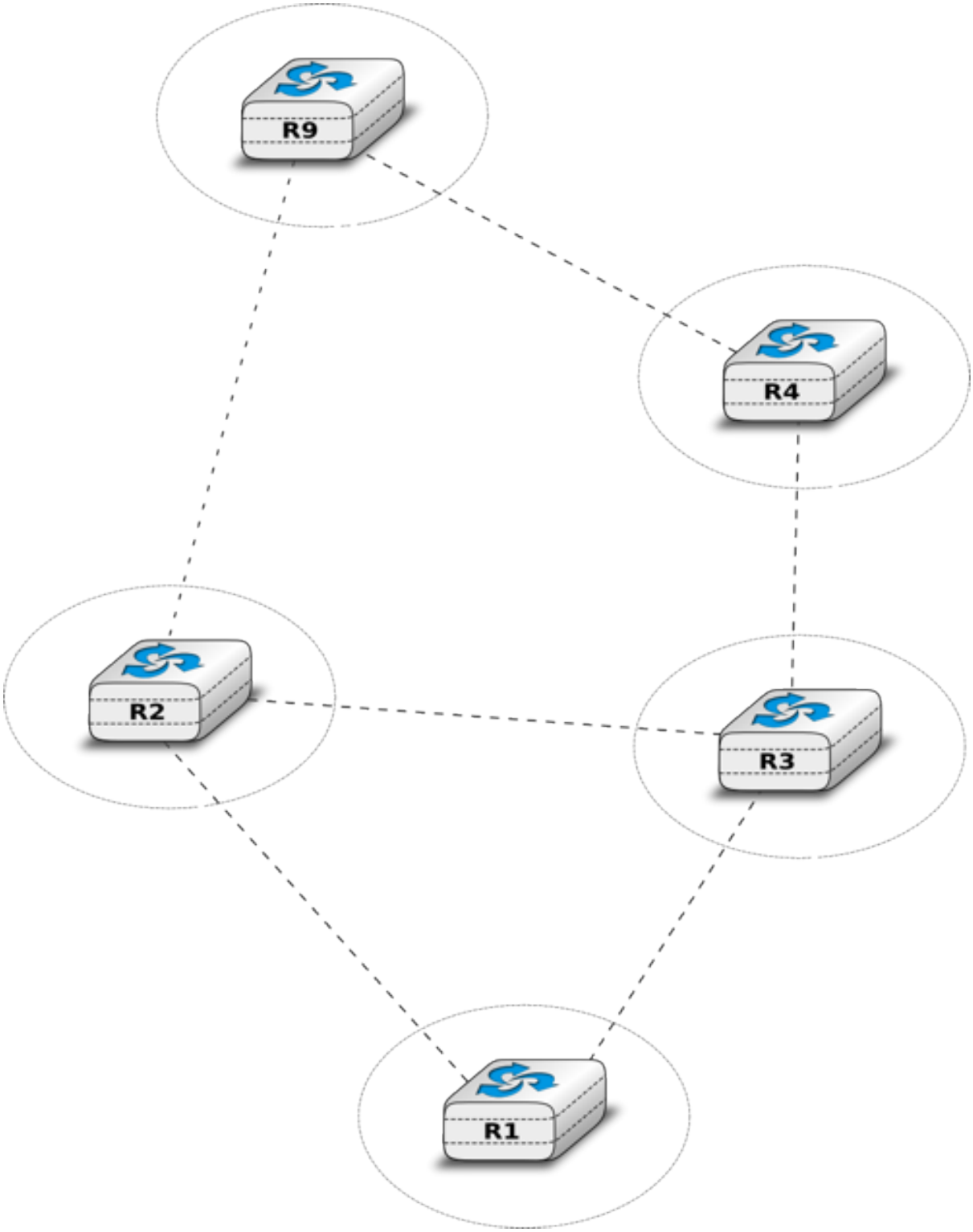
In the `telnet` terminal you can use:

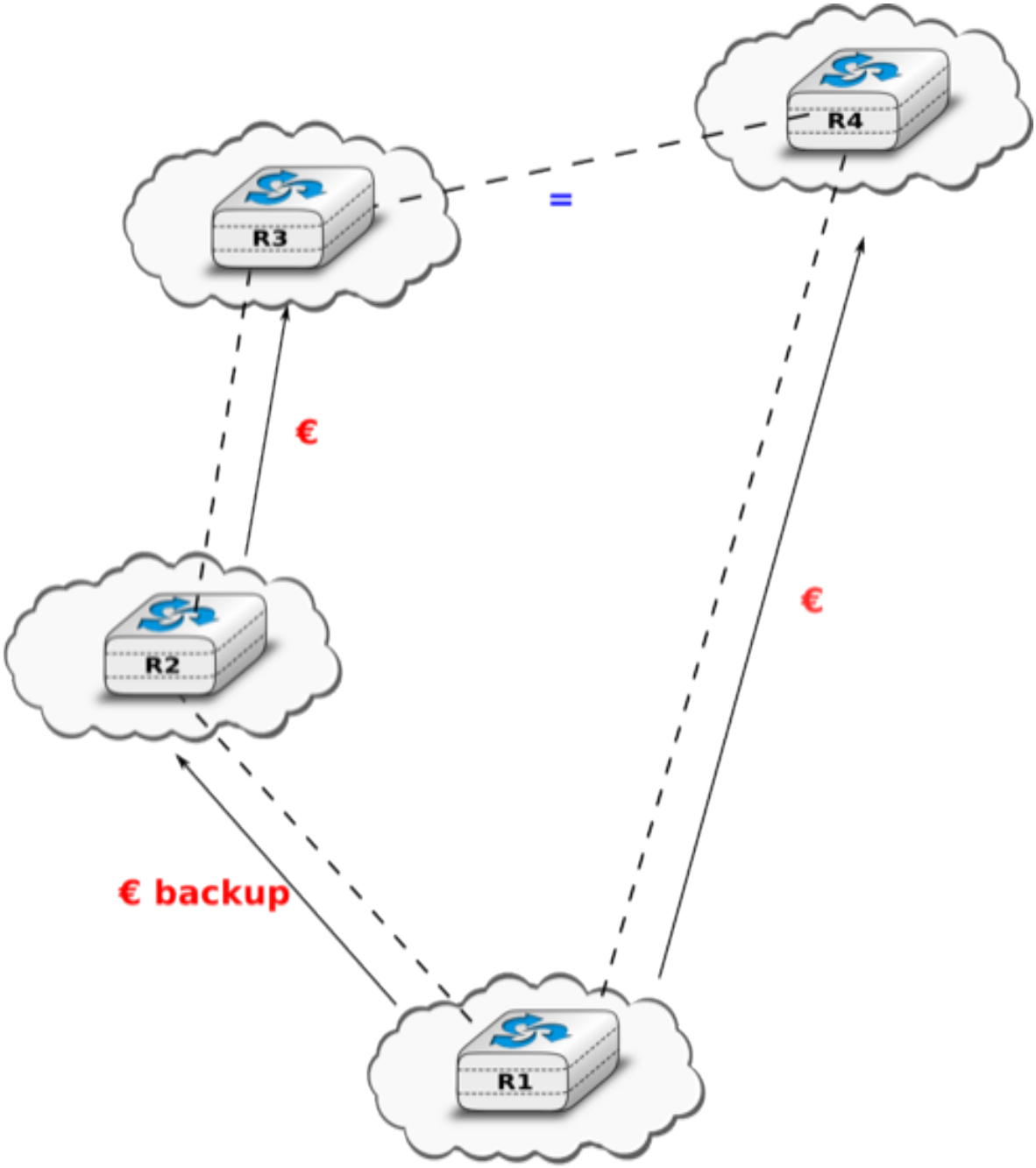
```
show ipv6 bgp summary
show ip bgp neighbors
show ip community-list
```

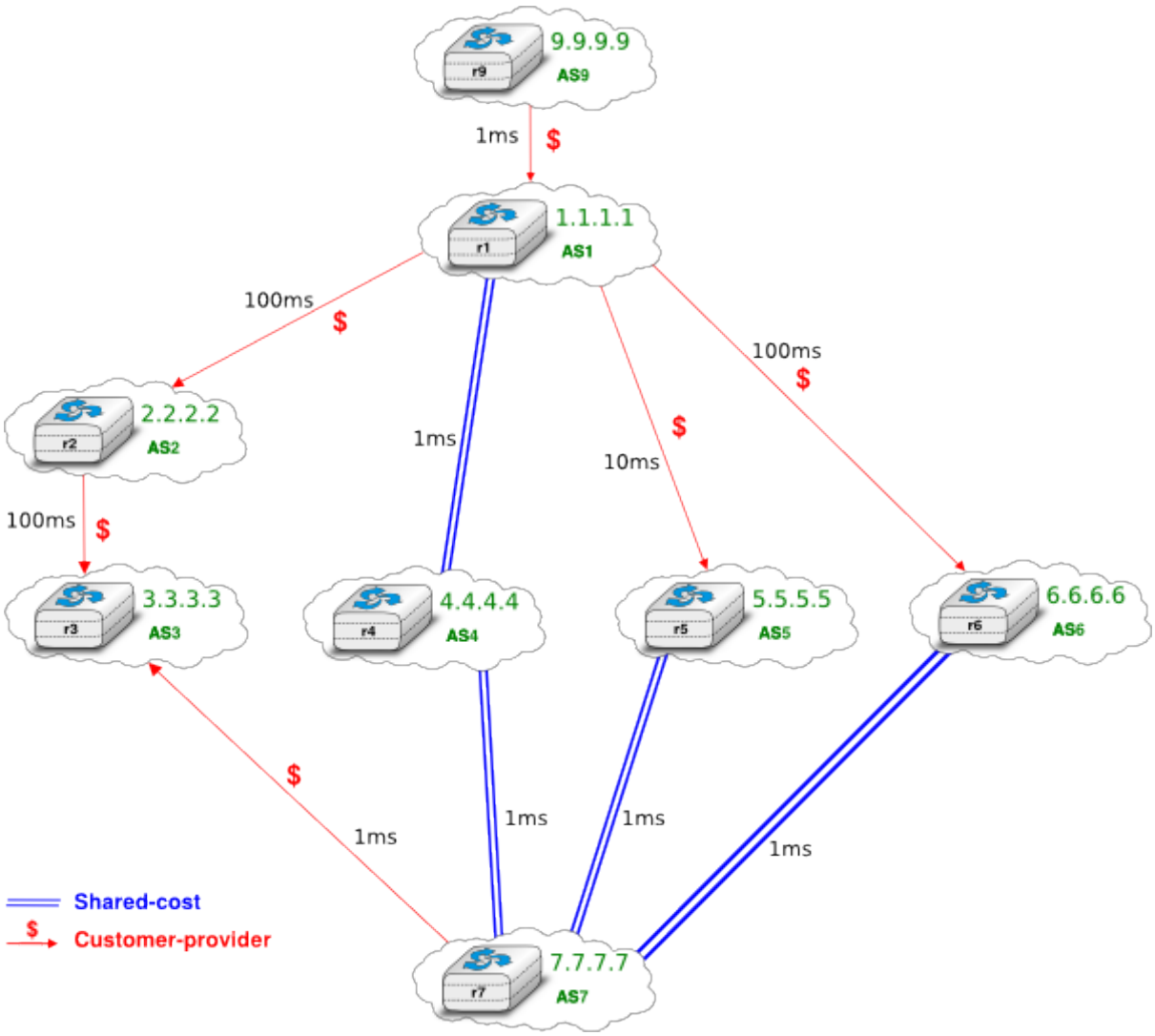
to get more infos. (Note that neighbors and community list queries use the *ip* command instead of the *ipv6* command). See the quagga manual for `bgpd` for a more complete description of available commands.

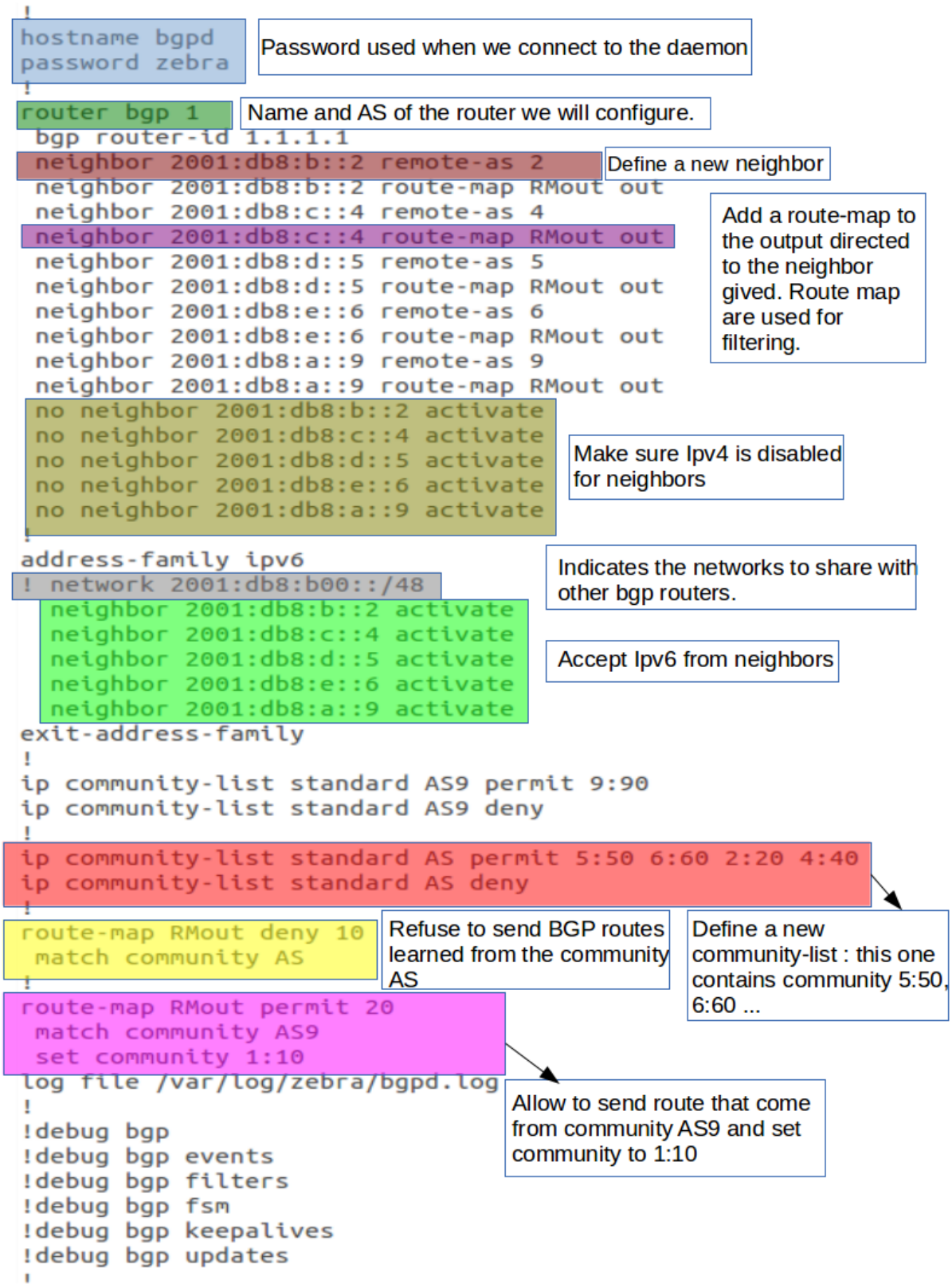
You can find the configuration files of the running daemons in the routers' folders. For instance, consider router `r1`. You can find 3 configuration files in `lab/r1/etc/quagga`:

- The first one is `daemons`. This file contains informations about which daemon should be started on our router.
- The second one is `zebra.conf`. This file contains the password that we use to connect to the zebra daemon when we are on the router. (The password asked when accessing `telnet localhost zebra`)
- The third one is `bgpd.conf`. This is the configuration file of our bgpd daemon. The following picture details the meaning of Let's see what all these lines means.


