Computer Networking : Principles, Protocols and Practice

Release 0.0

Olivier Bonaventure

April 26, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

When the first computers were built during the second world war, they were expensive and isolated. However, after about twenty years, as their prices were decreasing and the first experiments were started to connect computers together. In 1964, Paul Baran and Donald Davies published independently the first papers describing the idea of building computer networks. Given the cost of computers, sharing them over a long distance was an interesting idea. In the US, the *ARPANET* started in 1969 and continued until the mid 1980s. In France, Louis Pouzin developed the Cyclades network. During the 1970s, the telecommunication and the computer industry became interested in computer networks. The telecommunication industry bet on the X25. Many local area networks such as Ethernet or Token Ring were designed at that time. During the 1980s, the need to interconnect more and more computers lead most computer vendors to develop their own suite of networking protocols. Xerox developed XNS, DEC chose DECNet, IBM developed SNA, Microsoft introduced NetBIOS, Apple bet on Appletalk, ... In the research community, ARPANET was decommissioned and replaced by TCP/IP and the reference implementation was developed inside BSD Unix. Universities who were already running Unix could thus adopt TCP/IP easily and vendors of Unix workstations such as Sun or Silicon Graphics included TCP/IP in their variant of Unix. In parallel, the *ISO*, with support from the governments, worked on developing an open [1] suite of networking protocols. In the end, TCP/IP became the de facto standard that is used not only within the research community. During the 1990s and the early 2000s, the growth of the usage of TCP/IP continued and today proprietary protocols are seldom used. As shown by the figure below that provides the estimation of the number of hosts attached to the Internet, the Internet sustained a huge growth during the last 20+ years.

Recent estimations of the number of hosts attached to the Internet show a continuing growth since 20+ years. However, although the number of hosts attached to the Internet is high, it should be compared to the number of mobile phones that are in use today. More and more of these mobile phones will be connected to the Internet. Furthermore, thanks to the availability of TCP/IP implementations requiring limited resources such as uIP , we can expect to see a growth of the TCP/IP enabled embedded devices.

Before looking at the services that are provided by computer networks, it is useful to agree on some terminology that is widely used in the networking literature. First, computer networks are often classified as function of the geographical area that they cover

- *LAN* : a local area network typically interconnects hosts that are up to a few or maybe a few tens of kilometers apart.
- *MAN* : a metropolitan area network typically interconnects devices that are up to a few hundred kilometers apart
- *WAN* : a wide area network interconnect hosts that can be located anywhere on Earth

Another classification of computer networks is based on their physical topology. In the following figures, physical links are represented as lines while boxes show computers or other types of networking equipment.

[1] open in ISO terms was in contrast with the proprietary protocol suites whose specification was not always available. The US government even mandated the usage of the OSI protocols (see **RFC 1169**), but this was not sufficient to encourage all users to switch to the OSI protocol suite that was considered by many as too complex compared to other protocol suites.

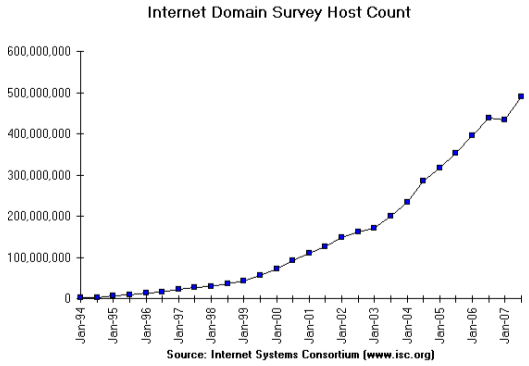


Figure 1.1: Estimation of the number of hosts on the Internet

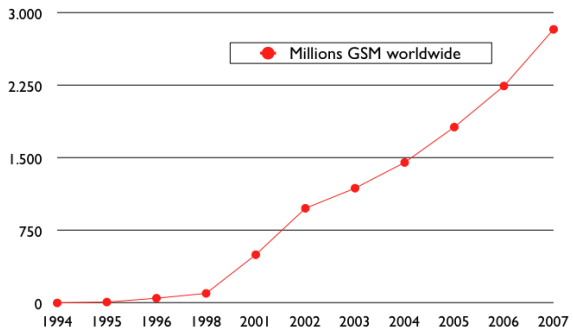
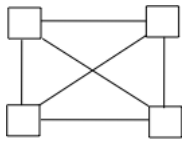


Figure 1.2: Estimation of the number of mobile phones

Computer networks are used to allow several hosts to exchange information between themselves. In order to allow any host to send messages to any other host in the network, the easiest solution is to organise them as a full-mesh with a direct and dedicated link between each pair of hosts. Such a physical topology is sometimes used, especially when high performance and high redundancy is required for a small number of hosts. However, it has two major drawbacks :