R9
R4
R2
R3
R1


R4
R3
=
€
R2
€
€ backup
R1


9.9.9.9
AS9
1ms
$
1.1.1.1
AS1
100ms
$
100ms
$
2.2.2.2
AS2
1ms
10ms
$
100ms
$
3.3.3.3
AS3
4.4.4.4
AS4
5.5.5.5
AS5
6.6.6.6
AS6
$
1ms
1ms
1ms
1ms
Shared-cost
$
Customer-provider
7.7.7.7
AS7


!
hostname bgpd
password zebra
Password used when we connect to the daemon
!
router bgp 1
Name and AS of the router we will configure.
bgp router-id 1.1.1.1
neighbor 2001:db8:b::2 remote-as 2
Define a new neighbor
neighbor 2001:db8:b::2 route-map RMout out
neighbor 2001:db8:c::4 remote-as 4
neighbor 2001:db8:c::4 route-map RMout out
neighbor 2001:db8:d::5 remote-as 5
neighbor 2001:db8:d::5 route-map RMout out
neighbor 2001:db8:e::6 remote-as 6
neighbor 2001:db8:e::6 route-map RMout out
neighbor 2001:db8:a::9 remote-as 9
neighbor 2001:db8:a::9 route-map RMout out
Add a route-map to the output directed to the neighbor gived. Route map are used for filtering.
no neighbor 2001:db8:b::2 activate
no neighbor 2001:db8:c::4 activate
no neighbor 2001:db8:d::5 activate
no neighbor 2001:db8:e::6 activate
no neighbor 2001:db8:a::9 activate
Make sure Ipv4 is disabled for neighbors
!
address-family ipv6
! network 2001:db8:b00::/48
Indicates the networks to share with other bgp routers.
neighbor 2001:db8:b::2 activate
neighbor 2001:db8:c::4 activate
neighbor 2001:db8:d::5 activate
neighbor 2001:db8:e::6 activate
neighbor 2001:db8:a::9 activate
Accept Ipv6 from neighbors
exit-address-family
!
ip community-list standard AS9 permit 9:90
ip community-list standard AS9 deny
!
ip community-list standard AS permit 5:50 6:60 2:20 4:40
ip community-list standard AS deny
Define a new community-list : this one contains community 5:50, 6:60 ...
!
route-map RMout deny 10
match community AS
Refuse to send BGP routes learned from the community AS
!
route-map RMout permit 20
match community AS9
set community 1:10
Allow to send route that come from community AS9 and set community to 1:10
log file /var/log/zebra/bgpd.log
!
!debug bgp
!debug bgp events
!debug bgp filters
!debug bgp fsm
!debug bgp keepalives
!debug bgp updates
!

With this in mind, you are able to play with the topology and even create new routers that use BGP. Try some different configurations, try to change how the filters work and observe what happens. On the original lab, for instance, you can cause a failure on the *AS9-AS1* link (with the command `ifconfig ... down`). Observe which BGP messages are exchanged and how the state of router `r7` changes. What are your expectations? Are your observations consistent with what you expected ?

4.12 Local Area Networks: The Spanning Tree Protocol and Virtual LANs

Warning: This is an unpolished draft of the second edition of this ebook. If you find any error or have suggestions to improve the text, please create an issue via https://github.com/obonaventure/cnp3/issues?milestone=6

4.12.1 Exercises

1. Consider the switched network shown in Fig. 1. What is the spanning tree that will be computed by 802.1d in this network assuming that all links have a unit cost ? Indicate the state of each port.

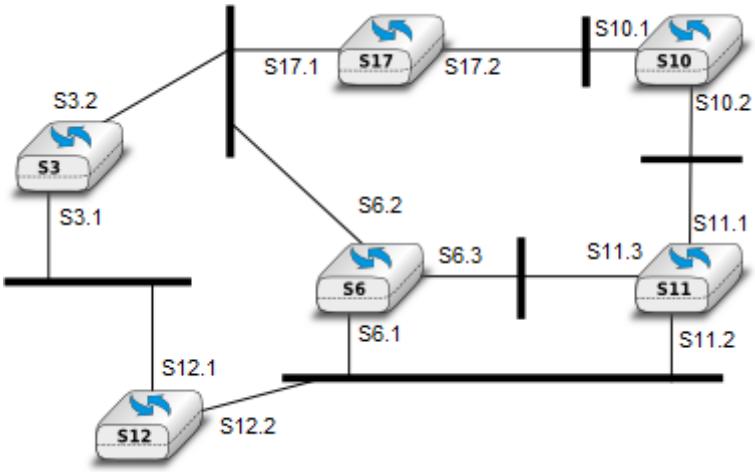


Figure 4.12: Fig. 1. A small network composed of Ethernet switches

2. Consider the switched network shown in Fig. 1. In this network, assume that the LAN between switches S3 and S12 fails. How should the switches update their port/address tables after the link failure ?
3. Many enterprise networks are organized with a set of backbone devices interconnected by using a full mesh of links as shown in Fig.2. In this network, what are the benefits and drawbacks of using Ethernet switches and IP routers running OSPF ?
4. In the network depicted in Fig. 3, the host *H0* performs a traceroute toward its peer *H1* (designated by its name) through a network composed of switches and routers. Explain precisely the frames, packets, and segments exchanged since the network was turned on. You may assign addresses if you need to.
5. In the network represented in Fig. 4, can the host *H0* communicate with *H1* and vice-versa? Explain. Add whatever you need in the network to allow them to communicate.

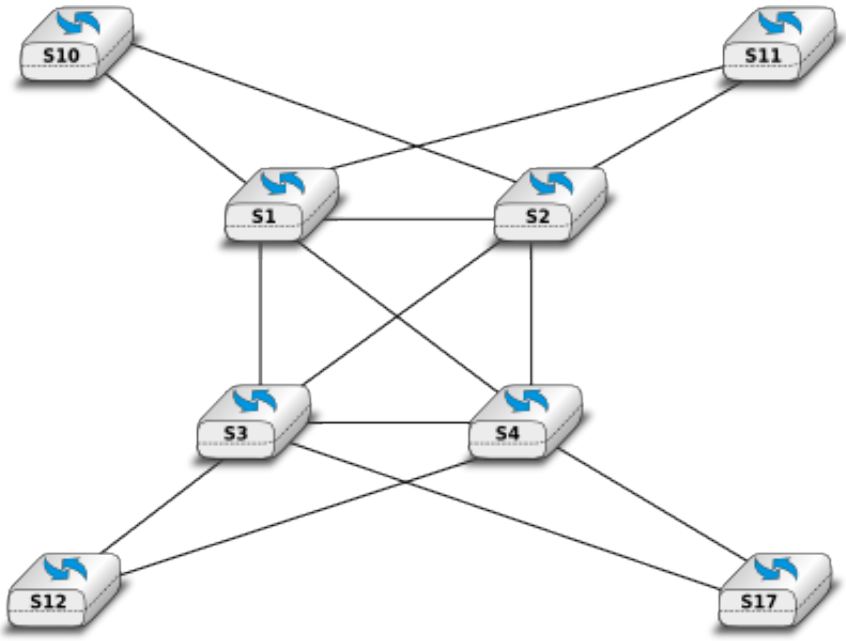


Figure 4.13: Fig. 2. A typical enterprise backbone network

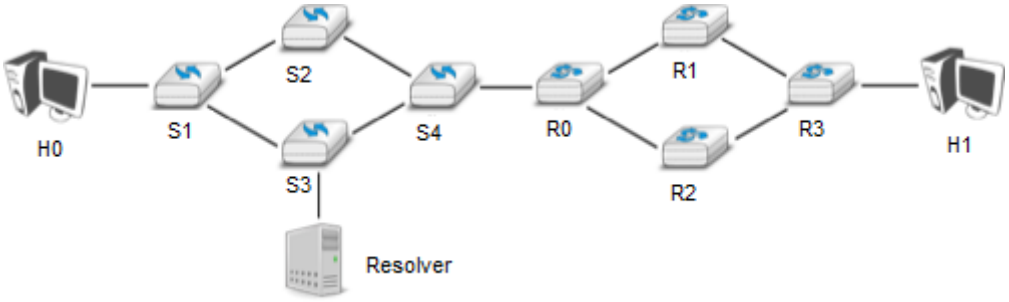


Figure 4.14: Fig. 3. Host *H0* performs a traceroute towards its peer *H1* through a network composed of switches and routers

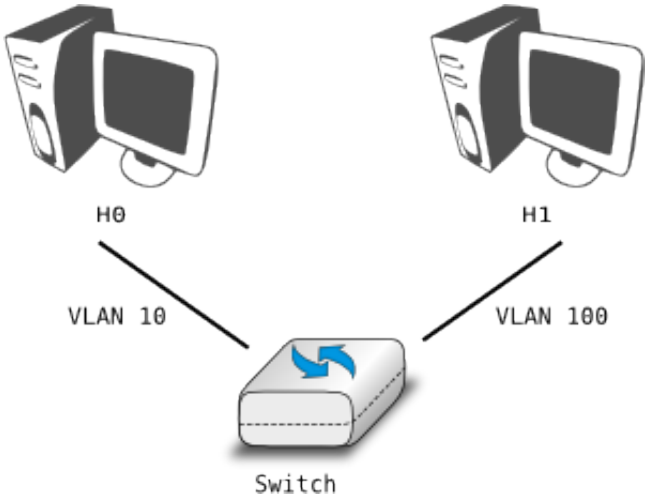
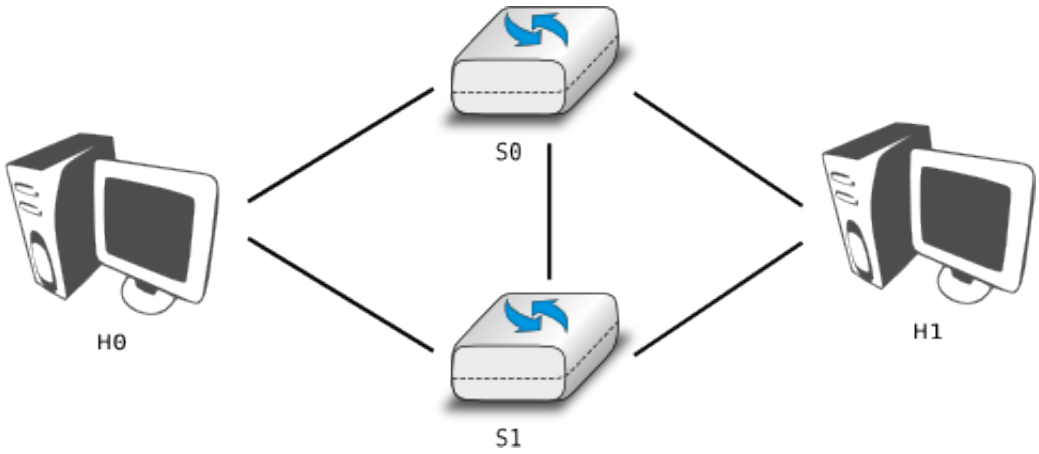


Figure 4.15: Fig. 4. Can *H0* and *H1* communicate ?

6. Consider the network depicted in Fig.5. Both of the hosts *H0* and *H1* have two interfaces: one connected to the switch *S0* and the other one to the switch *S1*. Will the link between *S0* and *S1* ever be used? If so, under which assumptions? Provide a comprehensive answer.

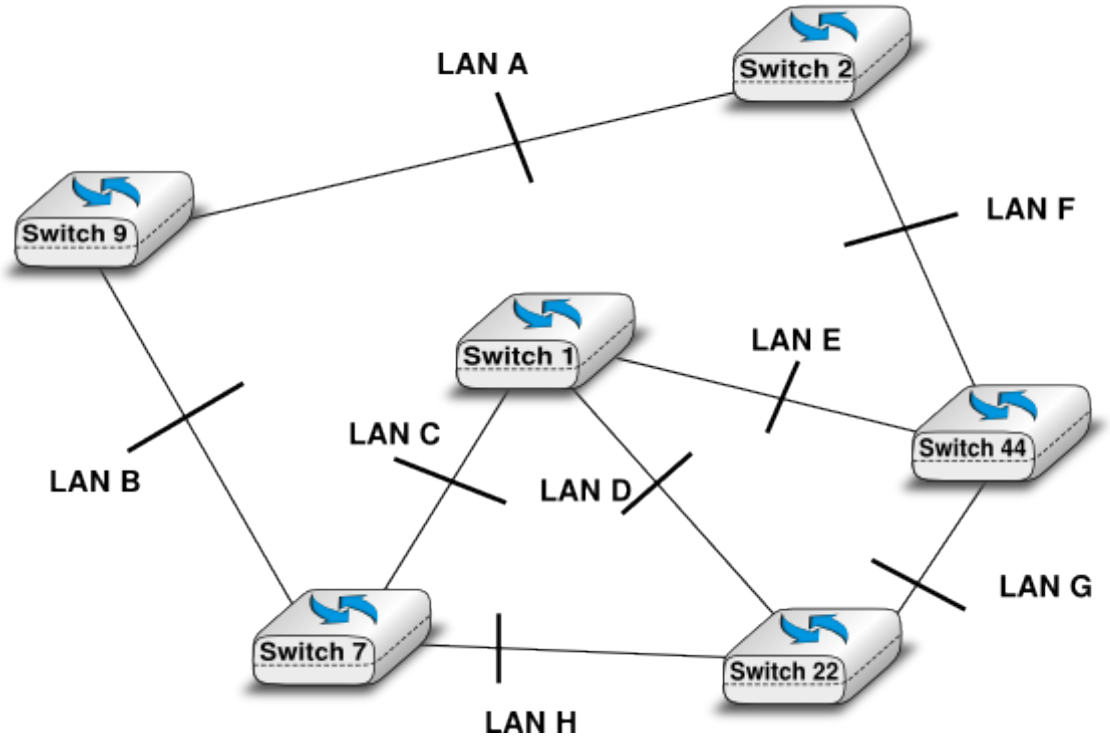


7. Most commercial Ethernet switches are able to run the Spanning tree protocol independently on each VLAN. What are the benefits of using per-VLAN spanning trees ?

4.12.2 Netkit STP lab

In the lab `lab_stp` (`/netkit/netkit-lab_stp.zip`), you can explore the behavior of a network with switches that use the Spanning Tree Protocol (STP). This protocol allows switches to automatically disable ports on Ethernet switches to ensure that the network does not contain any cycle that could cause frames to loop forever.

Here is the topology of the network:



To use STP, these switches uses `brctl`, a tool that allows to configure devices as Ethernet bridges and build the spanning tree.

For this lab, you can use the `wireshark` or `tcpdump` as packet sniffers.

To launch the lab you have to go in the directory of the lab and launch it with netkit using command `lstart` (you can add the option `-f` for a quick launch).

You can see that the 6 machines are launched. For the moment, no one of them runs STP. You will run it (via the `brctl` command) on two routers and activate wireshark on one of them as explained above. To activate the STP on one switch type in his terminal:

```
brctl stp br0 on
ifconfig br0 up
```

With these two routers you can see what messages are exchanged for the root bridge election. You can see the state of a bridge by typing :

```
brctl showstp br0
```

This command brings information about the designated root of the tree, the root port of the switch and the cost to the root switch.

Now, you can launch some other switches. By doing that you change the topology. With `wireshark` you can observe the packets of the spanning tree protocol that are exchanged. The switches already launched will generate a "topology change notification", then others switches will acknowlegde these changes.

When all switches are launched, you can look at the bridge state of each switches:

```
brctl showstp br0
```

You can see wich ports are in blocking state, wich are in forwarding state.

You can also look at the port-station table by entering :

```
brctl showmacs br0
```

You can make some links fail and observe what is happening. You can do that by stoping one interface on a switch or the entire bridge (if=br0) :

```
ifconfig IF down
```

where `IF` is the name of your interface.

CHAPTER 5

Appendices

5.1 Glossary

AIMD Additive Increase, Multiplicative Decrease. A rate adaption algorithm used notably by TCP where a host additively increases its transmission rate when the network is not congested and multiplicatively decreases when congested is detected.

anycast a transmission mode where an information is sent from one source to *one* receiver that belongs to a specified group

API Application Programming Interface

ARP The Address Resolution Protocol is a protocol used by IPv4 devices to obtain the datalink layer address that corresponds to an IPv4 address on the local area network. ARP is defined in **RFC 826**

ARPANET The Advanced Research Project Agency (ARPA) Network is a network that was built by network scientists in USA with funding from the ARPA of the US Ministry of Defense. ARPANET is considered as the grandfather of today's Internet.

ascii The American Standard Code for Information Interchange (ASCII) is a character-encoding scheme that defines a binary representation for characters. The ASCII table contains both printable characters and control characters. ASCII characters were encoded in 7 bits and only contained the characters required to write text in English. Other character sets such as Unicode have been developed later to support all written languages.

ASN.1 The Abstract Syntax Notation One (ASN.1) was designed by ISO and ITU-T. It is a standard and flexible notation that can be used to describe data structures for representing, encoding, transmitting, and decoding data between applications. It was designed to be used in the Presentation layer of the OSI reference model but is now used in other protocols such as *SNMP*.

ATM Asynchronous Transfer Mode

BGP The Border Gateway Protocol is the interdomain routing protocol used in the global Internet.

BNF A Backus-Naur Form (BNF) is a formal way to describe a language by using syntactic and lexical rules. BNFs are frequently used to define programming languages, but also to define the messages exchanged between networked applications. **RFC 5234** explains how a BNF must be written to specify an Internet protocol.

broadcast a transmission mode where is same information is sent to all nodes in the network

CIDR Classless Inter Domain Routing is the current address allocation architecture for IPv4. It was defined in **RFC 1518** and **RFC 4632**.

dial-up line A synonym for a regular telephone line, i.e. a line that can be used to dial any telephone number.

DNS The Domain Name System is a distributed database that allows to map names on IP addresses.

DNS The Domain Name System is defined in **RFC 1035**

DNS The Domain Name System is a distributed database that can be queried by hosts to map names onto IP addresses

eBGP An eBGP session is a BGP session between two directly connected routers that belong to two different Autonomous Systems. Also called an external BGP session.

EGP Exterior Gateway Protocol. Synonym of interdomain routing protocol

EIGRP The Enhanced Interior Gateway Routing Protocol (EIGRP) is a proprietary intradomain routing protocol that is often used in enterprise networks. EIGRP uses the DUAL algorithm described in [Garcia1993].

frame a frame is the unit of information transfer in the datalink layer

Frame-Relay A wide area networking technology using virtual circuits that is deployed by telecom operators.

ftp The File Transfer Protocol defined in **RFC 959** has been the de facto protocol to exchange files over the Internet before the widespread adoption of HTTP **RFC 2616**

FTP The File Transfer Protocol is defined in **RFC 959**

hosts.txt A file that initially contained the list of all Internet hosts with their IPv4 address. As the network grew, this file was replaced by the DNS, but each host still maintains a small hosts.txt file that can be used when DNS is not available.

HTML The HyperText Markup Language specifies the structure and the syntax of the documents that are exchanged on the world wide web. HTML is maintained by the HTML working group of the *W3C*

HTTP The HyperText Transport Protocol is defined in **RFC 2616**

hub A relay operating in the physical layer.

IANA The Internet Assigned Numbers Authority (IANA) is responsible for the coordination of the DNS Root, IP addressing, and other Internet protocol resources

iBGP An iBGP session is a BGP between two routers belonging to the same Autonomous System. Also called an internal BGP session.

ICANN The Internet Corporation for Assigned Names and Numbers (ICANN) coordinates the allocation of domain names, IP addresses and AS numbers as well protocol parameters. It also coordinates the operation and the evolution of the DNS root name servers.

IETF The Internet Engineering Task Force is a non-profit organisation that develops the standards for the protocols used in the Internet. The IETF mainly covers the transport and network layers. Several application layer protocols are also standardised within the IETF. The work in the IETF is organised in working groups. Most of the work is performed by exchanging emails and there are three IETF meetings every year. Participation is open to anyone. See http://www.ietf.org

IGP Interior Gateway Protocol. Synonym of intradomain routing protocol

IGRP The Interior Gateway Routing Protocol (IGRP) is a proprietary intradomain routing protocol that uses distance vector. IGRP supports multiple metrics for each route but has been replaced by *EIGRP*

IMAP The Internet Message Access Protocol is defined in **RFC 3501**

IMAP The Internet Message Access Protocol (IMAP), defined in **RFC 3501**, is an application-level protocol that allows a client to access and manipulate the emails stored on a server. With IMAP, the email messages remain on the server and are not downloaded on the client.

Internet a public internet, i.e. a network composed of different networks that are running *IPv4* or *IPv6*

internet an internet is an internetwork, i.e. a network composed of different networks. The *Internet* is a very popular internetwork, but other internets have been used in the path.

inverse query For DNS servers and resolvers, an inverse query is a query for the domain name that corresponds to a given IP address.

IP Internet Protocol is the generic term for the network layer protocol in the TCP/IP protocol suite. *IPv4* is widely used today and *IPv6* is expected to replace *IPv4*

IPv4 is the version 4 of the Internet Protocol, the connectionless network layer protocol used in most of the Internet today. IPv4 addresses are encoded as a 32 bits field.

IPv6 is the version 6 of the Internet Protocol, the connectionless network layer protocol which is intended to replace *IPv4* . IPv6 addresses are encoded as a 128 bits field.

IS-IS Intermediate System- Intermediate System. A link-state intradomain routing that was initially defined for the ISO CLNP protocol but was extended to support IPv4 and IPv6. IS-IS is often used in ISP networks. It is defined in [ISO10589]

ISN The Initial Sequence Number of a TCP connection is the sequence number chosen by the client (resp. server) that is placed in the *SYN* (resp. *SYN+ACK*) segment during the establishment of the TCP connection.

ISO The International Standardization Organisation is an agency of the United Nations that is based in Geneva and develop standards on various topics. Within ISO, country representatives vote to approve or reject standards. Most of the work on the development of ISO standards is done in expert working groups. Additional information about ISO may be obtained from http://www.iso.int

ISO The International Standardization Organisation

ISO-3166 An *ISO* standard that defines codes to represent countries and their subdivisions. See http://www.iso.org/iso/country_codes.htm

ISP An Internet Service Provider, i.e. a network that provides Internet access to its clients.

ITU The International Telecommunication Union is a United Nation's agency whose purpose is to develop standards for the telecommunication industry. It was initially created to standardise the basic telephone system but expanded later towards data networks. The work within ITU is mainly done by network specialists from the telecommunication industry (operators and vendors). See http://www.itu.int for more information

IXP Internet eXchange Point. A location where routers belonging to different domains are attached to the same Local Area Network to establish peering sessions and exchange packets. See http://www.euro-ix.net/ or http://en.wikipedia.org/wiki/List_of_Internet_exchange_points_by_size for a partial list of IXPs.

LAN Local Area Network

leased line A telephone line that is permanently available between two endpoints.

MAN Metropolitan Area Network

MIME The Multipurpose Internet Mail Extensions (MIME) defined in **RFC 2045** are a set of extensions to the format of email messages that allow to use non-ASCII characters inside mail messages. A MIME message can be composed of several different parts each having a different format.

MIME document A MIME document is a document, encoded by using the *MIME* format.

minicomputer A minicomputer is a multi-user system that was typically used in the 1960s/1970s to serve departments. See the corresponding wikipedia article for additional information : http://en.wikipedia.org/wiki/Minicomputer

modem A modem (modulator-demodulator) is a device that encodes (resp. decodes) digital information by modulating (resp. demodulating) an analog signal. Modems are frequently used to transmit digital information over telephone lines and radio links. See http://en.wikipedia.org/wiki/Modem for a survey of various types of modems

MSS A TCP option used by a TCP entity in SYN segments to indicate the Maximum Segment Size that it is able to receive.

multicast a transmission mode where an information is sent efficiently to *all* the receivers that belong to a given group

nameserver A server that implements the DNS protocol and can answer queries for names inside its own domain.

NAT A Network Address Translator is a middlebox that translates IP packets.

NBMA A Non Broadcast Mode Multiple Access Network is a subnetwork that supports multiple hosts/routers but does not provide an efficient way of sending broadcast frames to all devices attached to the subnetwork. ATM subnetworks are an example of NBMA networks.

network-byte order Internet protocol allow to transport sequences of bytes. These sequences of bytes are sufficient to carry ASCII characters. The network-byte order refers to the Big-Endian encoding for 16 and 32 bits integer. See http://en.wikipedia.org/wiki/Endianness

NFS The Network File System is defined in **RFC 1094**

NTP The Network Time Protocol is defined in **RFC 1305**

OSI Open Systems Interconnection. A set of networking standards developed by *ISO* including the 7 layers OSI reference model.

OSPF Open Shortest Path First. A link-state intradomain routing protocol that is often used in enterprise and ISP networks. OSPF is defined in and **RFC 2328** and **RFC 5340**

packet a packet is the unit of information transfer in the network layer

PBL Problem-based learning is a teaching approach that relies on problems.

POP The Post Office Protocol is defined in **RFC 1939**

POP The Post Office Protocol (POP), defined **RFC 1939**, is an application-level protocol that allows a client to download email messages stored on a server.

resolver A server that implements the DNS protocol and can resolve queries. A resolver usually serves a set of clients (e.g. all hosts in campus or all clients of a given ISP). It sends DNS queries to nameservers everywhere on behalf of its clients and stores the received answers in its cache. A resolver must know the IP addresses of the root nameservers.

RIP Routing Information Protocol. An intradomain routing protocol based on distance vectors that is sometimes used in enterprise networks. RIP is defined in **RFC 2453**.

RIR Regional Internet Registry. An organisation that manages IP addresses and AS numbers on behalf of *IANA*.

root nameserver A name server that is responsible for the root of the domain names hierarchy. There are currently a dozen root nameservers and each DNS resolver See http://www.root-servers.org/ for more information about the operation of these root servers.

round-trip-time The round-trip-time (RTT) is the delay between the transmission of a segment and the reception of the corresponding acknowledgement in a transport protocol.

router A relay operating in the network layer.

RPC Several types of remote procedure calls have been defined. The RPC mechanism defined in **RFC 5531** is used by applications such as NFS

SDU (Service Data Unit) a Service Data Unit is the unit information transferred between applications

segment a segment is the unit of information transfer in the transport layer

SMTP The Simple Mail Transfer Protocol is defined in **RFC 821**

SNMP The Simple Network Management Protocol is a management protocol defined for TCP/IP networks.

socket A low-level API originally defined on Berkeley Unix to allow programmers to develop clients and servers.

spoofed packet A packet is said to be spoofed when the sender of the packet has used as source address a different address than its own.