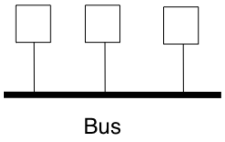
- for a network containing *n* hosts, each host must have *n-1* physical interfaces. Thus, in practice the number of physical interfaces on a node will limit the size of a full-mesh network that can be built
- for a network containing *n* hosts, *n*(n-1)/2* links are required. This is possible when there are a few nodes in the same room, but rarely when they are located several kilometers apart



Full-mesh

Figure 1.3: A Full mesh network

The second possible physical organisation, which is 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 hosts sends a signal on the bus, the signal is received by all hosts attached to the bus. A drawback of bus-based networks is that if the bus is physically cut, then the network is split in two different networks. For this reason, bus-based networks are sometimes considered to be difficult to manage especially when the cable is long and there are many places where it can break. Such as bus-based topology is used for example by Ethernet networks.



Bus

Figure 1.4: A network organised as a Bus

A third organisation of a computer network is a star topology. In such topologies, the 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 passive device such as an equipment that amplifies electrical signal or an active device such as an 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 manage than bus-shaped networks.

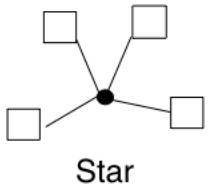


Figure 1.5: A network organised as a Star

A fourth physical organisation is the Ring. As with the bus, each host has a single physical interface that connects 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 viewpoint, a single ring is not the best solution as the signal travels only in one direction on the ring. If one of the links that compose the ring is cut, then the entire network fails. This makes failure easier to detect than in bus-based networks. In practice, such rings have been used in local area networks, but nowadays they are often replaced by star-shaped networks. In metropolitan are 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 switched quickly to the other 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.

In practice, most real networks combine part of these topologies. For example, a campus network can be organised as a ring between the key buildings while smaller buildings are attached as a tree or a star to important buildings. Or an ISP network may have a full mesh of devices in the core of its network and trees to connect remote users.

Throughout this book, our objective will be to understand the protocols and mechanisms that are necessary for a network such as the one shown below.

The last point of terminology that we need to discuss are the transmission modes. When exchanging information through a network, we often distinguish three transmission modes. In TV and radio transmission, *broadcast* is often used to indicate a technology that send a video or radio signal to all receivers in a given geographical area. Broadcast is sometimes used in computer networks, but only in local area networks where the number of recipients is limited.

The first and most widespread transmission mode is called *unicast* . In the unicast transmission mode, information is sent by one sender to one receiver. Most of today's Internet applications rely on the unicast transmission mode.

A second mode of transmission is the *multicast* transmission mode. This mode is used when the same information

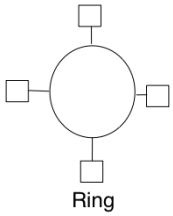


Figure 1.6: A network organised as a Ring

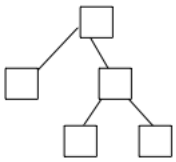


Figure 1.7: A network organised as a Tree

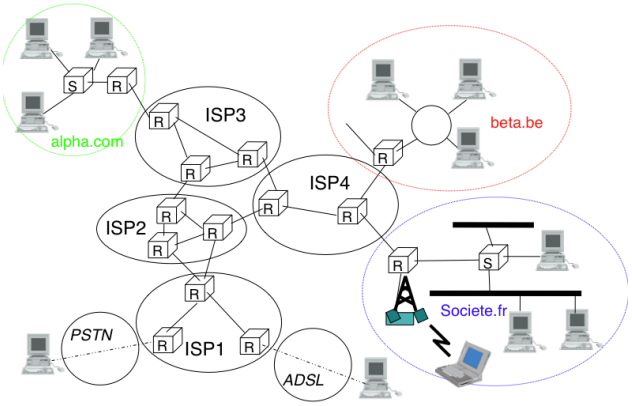


Figure 1.8: A simple internetwork

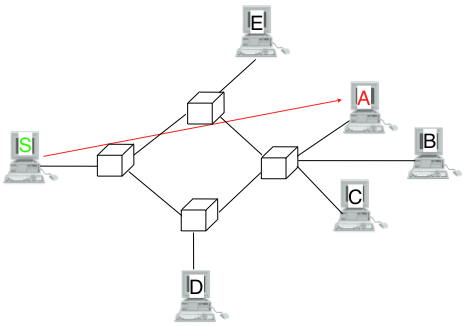


Figure 1.9: Unicast transmission

must be sent to a set of recipients. It was first used in LANs but became later supported in wide area networks. When a sender uses multicast to send information to ‘N‘receivers, the sender sends a single copy of the information and the network nodes duplicate the information whenever necessary so that it can reach all the recipients that belong to the destination group.