SSH The Secure Shell (SSH) Transport Layer Protocol is defined in **RFC 4253**

standard query For DNS servers and resolvers, a standard query is a query for a *A* or a *AAAA* record. Such a query typically returns an IP address.

switch A relay operating in the datalink layer.

SYN cookie The SYN cookies is a technique used to compute the initial sequence number (ISN)

TCB The Transmission Control Block is the set of variables that are maintained for each established TCP connection by a TCP implementation.

TCP The Transmission Control Protocol is a protocol of the transport layer in the TCP/IP protocol suite that provides a reliable bytestream connection-oriented service on top of IP

TCP/IP refers to the *TCP* and *IP* protocols

telnet The telnet protocol is defined in **RFC 854**

TLD A Top-level domain name. There are two types of TLDs. The ccTLD are the TLD that correspond to a two letters *ISO-3166* country code. The gTLD are the generic TLDs that are not assigned to a country.

TLS Transport Layer Security, defined in **RFC 5246** is a cryptographic protocol that is used to provide communication security for Internet applications. This protocol is used on top of the transport service but a detailed description is outside the scope of this book.

UDP User Datagram Protocol is a protocol of the transport layer in the TCP/IP protocol suite that provides an unreliable connectionless service that includes a mechanism to detect corruption

unicast a transmission mode where an information is sent from one source to one recipient

vnc A networked application that allows to remotely access a computer's Graphical User Interface. See http://en.wikipedia.org/wiki/Virtual_Network_Computing

W3C The world wide web consortium was created to standardise the protocols and mechanisms used in the global www. It is thus focused on a subset of the application layer. See http://www.w3c.org

WAN Wide Area Network

X.25 A wide area networking technology using virtual circuits that was deployed by telecom operators.

X11 The XWindow system and the associated protocols are defined in [SG1990]

XML The eXtensible Markup Language (XML) is a flexible text format derived from SGML. It was originally designed for the electronic publishing industry but is now used by a wide variety of applications that need to exchange structured data. The XML specifications are maintained by several working groups of the *W3C*

5.2 Bibliography

Whenever possible, the bibliography includes stable hypertext links to the references cited.

5.3 Indices and tables

- *genindex*
- *search*

[802.11] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Information Technology - Telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements - Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE, 1999.

[802.1d] LAN/MAN Standards Committee of the IEEE Computer Society, IEEE Standard for Local and metropolitan area networks Media Access Control (MAC) Bridges , IEEE Std 802.1DTM-2004, 2004,

[802.1q] LAN/MAN Standards Committee of the IEEE Computer Society, IEEE Standard for Local and metropolitan area networks— Virtual Bridged Local Area Networks, 2005,

[802.2] IEEE 802.2-1998 (ISO/IEC 8802-2:1998), IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 2: Logical Link Control. Available from http://standards.ieee.org/getieee802/802.2.html

[802.3] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Information Technology - Telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements - Part 3 : Carrier Sense multiple access with collision detection (CSMA/CD) access method and physical layer specification. IEEE, 2000. Available from http://standards.ieee.org/getieee802/802.3.html

[802.5] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 5: Token Ring Access Method and Physical Layer Specification. IEEE, 1998. available from http://standards.ieee.org/getieee802

[ACO+2006] Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., Friedman, T., Latapy, M., Magnien, C., Teixeira, R., Avoiding traceroute anomalies with Paris traceroute, Internet Measurement Conference, October 2006, See also http://www.paris-traceroute.net/

[AS2004] Androutsellis-Theotokis, S. and Spinellis, D.. 2004. A survey of peer-to-peer content distribution technologies. ACM Comput. Surv. 36, 4 (December 2004), 335-371.

[ATLAS2009] Labovitz, C., Iekel-Johnson, S., McPherson, D., Oberheide, J. and Jahanian, F., Internet interdomain traffic. In Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM (SIGCOMM '10). ACM, New York, NY, USA, 75-86.

[AW05] Arlitt, M. and Williamson, C. 2005. An analysis of TCP reset behaviour on the internet. SIGCOMM Comput. Commun. Rev. 35, 1 (Jan. 2005), 37-44.

[Abramson1970] Abramson, N., THE ALOHA SYSTEM: another alternative for computer communications. In Proceedings of the November 17-19, 1970, Fall Joint Computer Conference (Houston, Texas, November 17 - 19, 1970). AFIPS '70 (Fall). ACM, New York, NY, 281-285.

[B1989] Berners-Lee, T., Information Management: A Proposal, March 1989

[Baran] Baran, P., *On distributed communications series*, http://www.rand.org/about/history/baran.list.html,

[BE2007] Biondi, P. and A. Ebalard, IPv6 Routing Header Security, CanSecWest Security Conference 2007, April 2007.

[BF1995] Bonomi, F. and Fendick, K.W., The rate-based flow control framework for the available bit rate ATM service, IEEE Network, Mar/Apr 1995, Volume: 9, Issue: 2, pages : 25-39

[BG1992] Bertsekas, D., Gallager, G., Data networks, second edition, Prentice Hall, 1992

[BMO2006] Bhatia, M., Manral, V., Ohara, Y., IS-IS and OSPF Difference Discussions, work in progress, Internet draft, Jan. 2006

[BMvB2009] Bagnulo, M., Matthews, P., van Beijnum, I., NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers, Internet draft, work in progress, October 2009,

[BNT1997] Beech, W., Nielsen, D., Taylor, J., AX.25 Link Access Protocol for Amateur Packet Radio, version 2.2, Revision: July 1998

[BOP1994] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L., TCP Vegas: new techniques for congestion detection and avoidance. In Proceedings of the Conference on Communications Architectures, Protocols and Applications (London, United Kingdom, August 31 - September 02, 1994). SIGCOMM '94. ACM, New York, NY, 24-35.

[Benvenuti2005] Benvenuti, C., Understanding Linux Network Internals, O'Reilly Media, 2005

[BH2013] Bormann, C., Hoffman, P., Concise Binary Object Representation (CBOR), Internet draft, draft-bormann-cbor-09, work in progress, 2013

[Bush1945] Bush, V. As we may think The Atlantic Monthly 176 (July 1945), pp. 101–108

[Bush1993] Bush, R., FidoNet: technology, tools, and history. Commun. ACM 36, 8 (Aug. 1993), 31-35.

[Bux1989] Bux, W., Token-ring local-area networks and their performance, Proceedings of the IEEE, Vol 77, No 2, p. 238-259, Feb. 1989

[BYL2008] Buford, J., Yu, H., Lua, E.K., P2P Networking and Applications, Morgan Kaufmann, 2008

[CB2003] Cheswick, William R., Bellovin, Steven M., Rubin, Aviel D., Firewalls and internet security - Second edition - Repelling the Wily Hacker, Addison-Wesley 2003

[CCB+2013] Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B., Dukkipati, N., Chu, H., Terzis, A., and Herbert, T., packetdrill: scriptable network stack testing, from sockets to packets. In Proceedings of the 2013 USENIX conference on Annual Technical Conference (USENIX ATC'13). USENIX Association, Berkeley, CA, USA, 213-218.

[CD2008] Calvert, K., Donahoo, M., TCP/IP Sockets in Java : Practical Guide for Programmers, Morgan Kaufman, 2008

[CJ1989] Chiu, D., Jain, R., Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks, Computer Networks and ISDN Systems Vol 17, pp 1-14, 1989

[CK74] Cerf, V., Kahn, R., A Protocol for Packet Network Intercommunication, IEEE Transactions on Communications, May 1974

[CNPI09] Gont, F., Security Assessment of the Transmission Control Protocol (TCP),Security Assessment of the Transmission Control Protocol (TCP), Internet draft, work in progress, Jan. 2011

[COZ2008] Chi, Y., Oliveira, R., Zhang, L., Cyclops: The Internet AS-level Observatory, ACM SIGCOMM Computer Communication Review (CCR), October 2008

[CSP2009] Carr, B., Sury, O., Palet Martinez, J., Davidson, A., Evans, R., Yilmaz, F., Wijte, Y., IPv6 Address Allocation and Assignment Policy, RIPE document ripe-481, September 2009

[CT1980] Crane, R., Taft, E., Practical considerations in Ethernet local network design, Proc. of the 13th Hawaii International Conference on Systems Sciences, Honolulu, January, 1980, pp. 166–174

[Cheshire2010] Cheshire, S., Connect-By-Name for IPv6, presentation at IETF 79th, November 2010

[Cheswick1990] Cheswick, B., An Evening with Berferd In Which a Cracker is Lured, Endured, and Studied, Proc. Winter USENIX Conference, 1990, pp. 163-174

[Clark88] Clark D., The Design Philosophy of the DARPA Internet Protocols, Computer Communications Review 18:4, August 1988, pp. 106-114

[Comer1988] Comer, D., *Internetworking with TCP/IP : principles, protocols & architecture*, Prentice Hall, 1988

[Comer1991] Comer D., *Internetworking With TCP/IP : Design Implementation and Internals*, Prentice Hall, 1991

[Cohen1980] Cohen, D., *On Holy Wars and a Plea for Peace*, IEN 137, April 1980, http://www.ietf.org/rfc/ien/ien137.txt

[DC2009] Donahoo, M., Calvert, K., TCP/IP Sockets in C: Practical Guide for Programmers , Morgan Kaufman, 2009,

[DIX] Digital, Intel, Xerox, The Ethernet: a local area network: data link layer and physical layer specifications. SIGCOMM Comput. Commun. Rev. 11, 3 (Jul. 1981), 20-66.

[DKF+2007] Dimitropoulos, X., Krioukov, D., Fomenkov, M., Huffaker, B., Hyun, Y., Claffy, K., Riley, G., AS Relationships: Inference and Validation, ACM SIGCOMM Computer Communication Review (CCR), Jan. 2007

[DP1981] Dalal, Y. K. and Printis, R. S., 48-bit absolute internet and Ethernet host numbers. In Proceedings of the Seventh Symposium on Data Communications (Mexico City, Mexico, October 27 - 29, 1981). SIGCOMM '81. ACM, New York, NY, 240-245.

[DRC+2010] Dukkipati, N., Refice, T., Cheng, Y., Chu, J., Herbert, T., Agarwal, A., Jain, A., Sutin, N., An Argument for Increasing TCP's Initial Congestion Window, ACM SIGCOMM Computer Communications Review, vol. 40 (2010), pp. 27-33

[Dubuisson2000] 15. Dubuisson, *ASN.1 : Communication between Heterogeneous Systems <http://www.oss.com/asn1/resources/books-whitepapers-pubs/asn1-books.html#dubuisson>*, Morgan Kauffman, 2000

[Dunkels2003] Dunkels, A., Full TCP/IP for 8-Bit Architectures. In Proceedings of the first international conference on mobile applications, systems and services (MOBISYS 2003), San Francisco, May 2003.

[DT2007] Donnet, B. and Friedman, T., Internet Topology Discovery: a Survey. IEEE Communications Surveys and Tutorials, 9(4):2-15, December 2007

[DYGU2004] Davik, F. Yilmaz, M. Gjessing, S. Uzun, N., IEEE 802.17 resilient packet ring tutorial, IEEE Communications Magazine, Mar 2004, Vol 42, N 3, p. 112-118

[Dijkstra1959] Dijkstra, E., A Note on Two Problems in Connection with Graphs. Numerische Mathematik, 1:269- 271, 1959

[FDDI] ANSI. *Information systems - Fiber Distributed Data Interface (FDDI) - token ring media access control (MAC)*. ANSI X3.139-1987 (R1997), 1997

[Fletcher1982] Fletcher, J., An Arithmetic Checksum for Serial Transmissions, Communications, IEEE Transactions on, Jan. 1982, Vol. 30, N. 1, pp. 247-252

[FFEB2005] Francois, P., Filsfils, C., Evans, J., and Bonaventure, O., Achieving sub-second IGP convergence in large IP networks. SIGCOMM Comput. Commun. Rev. 35, 3 (Jul. 2005), 35-44.

[NGB+1997] Nielsen, H., Gettys, J., Baird-Smith, A., Prudhommeaux, E., Wium Lie, H., and Lilley, C. Network performance effects of HTTP/1.1, CSS1, and PNG. SIGCOMM Comput. Commun. Rev. 27, 4 (October 1997), 155-166.

[FJ1993] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. IEEE/ACM Trans. Netw. 1, 4 (August 1993), 397-413.

[FJ1994] Floyd, S., and Jacobson, V., The Synchronization of Periodic Routing Messages, IEEE/ACM Transactions on Networking, V.2 N.2, p. 122-136, April 1994

[FLM2008] Fuller, V., Lear, E., Meyer, D., Reclassifying 240/4 as usable unicast address space, Internet draft, March 2008, workin progress

[FRT2002] Fortz, B. Rexford, J. ,Thorup, M., Traffic engineering with traditional IP routing protocols, IEEE Communication Magazine, October 2002

[FTY99] Theodore Faber, Joe Touch, and Wei Yue, The TIME-WAIT state in TCP and Its Effect on Busy Servers, Proc. Infocom '99, pp. 1573

[Feldmeier95] Feldmeier, D. C., Fast software implementation of error detection codes. IEEE/ACM Trans. Netw. 3, 6 (Dec. 1995), 640-651.

[GAVE1999] Govindan, R., Alaettinoglu, C., Varadhan, K., Estrin, D., An Architecture for Stable, Analyzable Internet Routing, IEEE Network Magazine, Vol. 13, No. 1, pp. 29–35, January 1999

[GC2000] Grier, D., Campbell, M., A social history of Bitnet and Listserv, 1985-1991, Annals of the History of Computing, IEEE, Volume 22, Issue 2, Apr-Jun 2000, pp. 32 - 41

[Genilloud1990] Genilloud, G., X.400 MHS: first steps towards an EDI communication standard. SIGCOMM Comput. Commun. Rev. 20, 2 (Apr. 1990), 72-86.

[GGR2001] Gao, L., Griffin, T., Rexford, J., Inherently safe backup routing with BGP, Proc. IEEE INFOCOM, April 2001

[GN2011] Gettys, J., Nichols, K., Bufferbloat: dark buffers in the internet. Communications of the ACM 55, no. 1 (2012): 57-65.

[GR2001] Gao, L., Rexford, J., Stable Internet routing without global coordination, IEEE/ACM Transactions on Networking, December 2001, pp. 681-692

[GSW2002] Griffin, T. G., Shepherd, F. B., and Wilfong, G., The stable paths problem and interdomain routing. IEEE/ACM Trans. Netw. 10, 2 (Apr. 2002), 232-243

[GW1999] Griffin, T. G. and Wilfong, G., An analysis of BGP convergence properties. SIGCOMM Comput. Commun. Rev. 29, 4 (Oct. 1999), 277-288.

[GW2002] Griffin, T. and Wilfong, G. T., Analysis of the MED Oscillation Problem in BGP. In Proceedings of the 10th IEEE international Conference on Network Protocols (November 12 - 15, 2002). ICNP. IEEE Computer Society, Washington, DC, 90-99

[Garcia1993] Garcia-Lunes-Aceves, J., Loop-Free Routing Using Diffusing Computations, IEEE/ACM Transactions on Networking, Vol. 1, No, 1, Feb. 1993

[Gast2002] Gast, M., 802.11 Wireless Networks : The Definitive Guide, O'Reilly, 2002

[Gill2004] Gill, V. , Lack of Priority Queuing Considered Harmful, ACM Queue, December 2004

[Goralski2009] Goralski, W., The Illustrated network : How TCP/IP works in a modern network, Morgan Kaufmann, 2009

[HFPMC2002] Huffaker, B., Fomenkov, M., Plummer, D., Moore, D., Claffy, K., Distance Metrics in the Internet, Presented at the IEEE International Telecommunications Symposium (ITS) in 2002.

[HRX2008] Ha, S., Rhee, I., and Xu, L., CUBIC: a new TCP-friendly high-speed TCP variant. SIGOPS Oper. Syst. Rev. 42, 5 (Jul. 2008), 64-74.

[HV2008] Hogg, S. Vyncke, E., IPv6 Security, Cisco Press, 2008

[IMHM2013] Ishihara, K., Mukai, M., Hiromi, R., Mawatari, M., Packet Filter and Route Filter Recommendation for IPv6 at xSP routers, 2013

[ISO10589] ISO, Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473) , 2002

[Jacobson1988] Jacobson, V., Congestion avoidance and control. In Symposium Proceedings on Communications Architectures and Protocols (Stanford, California, United States, August 16 - 18, 1988). V. Cerf, Ed. SIGCOMM '88. ACM, New York, NY, 314-329.

[Jain1990] Jain, R., Congestion control in computer networks : Issues and trends, IEEE Network Magazine, May 1990, pp. 24-30

[JLT2013] Jesup, R., Loreto, S., Tuexen, M., RTCWeb Data Channels, Internet draft, draft-ietf-rtcweb-data-channel, work in progress, 2013

[JSBM2002] Jung, J., Sit, E., Balakrishnan, H., and Morris, R. 2002. DNS performance and the effectiveness of caching. IEEE/ACM Trans. Netw. 10, 5 (Oct. 2002), 589-603.

[JSON-RPC2] JSON-RPC Working group, JSON-RPC 2.0 Specification, available on http://www.jsonrpc.org, 2010

[Kerrisk2010] Kerrisk, M., The Linux Programming Interface, No Starch Press, 2010

[KM1995] Kent, C. A. and Mogul, J. C., Fragmentation considered harmful. SIGCOMM Comput. Commun. Rev. 25, 1 (Jan. 1995), 75-87.

[KNT2013] Kühlewind, M., Neuner, S., Trammell, B., On the state of ECN and TCP Options on the Internet. Proceedings of the 14th Passive and Active Measurement conference (PAM 2013), Hong Kong, March 2013

[KP91] Karn, P. and Partridge, C., Improving round-trip time estimates in reliable transport protocols. ACM Trans. Comput. Syst. 9, 4 (Nov. 1991), 364-373.

[KPD1985] Karn, P., Price, H., Diersing, R., Packet radio in amateur service, IEEE Journal on Selected Areas in Communications, 3, May, 1985

[KPS2003] Kaufman, C., Perlman, R., and Sommerfeld, B. DoS protection for UDP-based protocols. In Proceedings of the 10th ACM Conference on Computer and Communications Security (Washington D.C., USA, October 27 - 30, 2003). CCS '03. ACM, New York, NY, 2-7.

[KR1995] Kung, N.T. Morris, R., Credit-based flow control for ATM networks, IEEE Network, Mar/Apr 1995, Volume: 9, Issue: 2, pages: 40-48

[KT1975] Kleinrock, L., Tobagi, F., Packet Switching in Radio Channels: Part I–Carrier Sense Multiple-Access Modes and their Throughput-Delay Characteristics, IEEE Transactions on Communications, Vol. COM-23, No. 12, pp. 1400-1416, December 1975.

[KW2009] Katz, D., Ward, D., *Bidirectional Forwarding Detection*, **RFC 5880**, June 2010

[KZ1989] Khanna, A. and Zinky, J. 1989. The revised ARPANET routing metric. SIGCOMM Comput. Commun. Rev. 19, 4 (Aug. 1989), 45-56.

[KuroseRoss09] Kurose J. and Ross K., Computer networking : a top-down approach featuring the Internet, Addison-Wesley, 2009

[Licklider1963] Licklider, J., Memorandum For Members and Affiliates of the Intergalactic Computer Network, 1963

[LCCD09] Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G., and Wolff, S., A brief history of the internet. SIGCOMM Comput. Commun. Rev. 39, 5 (Oct. 2009), 22-31.

[LCP2005] Eng Keong Lua, Crowcroft, J., Pias, M., Sharma, R., Lim, S., A survey and comparison of peer-to-peer overlay network schemes, Communications Surveys & Tutorials, IEEE, Volume: 7 , Issue: 2, 2005, pp. 72-93

[LeB2009] Leroy, D. and O. Bonaventure, Preparing network configurations for IPv6 renumbering, International of Network Management, 2009

[LFJLMT] Leffler, S., Fabry, R., Joy, W., Lapsley, P., Miller, S., Torek, C., An Advanced 4.4BSD Interprocess Communication Tutorial, 4.4 BSD Programmer's Supplementary Documentation

[LNO1996] 20. (a) Lakshman, Arnold Neidhardt, and Teunis J. Ott. 1996. The drop from front strategy in TCP and in TCP over ATM. INFOCOM'96, Vol. 3. IEEE Computer Society, Washington, DC, USA, 1242-1250.

[LSP1982] Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (Jul. 1982), 382-401.

[Leboudec2008] Leboudec, J.-Y., Rate Adaptation Congestion Control and Fairness : a tutorial, Dec. 2008

[Malamud1991] Malamud, C., Analyzing DECnet/OSI phase V, Van Nostrand Reinhold, 1991

[McFadyen1976] McFadyen, J., Systems Network Architecture: An overview, IBM Systems Journal, Vol. 15, N. 1, pp. 4-23, 1976

[McKusick1999] McKusick, M., Twenty Years of Berkeley Unix : From AT&T-Owned to Freely Redistributable, in Open Sources: Voices from the Open Source Revolution, Oreilly, 1999, http://oreilly.com/catalog/opensources/book/toc.html

[ML2011] Minei I. and Lucek J. ,'MPLS-Enabled Applications: Emerging Developments and New Technologies <http://www.amazon.com/MPLS-Enabled-Applications-Developments-Technologies-Communications/dp/0470665459>'_ (Wiley Series on Communications Networking & Distributed Systems), Wiley, 2011

[MRR1979] McQuillan, J. M., Richer, I., and Rosen, E. C., An overview of the new routing algorithm for the ARPANET. In Proceedings of the Sixth Symposium on Data Communications (Pacific Grove, California, United States, November 27 - 29, 1979). SIGCOMM '79. ACM, New York, NY, 63-68.

[MRR1980] McQuillan, J.M., Richer, I., Rosen, E., The New Routing Algorithm for the ARPANET Communications, IEEE Transactions on , vol.28, no.5, pp.711,719, May 1980

[MSMO1997] Mathis, M., Semke, J., Mahdavi, J., and Ott, T. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. SIGCOMM Comput. Commun. Rev. 27, 3 (Jul. 1997), 67-82.

[MSV1987] Molle, M., Sohraby, K., Venetsanopoulos, A., Space-Time Models of Asynchronous CSMA Protocols for Local Area Networks, IEEE Journal on Selected Areas in Communications, Volume: 5 Issue: 6, Jul 1987 Page(s): 956 -96

[MUF+2007] Mühlbauer, W., Uhlig, S., Fu, B., Meulle, M., and Maennel, O., In search for an appropriate granularity to model routing policies. In Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (Kyoto, Japan, August 27 - 31, 2007). SIGCOMM '07. ACM, New York, NY, 145-156.

[Malkin1999] Malkin, G., RIP: An Intra-Domain Routing Protocol, Addison Wesley, 1999

[Metcalfe1976] Metcalfe R., Boggs, D., Ethernet: Distributed packet-switching for local computer networks. Communications of the ACM, 19(7):395–404, 1976.

[Mills2006] Mills, D.L., Computer Network Time Synchronization: the Network Time Protocol. CRC Press, March 2006, 304 pp.

[Miyakawa2008] Miyakawa, S., From IPv4 only To v4/v6 Dual Stack, IETF72 IAB Technical Plenary, July 2008

[Mogul1995] Mogul, J. , The case for persistent-connection HTTP. In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication (Cambridge, Massachusetts, United States, August 28 - September 01, 1995). D. Oran, Ed. SIGCOMM '95. ACM, New York, NY, 299-313.

[Moore] Moore, R., *Packet switching history*, http://rogerdmoore.ca/PS/

[Moy1998] Moy, J., OSPF: Anatomy of an Internet Routing Protocol, Addison Wesley, 1998

[Myers1998] Myers, B. A., A brief history of human-computer interaction technology. interactions 5, 2 (Mar. 1998), 44-54.

[Nelson1965] Nelson, T. H., Complex information processing: a file structure for the complex, the changing and the indeterminate. In Proceedings of the 1965 20th National Conference (Cleveland, Ohio, United States, August 24 - 26, 1965). L. Winner, Ed. ACM '65. ACM, New York, NY, 84-100.

[Paxson99] Paxson, V. , End-to-end Internet packet dynamics. SIGCOMM Comput. Commun. Rev. 27, 4 (Oct. 1997), 139-152.

[Perlman1985] Perlman, R., An algorithm for distributed computation of a spanning tree in an extended LAN. SIGCOMM Comput. Commun. Rev. 15, 4 (Sep. 1985), 44-53.

[Perlman2000] Perlman, R., Interconnections : Bridges, routers, switches and internetworking protocols, 2nd edition, Addison Wesley, 2000

[Perlman2004] Perlman, R., RBridges: Transparent Routing, Proc. IEEE Infocom , March 2004.

[Pouzin1975] Pouzin, L., The CYCLADES Network - Present state and development trends, Symposium on Computer Networks, 1975 pp 8-13.

[Rago1993] Rago, S., UNIX System V network programming, Addison Wesley, 1993

[RE1989] Rochlis, J. A. and Eichin, M. W., With microscope and tweezers: the worm from MIT's perspective. Commun. ACM 32, 6 (Jun. 1989), 689-698.

[RFC20] Cerf, V., *ASCII format for network interchange*, **RFC 20**, Oct. 1969

[RFC768] Postel, J., *User Datagram Protocol*, **RFC 768**, Aug. 1980

[RFC789] Rosen, E., *Vulnerabilities of network control protocols: An example*, **RFC 789**, July 1981

[RFC791] Postel, J., *Internet Protocol*, **RFC 791**, Sep. 1981

[RFC792] Postel, J., *Internet Control Message Protocol*, **RFC 792**, Sep. 1981

[RFC793] Postel, J., *Transmission Control Protocol*, **RFC 793**, Sept. 1981

[RFC813] Clark, D., *Window and Acknowledgement Strategy in TCP*, **RFC 813**, July 1982

[RFC819] Su, Z. and Postel, J., *Domain naming convention for Internet user applications*, **RFC 819**, Aug. 1982

[RFC821] Postel, J., *Simple Mail Transfer Protocol*, **RFC 821**, Aug. 1982

[RFC822] Crocker, D., *Standard for the format of ARPA Internet text messages, :rfc:'822*, Aug. 1982

[RFC826] Plummer, D., ''Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware', **RFC 826**, Nov. 1982

[RFC879] Postel, J., *TCP maximum segment size and related topics*, **RFC 879**, Nov. 1983

[RFC893] Leffler, S. and Karels, M., *Trailer encapsulations*, **RFC 893**, April 1984

[RFC894] Hornig, C., *A Standard for the Transmission of IP Datagrams over Ethernet Networks*, **RFC 894**, April 1984

[RFC896] Nagle, J., *Congestion Control in IP/TCP Internetworks*, **RFC 896**, Jan. 1984

[RFC952] Harrenstien, K. and Stahl, M. and Feinler, E., *DoD Internet host table specification*, **RFC 952**, Oct. 1985

[RFC959] Postel, J. and Reynolds, J., *File Transfer Protocol*, **RFC 959**, Oct. 1985

[RFC974] Partridge, C., *Mail routing and the domain system*, **RFC 974**, Jan. 1986

[RFC1032] Stahl, M., *Domain administrators guide*, **RFC 1032**, Nov. 1987

[RFC1035] Mockapteris, P., *Domain names - implementation and specification*, **RFC 1035**, Nov. 1987

[RFC1042] Postel, J. and Reynolds, J., *Standard for the transmission of IP datagrams over IEEE 802 networks*, **RFC 1042**, Feb. 1988

[RFC1055] Romkey, J., *Nonstandard for transmission of IP datagrams over serial lines: SLIP*, **RFC 1055**, June 1988

[RFC1071] Braden, R., Borman D. and Partridge, C., *Computing the Internet checksum*, **RFC 1071**, Sep. 1988

[RFC1122] Braden, R., *Requirements for Internet Hosts - Communication Layers*, **RFC 1122**, Oct. 1989

[RFC1144] Jacobson, V., *Compressing TCP/IP Headers for Low-Speed Serial Links*, **RFC 1144**, Feb. 1990

[RFC1149] Waitzman, D., *Standard for the transmission of IP datagrams on avian carriers*, **RFC 1149**, Apr. 1990

[RFC1169] Cerf, V. and Mills, K., *Explaining the role of GOSIP*, **RFC 1169**, Aug. 1990

[RFC1191] Mogul, J. and Deering, S., *Path MTU discovery*, **RFC 1191**, Nov. 1990

[RFC1195] Callon, R., *Use of OSI IS-IS for routing in TCP/IP and dual environments*, **RFC 1195**, Dec. 1990

[RFC1258] Kantor, B., *BSD Rlogin*, **RFC 1258**, Sept. 1991

[RFC1321] Rivest, R., *The MD5 Message-Digest Algorithm*, **RFC 1321**, April 1992

[RFC1323] Jacobson, V., Braden R. and Borman, D., *TCP Extensions for High Performance*, **RFC 1323**, May 1992

[RFC1347] Callon, R., TCP and UDP with Bigger Addresses (TUBA), *A Simple Proposal for Internet Addressing and Routing*, **RFC 1347**, June 1992

[RFC1518] Rekhter, Y. and Li, T., *An Architecture for IP Address Allocation with CIDR*, **RFC 1518**, Sept. 1993

[RFC1519] Fuller V., Li T., Yu J. and Varadhan, K., *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*, **RFC 1519**, Sept. 1993

[RFC1542] Wimer, W., *Clarifications and Extensions for the Bootstrap Protocol*, **RFC 1542**, Oct. 1993

[RFC1548] Simpson, W., *The Point-to-Point Protocol (PPP)*, **RFC 1548**, Dec. 1993

[RFC1550] Bradner, S. and Mankin, A., *IP: Next Generation (IPng) White Paper Solicitation*, **RFC 1550**, Dec. 1993

[RFC1561] Piscitello, D., *Use of ISO CLNP in TUBA Environments*, **RFC 1561**, Dec. 1993

[RFC1621] Francis, P., *PIP Near-term architecture*, **RFC 1621**, May 1994

[RFC1624] Risjsighani, A., *Computation of the Internet Checksum via Incremental Update*, **RFC 1624**, May 1994

[RFC1631] Egevang K. and Francis, P., *The IP Network Address Translator (NAT)*, **RFC 1631**, May 1994

[RFC1661] Simpson, W., *The Point-to-Point Protocol (PPP)*, **RFC 1661**, Jul. 1994

[RFC1662] Simpson, W., *PPP in HDLC-like Framing*, **RFC 1662**, July 1994

[RFC1710] Hinden, R., *Simple Internet Protocol Plus White Paper*, **RFC 1710**, Oct. 1994

[RFC1738] Berners-Lee, T., Masinter, L., and McCahill M., *Uniform Resource Locators (URL)*, **RFC 1738**, Dec. 1994

[RFC1752] Bradner, S. and Mankin, A., *The Recommendation for the IP Next Generation Protocol*, **RFC 1752**, Jan. 1995

[RFC1812] Baker, F., *Requirements for IP Version 4 Routers*, **RFC 1812**, June 1995

[RFC1819] Delgrossi, L., Berger, L., *Internet Stream Protocol Version 2 (ST2) Protocol Specification - Version ST2+*, **RFC 1819**, Aug. 1995

[RFC1889] Schulzrinne H., Casner S., Frederick, R. and Jacobson, V., *RTP: A Transport Protocol for Real-Time Applications*, **RFC 1889**, Jan. 1996

[RFC1896] Resnick P., Walker A., *The text/enriched MIME Content-type*, **RFC 1896**, Feb. 1996

[RFC1918] Rekhter Y., Moskowitz B., Karrenberg D., de Groot G. and Lear, E., *Address Allocation for Private Internets*, **RFC 1918**, Feb. 1996

[RFC1939] Myers, J. and Rose, M., *Post Office Protocol - Version 3*, **RFC 1939**, May 1996

[RFC1945] Berners-Lee, T., Fielding, R. and Frystyk, H., *Hypertext Transfer Protocol – HTTP/1.0*, **RFC 1945**, May 1996

[RFC1948] Bellovin, S., *Defending Against Sequence Number Attacks*, **RFC 1948**, May 1996

[RFC1951] Deutsch, P., *DEFLATE Compressed Data Format Specification version 1.3*, **RFC 1951**, May 1996

[RFC1981] McCann, J., Deering, S. and Mogul, J., *Path MTU Discovery for IP version 6*, **RFC 1981**, Aug. 1996

[RFC2003] Perkins, C., *IP Encapsulation within IP*, **RFC 2003**, Oct. 1996

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S. and Romanow, A., *TCP Selective Acknowledgment Options*, **RFC 2018**, Oct. 1996

[RFC2045] Freed, N. and Borenstein, N., *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, **RFC 2045**, Nov. 1996

[RFC2046] Freed, N. and Borenstein, N., *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, **RFC 2046**, Nov. 1996

[RFC2050] Hubbard, K. and Kosters, M. and Conrad, D. and Karrenberg, D. and Postel, J., *Internet Registry IP Allocation Guidelines*, **RFC 2050**, Nov. 1996

[RFC2080] Malkin, G. and Minnear, R., *RIPng for IPv6*, **RFC 2080**, Jan. 1997

[RFC2082] Baker, F. and Atkinson, R., *RIP-2 MD5 Authentication*, **RFC 2082**, Jan. 1997

[RFC2131] Droms, R., *Dynamic Host Configuration Protocol*, **RFC 2131**, March 1997

[RFC2140] Touch, J., *TCP Control Block Interdependence*, **RFC 2140**, April 1997

[RFC2225] Laubach, M., Halpern, J., *Classical IP and ARP over ATM*, **RFC 2225**, April 1998

[RFC2328] Moy, J., *OSPF Version 2*, **RFC 2328**, April 1998

[RFC2332] Luciani, J. and Katz, D. and Piscitello, D. and Cole, B. and Doraswamy, N., *NBMA Next Hop Resolution Protocol (NHRP)*, **RFC 2332**, April 1998

[RFC2364] Gross, G. and Kaycee, M. and Li, A. and Malis, A. and Stephens, J., *PPP Over AAL5*, **RFC 2364**, July 1998

[RFC2368] Hoffman, P. and Masinter, L. and Zawinski, J., *The mailto URL scheme*, **RFC 2368**, July 1998

[RFC2453] Malkin, G., *RIP Version 2*, **RFC 2453**, Nov. 1998

[RFC2460] Deering S., Hinden, R., *Internet Protocol, Version 6 (IPv6) Specification*, **RFC 2460**, Dec. 1998

[RFC2464] Crawford, M., *Transmission of IPv6 Packets over Ethernet Networks*, **RFC 2464**, Dec. 1998

[RFC2507] Degermark, M. and Nordgren, B. and Pink, S., *IP Header Compression*, **RFC 2507**, Feb. 1999

[RFC2516] Mamakos, L. and Lidl, K. and Evarts, J. and Carrel, J. and Simone, D. and Wheeler, R., *A Method for Transmitting PPP Over Ethernet (PPPoE)*, **RFC 2516**, Feb. 1999

[RFC2581] Allman, M. and Paxson, V. and Stevens, W., *TCP Congestion Control*, **RFC 2581**, April 1999

[RFC2616] Fielding, R. and Gettys, J. and Mogul, J. and Frystyk, H. and Masinter, L. and Leach, P. and Berners-Lee, T., *Hypertext Transfer Protocol – HTTP/1.1*, **RFC 2616**, June 1999

[RFC2617] Franks, J. and Hallam-Baker, P. and Hostetler, J. and Lawrence, S. and Leach, P. and Luotonen, A. and Stewart, L., *HTTP Authentication: Basic and Digest Access Authentication*, **RFC 2617**, June 1999

[RFC2622] Alaettinoglu, C. and Villamizar, C. and Gerich, E. and Kessens, D. and Meyer, D. and Bates, T. and Karrenberg, D. and Terpstra, M., *Routing Policy Specification Language (RPSL)*, **RFC 2622**, June 1999

[RFC2675] Tsirtsis, G. and Srisuresh, P., *Network Address Translation - Protocol Translation (NAT-PT)*, **RFC 2766**, Feb. 2000

[RFC2854] Connolly, D. and Masinter, L., *The 'text/html' Media Type*, **RFC 2854**, June 2000

[RFC2965] Kristol, D. and Montulli, L., *HTTP State Management Mechanism*, **RFC 2965**, Oct. 2000

[RFC2988] Paxson, V. and Allman, M., *Computing TCP's Retransmission Timer*, **RFC 2988**, Nov. 2000

[RFC2991] Thaler, D. and Hopps, C., *Multipath Issues in Unicast and Multicast Next-Hop Selection*, **RFC 2991**, Nov. 2000

[RFC3021] Retana, A. and White, R. and Fuller, V. and McPherson, D., *Using 31-Bit Prefixes on IPv4 Point-to-Point Links*, **RFC 3021**, Dec. 2000

[RFC3022] Srisuresh, P., Egevang, K., *Traditional IP Network Address Translator (Traditional NAT)*, **RFC 3022**, Jan. 2001

[RFC3031] Rosen, E. and Viswanathan, A. and Callon, R., *Multiprotocol Label Switching Architecture*, **RFC 3031**, Jan. 2001

[RFC3168] Ramakrishnan, K. and Floyd, S. and Black, D., *The Addition of Explicit Congestion Notification (ECN) to IP*, **RFC 3168**, Sept. 2001

[RFC3243] Carpenter, B. and Brim, S., *Middleboxes: Taxonomy and Issues*, **RFC 3234**, Feb. 2002

[RFC3235] Senie, D., *Network Address Translator (NAT)-Friendly Application Design Guidelines*, **RFC 3235**, Jan. 2002

[RFC3309] Stone, J. and Stewart, R. and Otis, D., *Stream Control Transmission Protocol (SCTP) Checksum Change*, **RFC 3309**, Sept. 2002

[RFC3315] Droms, R. and Bound, J. and Volz, B. and Lemon, T. and Perkins, C. and Carney, M., *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*, **RFC 3315**, July 2003

[RFC3330] IANA, *Special-Use IPv4 Addresses*, **RFC 3330**, Sept. 2002

[RFC3360] Floyd, S., *Inappropriate TCP Resets Considered Harmful*, **RFC 3360**, Aug. 2002

[RFC3390] Allman, M. and Floyd, S. and Partridge, C., *Increasing TCP's Initial Window*, **RFC 3390**, Oct. 2002

[RFC3490] Faltstrom, P. and Hoffman, P. and Costello, A., *Internationalizing Domain Names in Applications (IDNA)*, **RFC 3490**, March 2003

[RFC3501] Crispin, M., *Internet Message Access Protocol - Version 4 rev1*, **RFC 3501**, March 2003

[RFC3513] Hinden, R. and Deering, S., *Internet Protocol Version 6 (IPv6) Addressing Architecture*, **RFC 3513**, April 2003

[RFC3596] Thomson, S. and Huitema, C. and Ksinant, V. and Souissi, M., *DNS Extensions to Support IP Version 6*, **RFC 3596**, October 2003

[RFC3748] Aboba, B. and Blunk, L. and Vollbrecht, J. and Carlson, J. and Levkowetz, H., *Extensible Authentication Protocol (EAP)*, **RFC 3748**, June 2004

[RFC3819] Karn, P. and Bormann, C. and Fairhurst, G. and Grossman, D. and Ludwig, R. and Mahdavi, J. and Montenegro, G. and Touch, J. and Wood, L., *Advice for Internet Subnetwork Designers*, **RFC 3819**, July 2004

[RFC3828] Larzon, L-A. and Degermark, M. and Pink, S. and Jonsson, L-E. and Fairhurst, G., *The Lightweight User Datagram Protocol (UDP-Lite)*, **RFC 3828**, July 2004

[RFC3927] Cheshire, S. and Aboba, B. and Guttman, E., *Dynamic Configuration of IPv4 Link-Local Addresses*, **RFC 3927**, May 2005

[RFC3931] Lau, J. and Townsley, M. and Goyret, I., *Layer Two Tunneling Protocol - Version 3 (L2TPv3)*, **RFC 3931**, March 2005

[RFC3971] Arkko, J. and Kempf, J. and Zill, B. and Nikander, P., *SEcure Neighbor Discovery (SEND)*, **RFC 3971**, March 2005

[RFC3972] Aura, T., *Cryptographically Generated Addresses (CGA)*, **RFC 3972**, March 2005

[RFC3986] Berners-Lee, T. and Fielding, R. and Masinter, L., *Uniform Resource Identifier (URI): Generic Syntax*, **RFC 3986**, January 2005

[RFC4033] Arends, R. and Austein, R. and Larson, M. and Massey, D. and Rose, S., *DNS Security Introduction and Requirements*, **RFC 4033**, March 2005

[RFC4193] Hinden, R. and Haberman, B., *Unique Local IPv6 Unicast Addresses*, **RFC 4193**, Oct. 2005

[RFC4251] Ylonen, T. and Lonvick, C., *The Secure Shell (SSH) Protocol Architecture*, **RFC 4251**, Jan. 2006

[RFC4264] Griffin, T. and Huston, G., *BGP Wedgies*, **RFC 4264**, Nov. 2005

[RFC4271] Rekhter, Y. and Li, T. and Hares, S., *A Border Gateway Protocol 4 (BGP-4)*, **RFC 4271**, Jan. 2006

[RFC4291] Hinden, R. and Deering, S., *IP Version 6 Addressing Architecture*, **RFC 4291**, Feb. 2006

[RFC4301] Kent, S. and Seo, K., *Security Architecture for the Internet Protocol*, **RFC 4301**, Dec. 2005

[RFC4302] Kent, S., *IP Authentication Header*, **RFC 4302**, Dec. 2005

[RFC4303] Kent, S., *IP Encapsulating Security Payload (ESP)*, **RFC 4303**, Dec. 2005

[RFC4340] Kohler, E. and Handley, M. and Floyd, S., *Datagram Congestion Control Protocol (DCCP)*, **RFC 4340**, March 2006

[RFC4443] Conta, A. and Deering, S. and Gupta, M., *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, **RFC 4443**, March 2006

[RFC4451] McPherson, D. and Gill, V., *BGP MULTI_EXIT_DISC (MED) Considerations*, **RFC 4451**, March 2006

[RFC4456] Bates, T. and Chen, E. and Chandra, R., *BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP)*, **RFC 4456**, April 2006

[RFC4614] Duke, M. and Braden, R. and Eddy, W. and Blanton, E., *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*, **RFC 4614**, Oct. 2006

[RFC4648] Josefsson, S., *The Base16, Base32, and Base64 Data Encodings*, **RFC 4648**, Oct. 2006

[RFC4822] Atkinson, R. and Fanto, M., *RIPv2 Cryptographic Authentication*, **RFC 4822**, Feb. 2007

[RFC4838] Cerf, V. and Burleigh, S. and Hooke, A. and Torgerson, L. and Durst, R. and Scott, K. and Fall, K. and Weiss, H., *Delay-Tolerant Networking Architecture*, **RFC 4838**, April 2007

[RFC4861] Narten, T. and Nordmark, E. and Simpson, W. and Soliman, H.,'Neighbor Discovery for IP version 6 (IPv6)', **RFC 4861**, Sept. 2007

[RFC4862] Thomson, S. and Narten, T. and Jinmei, T., *IPv6 Stateless Address Autoconfiguration*, **RFC 4862**, Sept. 2007

[RFC4870] Delany, M., *Domain-Based Email Authentication Using Public Keys Advertised in the DNS (DomainKeys)*, **RFC 4870**, May 2007

[RFC4871] Allman, E. and Callas, J. and Delany, M. and Libbey, M. and Fenton, J. and Thomas, M., *DomainKeys Identified Mail (DKIM) Signatures*, **RFC 4871**, May 2007

[RFC4941] Narten, T. and Draves, R. and Krishnan, S., *Privacy Extensions for Stateless Address Autoconfiguration in IPv6*, **RFC 4941**, Sept. 2007

[RFC4944] Montenegro, G. and Kushalnagar, N. and Hui, J. and Culler, D., *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*, **RFC 4944**, Sept. 2007

[RFC4952] Klensin, J. and Ko, Y., *Overview and Framework for Internationalized Email*, **RFC 4952**, July 2007

[RFC4953] Touch, J., *Defending TCP Against Spoofing Attacks*, **RFC 4953**, July 2007

[RFC4954] Simeborski, R. and Melnikov, A., *SMTP Service Extension for Authentication*, **RFC 4954**, July 2007

[RFC4963] Heffner, J. and Mathis, M. and Chandler, B., *IPv4 Reassembly Errors at High Data Rates*, **RFC 4963**, July 2007

[RFC4966] Aoun, C. and Davies, E., *Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status*, **RFC 4966**, July 2007

[RFC4987] Eddy, W., *TCP SYN Flooding Attacks and Common Mitigations*, **RFC 4987**, Aug. 2007

[RFC5004] Chen, E. and Sangli, S., *Avoid BGP Best Path Transitions from One External to Another*, **RFC 5004**, Sept. 2007

[RFC5065] Traina, P. and McPherson, D. and Scudder, J., *Autonomous System Confederations for BGP*, **RFC 5065**, Aug. 2007

[RFC5068] Hutzler, C. and Crocker, D. and Resnick, P. and Allman, E. and Finch, T., *Email Submission Operations: Access and Accountability Requirements*, **RFC 5068**, Nov. 2007

[RFC5072] Varada, S. and Haskins, D. and Allen, E., *IP Version 6 over PPP*, **RFC 5072**, Sept. 2007

[RFC5095] Abley, J. and Savola, P. and Neville-Neil, G., *Deprecation of Type 0 Routing Headers in IPv6*, **RFC 5095**, Dec. 2007

[RFC5227] Cheshire, S., *IPv4 Address Conflict Detection*, **RFC 5227**, July 2008

[RFC5234] Crocker, D. and Overell, P., *Augmented BNF for Syntax Specifications: ABNF*, **RFC 5234**, Jan. 2008

[RFC5321] Klensin, J., *Simple Mail Transfer Protocol*, **RFC 5321**, Oct. 2008

[RFC5322] Resnick, P., *Internet Message Format*, **RFC 5322**, Oct. 2008

[RFC5340] Coltun, R. and Ferguson, D. and Moy, J. and Lindem, A., *OSPF for IPv6*, **RFC 5340**, July 2008

[RFC5598] Crocker, D., *Internet Mail Architecture*, **RFC 5598**, July 2009

[RFC5646] Phillips, A. and Davis, M., *Tags for Identifying Languages*, **RFC 5646**, Sept. 2009

[RFC5681] Allman, M. and Paxson, V. and Blanton, E., *TCP congestion control*, **RFC 5681**, Sept. 2009

[RFC5735] Cotton, M. and Vegoda, L., *Special Use IPv4 Addresses*, **RFC 5735**, January 2010

[RFC5795] Sandlund, K. and Pelletier, G. and Jonsson, L-E., *The RObust Header Compression (ROHC) Framework*, **RFC 5795**, March 2010

[RFC6077] Papadimitriou, D. and Welzl, M. and Scharf, M. and Briscoe, B., *Open Research Issues in Internet Congestion Control*, **RFC 6077**, February 2011

[RFC6068] Duerst, M., Masinter, L. and Zawinski, J., *The 'mailto' URI Scheme* , **RFC 6068**, October 2010

[RFC6144] Baker, F. and Li, X. and Bao, X. and Yin, K., *Framework for IPv4/IPv6 Translation*, **RFC 6144**, April 2011

[RFC6265] Barth, A., *HTTP State Management Mechanism*, **RFC 6265**, April 2011

[RFC6274] Gont, F., *Security Assessment of the Internet Protocol Version 4*, **RFC 6274**, July 2011

[RG2010] Rhodes, B. and Goerzen, J., Foundations of Python Network Programming: The Comprehensive Guide to Building Network Applications with Python, Second Edition, Academic Press, 2004

[RJ1995] Ramakrishnan, K. K. and Jain, R., A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. SIGCOMM Comput. Commun. Rev. 25, 1 (Jan. 1995), 138-156.

[RIB2013] Raiciu, C., Iyengar, J., Bonaventure, O., Recent Advances in Reliable Transport Protocols, in H. Haddadi, O. Bonaventure (Eds.), Recent Advances in Networking, (2013), pp. 59-106.

[RY1994] Ramakrishnan, K.K. and Henry Yang, The Ethernet Capture Effect: Analysis and Solution, Proceedings of IEEE 19th Conference on Local Computer Networks, MN, Oct. 1994.

[Roberts1975] Roberts, L., ALOHA packet system with and without slots and capture. SIGCOMM Comput. Commun. Rev. 5, 2 (Apr. 1975), 28-42.

[Ross1989] Ross, F., An overview of FDDI: The fiber distributed data interface, IEEE J. Selected Areas in Comm., vol. 7, no. 7, pp. 1043-1051, Sept. 1989

[Russel06] Russell A., Rough Consensus and Running Code and the Internet-OSI Standards War, IEEE Annals of the History of Computing, July-September 2006

[SAO1990] Sidhu, G., Andrews, R., Oppenheimer, A., Inside AppleTalk, Addison-Wesley, 1990

[SARK2002] Subramanian, L., Agarwal, S., Rexford, J., Katz, R.. Characterizing the Internet hierarchy from multiple vantage points. In IEEE INFOCOM, 2002

[Sechrest] Sechrest, S., An Introductory 4.4BSD Interprocess Communication Tutorial, 4.4BSD Programmer's Supplementary Documentation

[SG1990] Scheifler, R., Gettys, J., X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD, X Version 11, Release 4, Digital Press

[SGP98] Stone, J., Greenwald, M., Partridge, C., and Hughes, J., Performance of checksums and CRC's over real data. IEEE/ACM Trans. Netw. 6, 5 (Oct. 1998), 529-543.

[SH1980] Shoch, J. F. and Hupp, J. A., Measured performance of an Ethernet local network. Commun. ACM 23, 12 (Dec. 1980), 711-721.

[SH2004] Senapathi, S., Hernandez, R., Introduction to TCP Offload Engines, March 2004

[SMKKB2001] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H., Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01). ACM, New York, NY, USA, 149-160

[SMM1998] Semke, J., Mahdavi, J., and Mathis, M., Automatic TCP buffer tuning. SIGCOMM Comput. Commun. Rev. 28, 4 (Oct. 1998), 315-323.

[SPMR09] Stigge, M., Plotz, H., Muller, W., Redlich, J., Reversing CRC - Theory and Practice. Berlin: Humboldt University Berlin. pp. 24.

[STBT2009] Sridharan, M., Tan, K., Bansal, D., Thaler, D., Compound TCP: A New TCP Congestion Control for High-Speed and Long Distance Networks, Internet draft, work in progress, April 2009

[STD2013] Stewart, R., Tuexen, M., Dong, X., ECN for Stream Control Transmission Protocol (SCTP), Internet draft, draft-stewart-tsvwg-sctpecn-04, April 2013, work in progress

[Seifert2008] Seifert, R., Edwards, J., The All-New Switch Book : The complete guide to LAN switching technology, Wiley, 2008

[Selinger] Selinger, P., *MD5 collision demo*, http://www.mscs.dal.ca/~selinger/md5collision/

[SFR2004] Stevens R. and Fenner, and Rudoff, A., UNIX Network Programming: The sockets networking API, Addison Wesley, 2004

[Sklower89] Sklower, K. 1989. Improving the efficiency of the OSI checksum calculation. SIGCOMM Comput. Commun. Rev. 19, 5 (Oct. 1989), 32-43.

[SMASU2012] Sarrar, N., Maier, G., Ager, B., Sommer, R. and Uhlig, S., Investigating IPv6 traffic, Passive and Active Measurements, Lecture Notes in Computer Science vol 7192, 2012, pp.11-20

[SMM98] Semke, J., Mahdavi, J., and Mathis, M., Automatic TCP buffer tuning. SIGCOMM Comput. Commun. Rev. 28, 4 (Oct. 1998), 315-323.

[Stevens1994] Stevens, R., TCP/IP Illustrated : the Protocols, Addison-Wesley, 1994

[Stevens1998] Stevens, R., UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI, Prentice Hall, 1998

[Stewart1998] Stewart, J., BGP4: Inter-Domain Routing In The Internet, Addison-Wesley, 1998

[Stoll1988] Stoll, C., Stalking the wily hacker, Commun. ACM 31, 5 (May. 1988), 484-497.

[SV1995] 13. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin SIGCOMM Comput. Commun. Rev. 25, 4 (October 1995), 231-242.

[TE1993] Tsuchiya, P. F. and Eng, T., Extending the IP internet through address reuse. SIGCOMM Comput. Commun. Rev. 23, 1 (Jan. 1993), 16-33.

[Thomborson1992] Thomborson, C., The V.42bis Standard for Data-Compressing Modems, IEEE Micro, September/October 1992 (vol. 12 no. 5), pp. 41-53

[Unicode] The Unicode Consortium. The Unicode Standard, Version 5.0.0, defined by: The Unicode Standard, Version 5.0 (Boston, MA, Addison-Wesley, 2007

[VPD2004] Vasseur, J., Pickavet, M., and Demeester, P., Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS. Morgan Kaufmann Publishers Inc., 2004

[Varghese2005] Varghese, G., Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices, Morgan Kaufmann, 2005

[Vyncke2007] Vyncke, E., Paggen, C., LAN Switch Security: What Hackers Know About Your Switches, Cisco Press, 2007

[WB2008] Waserman, M., Baker, F., *IPv6-to-IPv6 Network Address Translation (NAT66)*, Internet draft, November 2008, http://tools.ietf.org/html/draft-mrw-behave-nat66-02

[WMH2008] Wilson, P., Michaelson, G., Huston, G., *Redesignation of 240/4 from "Future Use" to "Private Use"*, Internet draft, September 2008, work in progress, http://tools.ietf.org/html/draft-wilson-class-e-02

[WMS2004] White, R., Mc Pherson, D., Srihari, S., Practical BGP, Addison-Wesley, 2004

[Watson1981] Watson, R., Timer-Based Mechanisms in Reliable Transport Protocol Connection Management. Computer Networks 5: 47-56 (1981)

[Williams1993] Williams, R. *A painless guide to CRC error detection algorithms*, August 1993, unpublished manuscript, http://www.ross.net/crc/download/crc_v3.txt

[Winston2003] Winston, G., NetBIOS Specification, 2003

[WY2011] Wing, D. and Yourtchenko, A., *Happy Eyeballs: Success with Dual-Stack Hosts*, Internet draft, work in progress, July 2011, http://tools.ietf.org/html/draft-ietf-v6ops-happy-eyeballs-03

[X200] ITU-T, recommendation X.200, Open Systems Interconnection - Model and Notation, 1994

[X224] ITU-T, recommendation X.224, Information technology - Open Systems Interconnection - Protocol for providing the connection-mode transport service, 1995

[XNS] Xerox, Xerox Network Systems Architecture, XNSG058504, 1985

[Zimmermann80] Zimmermann, H., OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Transactions on Communications, vol. 28, no. 4, April 1980, pp. 425 - 432.

Symbols

::, 173
::1, 173
100BaseTX, 215
10Base2, 215
10Base5, 214
10BaseT, 215
802.11 frame format, 225
802.5 data frame, 97
802.5 token frame, 96

A

abrupt connection release, 60, 71
Additive Increase Multiplicative Decrease (AIMD), 101
address, 29
address learning, 216
Address resolution problem, 187
addressing, 71
adhoc network, 224
AF_INET, 238
AF_INET6, 238
AF_UNSPEC, 238
AIMD, **289**
ALOHA, 85
Alternating Bit Protocol, 16
anycast, **289**
API, **289**
Application layer, 109
ARP, **289**
ARPANET, **289**
ascii, **289**
ASN.1, **289**
ATM, **289**

B

Base64 encoding, 120
Basic Service Set (BSS), 224
beacon frame (802.11), 227
BGP, 199, **289**
BGP Adj-RIB-In, 202
BGP Adj-RIB-Out, 202
BGP decision process, 205
BGP KEEPALIVE, 201
BGP local-preference, 205
BGP nexthop, 204
BGP NOTIFICATION, 201
BGP OPEN, 201
BGP peer, 200
BGP RIB, 202
BGP UPDATE, 201
binary exponential back-off (CSMA/CD), 90
bit stuffing, 9
black hole, 30
BNF, **289**
Border Gateway Protocol, 199
bridge, 216
broadcast, **289**
BSS, 224

C

Carrier Sense Multiple Access, 86
Carrier Sense Multiple Access with Collision Avoidance, 91
Carrier Sense Multiple Access with Collision Detection, 87
character stuffing, 10
Checksum computation, 139
CIDR, **289**
Clear To Send, 94
collision, 84
collision detection, 88
collision domain, 215
congestion collapse, 79, 99
congestion control, 75
congestion window, 105
connection establishment, 58
connectionless service, 56
control plane, 30
count to infinity, 46
CSMA, 86
CSMA (non-persistent), 87
CSMA (persistent), 87
CSMA/CA, 91
CSMA/CD, 87
CTS, 94
CTS frame (802.11), 226
cumulative acknowledgements, 19
customer-provider peering relationship, 198

D

data plane, 30
Datalink layer, 9, 107
delayed acknowledgements, 151
Denial of Service, 144
DHCPv6, 188, 190
dial-up line, **289**
DIFS, 92
Distance vector, 44
Distributed Coordination Function Inter Frame Space, 92
DNS, **289**, **290**
DNS message format, 113
Duplicate Address Detection, 188

E

EAP, 211
eBGP, **290**
EGP, **290**
EIFS, 92
EIGRP, **290**
electrical cable, 5
email message format, 116
Ending Delimiter (Token Ring), 96
Ethernet bridge, 216
Ethernet DIX frame format, 212
Ethernet hub, 215
Ethernet switch, 216
Ethernet Type field, 212
EtherType, 212
exponential backoff, 151
export policy, 199
Extended Inter Frame Space, 92
Extensible Authentication Protocol, 211

F

Fairness, 99
Fast Ethernet, 215
FDM, 84
FECN, 82
Five layers reference model, 107
Forward Explicit Congestion Notification, 82
forwarding loop, 30
forwarding table, 29
frame, 9, 107, **290**
Frame-Relay, **290**
framing, 9
Frequency Division Multiplexing, 84
FTP, **290**
ftp, **290**

G

getaddrinfo, 238
go-back-n, 19
graceful connection release, 60, 71

H

head-of-line blocking, 156
Hello message, 49
hidden station problem, 94
hop-by-hop forwarding, 29
hosts.txt, 72, **290**
HTML, **290**
HTTP, **290**
hub, **290**

I

IANA, **290**
iBGP, **290**
ICANN, **290**
IETF, **290**
IGP, **290**
IGRP, **290**
IMAP, **290**
import policy, 199
independent network, 224
infrastructure network, 224
interdomain routing policy, 199
Internet, **290**
internet, **290**
inverse query, **290**
IP, **290**
IPv4, **291**
IPv4 fragmentation and reassembly, 178
IPv6, **291**
IPv6 fragmentation, 177
IPv6 Renumbering, 189
IS-IS, **291**
ISN, **291**
ISO, **291**
ISO-3166, **291**
ISP, **291**
ITU, **291**
IXP, **291**

J

jamming, 88
jumbogram, 177

L

label switching, 41
LAN, **291**
large window, 148
leased line, **291**
Link Local address, 173
link-local IPv6 address, 186
link-state routing, 48
LLC, 214
Logical Link Control (LLC), 214

M

MAC address learning, 216
MAC address table (Ethernet switch), 216
MAN, **291**
Manchester encoding, 8
max-min fairness, 78, 99

Maximum Segment Lifetime (MSL), 64
maximum segment lifetime (MSL), 25, 69
Maximum Segment Size, 145
Maximum Transmission Unit, 178
medium access control, 75
message-mode data transfer, 59
MIME, **291**
MIME document, **291**
minicomputer, **291**
modem, **291**
Monitor station, 97
monomode optical fiber, 5
MSS, 145, **291**
MTU, 178
multicast, **291**
multimode optical fiber, 5

N

Nagle algorithm, 147
nameserver, **291**
naming, 71
NAT, **291**
NBMA, 44, 169, **291**
NDP, 187
Neighbor Discovery Protocol, 187
Neighbor Solicitation message, 187
Neighbour Discovery Protocol, 186
network congestion, 79
Network Information Center, 72
Network layer, 108
network-byte order, **292**
NFS, **292**
Non-Broadcast Multi-Access Networks, 44, 169
non-persistent CSMA, 87
NTP, **292**

O

Open Shortest Path First, 193
optical fiber, 5
Organisation Unique Identifier, 212
OSI, **292**
OSI reference model, 109
OSPF, 193, **292**
OSPF area, 193
OSPF Designated Router, 195
OUI, 212

P

packet, 108, **292**
packet discard mechanism, 81
packet radio, 85
packet size distribution, 148
Path MTU discovery, 184
PBL, **292**
peer-to-peer, 56
persistence timer, 25, 69
persistent CSMA, 87
Physical layer, 8, 107
physical layer, 107
piggybacking, 25
ping6, 184
Point-to-Point Protocol, 210
POP, **292**
port-address table, 31
portmapper, 137
Post Office Protocol, 124
PPP, 210
Provider Aggregatable address, 171
Provider Independent address, 171

R

record route, 37
Reference models, 106
reliable connectionless service, 56
Remote Procedure Call, 62
Request To Send, 94
request-response service, 61
resolver, **292**
RFC
    RFC 1032, 72, 301
    RFC 1035, 73, 113, 114, 289, 301
    RFC 1042, 227, 301
    RFC 1055, 210, 301
    RFC 1071, 139, 231, 301
    RFC 1094, 292
    RFC 1122, 109, 140, 142, 146, 152, 212, 301
    RFC 1144, 210, 301
    RFC 1149, 71, 301
    RFC 1169, 301
    RFC 1191, 301
    RFC 1195, 193, 301
    RFC 1258, 142, 301
    RFC 1305, 292
    RFC 1321, 231, 301
    RFC 1323, 146, 148–150, 154, 302
    RFC 1347, 169, 302
    RFC 1350, 244
    RFC 1518, 289, 302
    RFC 1519, 172, 302
    RFC 1542, 302
    RFC 1548, 210, 302
    RFC 1550, 169, 302
    RFC 1561, 169, 302
    RFC 1621, 169, 302
    RFC 1624, 302
    RFC 1631, 169, 302
    RFC 1661, 27, 168, 302
    RFC 1662, 210, 302
    RFC 1710, 169, 175, 302
    RFC 1738, 127, 133, 302
    RFC 1752, 169, 302
    RFC 1812, 102, 302
    RFC 1819, 169, 302
    RFC 1831, 134, 137
    RFC 1832, 134, 135
    RFC 1833, 137

RFC 1889, 302
RFC 1896, 119, 302
RFC 1918, 173, 302
RFC 1939, 124, 125, 254, 292, 302
RFC 1945, 129, 130, 302
RFC 1948, 143, 302
RFC 1951, 180, 302
RFC 1981, 182, 184, 302
RFC 20, 10, 54, 120, 301
RFC 2003, 302
RFC 2018, 153, 302
RFC 2045, 118, 120, 291, 302
RFC 2046, 118, 119, 302
RFC 2050, 303
RFC 2080, 192, 193, 275, 303
RFC 2082, 303
RFC 2131, 190, 303
RFC 2140, 147, 150, 303
RFC 2225, 303
RFC 2328, 193, 195, 292, 303
RFC 2332, 303
RFC 2364, 211, 303
RFC 2368, 303
RFC 2402, 177
RFC 2406, 177
RFC 2453, 192, 292, 303
RFC 2460, 139, 175, 177, 180, 181, 303
RFC 2464, 214, 303
RFC 2507, 303
RFC 2516, 211, 303
RFC 2581, 153, 303
RFC 2616, 122, 129–131, 254, 290, 303
RFC 2617, 133, 303
RFC 2622, 199, 303
RFC 2675, 177
RFC 2711, 177
RFC 2766, 303
RFC 2821, 122
RFC 2854, 119, 303
RFC 2965, 303
RFC 2988, 145, 149–151, 303
RFC 2991, 196, 303
RFC 3021, 303
RFC 3022, 303
RFC 3031, 303
RFC 3168, 102, 141, 163–165, 303
RFC 3187, 126
RFC 3234, 303
RFC 3235, 303
RFC 3286, 159
RFC 3309, 304
RFC 3315, 190, 304
RFC 3330, 304
RFC 3360, 155, 304
RFC 3390, 163, 304
RFC 3490, 304
RFC 3501, 124, 290, 304
RFC 3513, 170, 304
RFC 3540, 164
RFC 3550, 137
RFC 3596, 115, 304
RFC 3708, 160
RFC 3736, 191
RFC 3748, 211, 304
RFC 3758, 157
RFC 3782, 167
RFC 3819, 168, 304
RFC 3828, 304
RFC 3927, 173, 304
RFC 3931, 304
RFC 3971, 190, 304
RFC 3972, 190, 304
RFC 3986, 126, 133, 304
RFC 4033, 113, 304
RFC 4151, 126
RFC 4193, 173, 304
RFC 4251, 125, 304
RFC 4253, 292
RFC 4264, 208, 304
RFC 4271, 200, 201, 304
RFC 4287, 126
RFC 4291, 173, 175, 186, 187, 304
RFC 4301, 304
RFC 4302, 304
RFC 4303, 304
RFC 4340, 304
RFC 4443, 180, 182, 184, 270, 304
RFC 4451, 305
RFC 4456, 305
RFC 4614, 139, 305
RFC 4627, 134, 135
RFC 4632, 289
RFC 4634, 231
RFC 4648, 120, 305
RFC 4822, 305
RFC 4838, 305
RFC 4861, 187, 305
RFC 4862, 188, 305
RFC 4870, 124, 305
RFC 4871, 124, 305
RFC 4941, 188, 305
RFC 4944, 180, 305
RFC 4952, 120, 305
RFC 4953, 155, 305
RFC 4954, 121, 124, 305
RFC 4960, 137, 156–160
RFC 4963, 184, 305
RFC 4966, 305
RFC 4987, 145, 305
RFC 5004, 305
RFC 5065, 305
RFC 5068, 124, 305
RFC 5072, 210, 305
RFC 5082, 190
RFC 5095, 180, 305
RFC 5227, 305

RFC 5234, 55, 289, 305
RFC 5246, 293
RFC 5321, 121, 122, 253, 305
RFC 5322, 116, 117, 122, 305
RFC 5340, 193, 273, 292, 305
RFC 5531, 292
RFC 5598, 116, 306
RFC 5646, 306
RFC 5681, 161, 162, 306
RFC 5735, 306
RFC 5795, 180, 210, 306
RFC 5880, 299
RFC 5890, 73, 120
RFC 6068, 127, 306
RFC 6077, 167, 306
RFC 6106, 191
RFC 6144, 306
RFC 6164, 204
RFC 6214, 212
RFC 6265, 133, 306
RFC 6274, 212, 306
RFC 6275, 177
RFC 6437, 176
RFC 6824, 156
RFC 6928, 163
RFC 7010, 189
RFC 768, 138, 301
RFC 789, 50, 301
RFC 791, 55, 142, 301
RFC 792, 301
RFC 793, 71, 139, 140, 142, 146, 147, 150, 151, 155, 263, 301
RFC 813, 152, 301
RFC 819, 72, 301
RFC 821, 292, 301
RFC 822, 118, 119
RFC 826, 289, 301
RFC 854, 293
RFC 867, 261
RFC 868, 243
RFC 879, 145, 301
RFC 893, 212, 301
RFC 894, 214, 227, 301
RFC 896, 79, 99, 148, 301
RFC 952, 72, 301
RFC 959, 122, 125, 290, 301
RFC 974, 301
RIP, 192, **292**
RIR, **292**
Robustness principle, 155
root nameserver, **292**
round-trip-time, **292**
router, **292**
Routing Information Protocol, 192
RPC, 62, **292**
RTS, 94
RTS frame (802.11), 226

S

scheduler, 82
scheduling algorithm, 82
SCTP, 156
SCTP chunk, 157
SCTP common header, 157
SCTP CWR chunk, 165
SCTP data chunk, 159
SCTP ECN Echo chunk, 165
SCTP SACK chunk, 159
SCTP segment, 157
SCTP Selective Acknowledgement chunk, 159
SCTP TSN, 159
SDU (Service Data Unit), **292**
segment, 63, 108, **292**
selective acknowledgements, 24
selective repeat, 21
sendto, 238
sequence number, 16
Serial Line IP, 210
Service Set Identity (SSID), 227
shared-cost peering relationship, 199
Short Inter Frame Spacing, 91
sibling peering relationship, 199
SIFS, 91
SLAC, 188
slot time (Ethernet), 90
slotted ALOHA, 86
slotTime (CSMA/CA), 92
SMTP, **292**
SNMP, **292**
SOCK_DGRAM, 238
SOCK_STREAM, 238
socket, 238, **292**
socket.bind, 241
socket.close, 239
socket.connect, 239
socket.recv, 239
socket.recvfrom, 241
socket.send, 239
socket.shutdown, 239
source routing, 35
speed of light, 88
split horizon, 47
split horizon with poison reverse, 47
spoofed packet, **292**
SSH, **292**
SSID, 227
standard query, **292**
Starting Delimiter (Token Ring), 96
Stateless Address Autoconfiguration, 188
stream-mode data transfer, 60
stub domain, 197
stuffing (bit), 9
stuffing (character), 10
switch, 216, **292**
SYN cookie, **292**
SYN cookies, 144

T

TCB, **292**
TCP, 139, **292**
TCP Connection establishment, 141
TCP connection release, 154
TCP fast retransmit, 152
TCP header, 140
TCP Initial Sequence Number, 142
TCP MSS, 145
TCP Options, 145
TCP RST, 143
TCP SACK, 153
TCP selective acknowledgements, 153
TCP self clocking, 98
TCP SYN, 141
TCP SYN+ACK, 141
TCP/IP, **293**
TCP/IP reference model, 109
telnet, **293**
Tier-1 ISP, 209
Time Division Multiplexing, 85
time-sequence diagram, 6
TLD, **293**
TLS, **293**
Token Holding Time, 97
Token Ring data frame, 97
Token Ring Monitor, 96
Token Ring token frame, 96
traceroute6, 184
transit domain, 197
Transmission Control Block, 147
Transmission Sequence Number, 159
transport clock, 64
Transport layer, 108
two-way connectivity, 52

U

UDP, 137, **293**
UDP Checksum, 139
UDP segment, 138
unicast, **293**
Unique Local Unicast IPv6, 173
unreliable connectionless service, 56

V

Virtual LAN, 222
VLAN, 222
vnc, **293**

W

W3C, **293**
WAN, **293**
Wavelength Division Multiplexing, 85
WDM, 85
WiFi, 223

X

X.25, **293**
X11, **293**
XDR, 134
XML, **293**