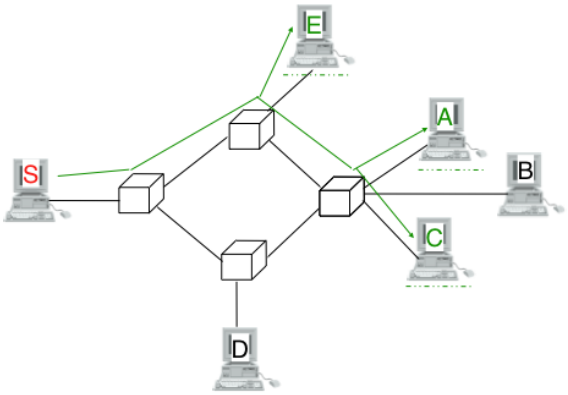


Figure 1.10: Multicast transmission

The last transmission mode is the *anycast* transmission mode. It was initially defined in **RFC 1542**. In this transmission mode, a set of receivers is identified. When a source sends information towards this set of receivers, the network ensures that the information is delivered to *one* receiver that belongs to this set. Usually the receiver that is closest to the source is the one that receives the information sent by this particular source. The anycast transmission mode is useful to ensure redundancy as when one of the receivers fails, the network will ensure that information will be delivered to another receiver belonging to the same group. However, in practice supporting the anycast transmission mode can be difficult.

1.1 Services and protocols

An important point to understand before studying computer networks is the different between a *service* and a *protocol*.

To understand the difference between the two, it is useful to start from the real world. The traditional Post provides a service which is to deliver letters to recipients. The service can be more precisely

In computer networks, the notion of service is more formally defined in [X200] . It can be better understood by considering a computer networks, what ever its size or complexity as a black box that provides a service to ‘ users‘ as shown in the figure below. These users could be human users or processes running on a computer system. Many users can be attached to the same service provider. Through this provider, each user must be able to exchange messages with any other user. To be able to deliver these messages, the service provider must be able to unambiguously identify each user. In computer networks, each user is identified by a unique *address*. We will discuss later how these addresses are built and used. At this point, and when considering unicast transmission, the important characteristics

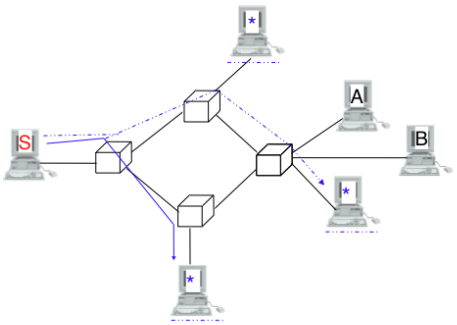


Figure 1.11: Anycast transmission

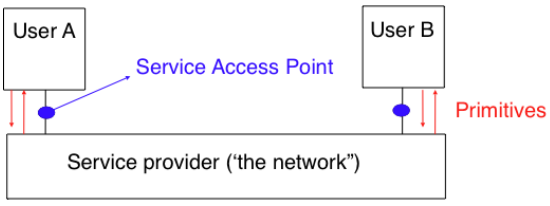


Figure 1.12: Users and service provider

of these *addresses* is that they are unique. Two different users attached to the network cannot have the same address. Throughout this book, we will define a service as a set of capabilities provided by a system (and its underlying elements) to its user. A user interacts with a service through a *service access point*. Note that as shown in the figure above, the users interact with one service provider. In practice, the service provider is distributed over several hosts, but these are implementation details that are not important at this stage. These interactions between a user an a service provider are expressed in [X200] by using primitives as show in the figure below. These primitives are an abstract representation of the interactions between a user and a service provider. In practice, these interactions could be implemented as system calls for example. Four types of primitives are defined :

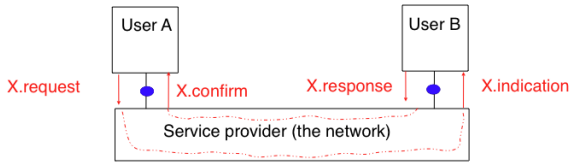


Figure 1.13: The four types of primitives

- *X.request*. This type of primitive corresponds to a request issued by a user to a service provider
- *X.indication*. This type of primitive is generated by the network provider and delivered to a user (often related to an earlier and remote *X.request* primitive)
- *X.response*. This type of primitive is generated by a user to answer to an earlier *X.indication* primitive
- *X.confirm*. This type of primitive is delivered by the service provide to confirm to a user that a previous *X.request* has been successfully processed.

Primitives can be combined to model different types of services. The simplest service in computer networks is the connectionless service. This service can be modelled by using two primitives :

- *Data.request(source,destination,SDU)*. This primitive is issued by a user that specifies as parameters its (source) address, the address of the recipient of the message and the message itself. We will use *Service Data Unit* (SDU) to name the message that is exchanged transparently between two users of a service.
- *Data.indication(source,destination,SDU)*. This primitive is delivered by a service provider to a user. It contains as parameters a *Service Data Unit* as well as the addresses of the sender and the destination users.

When discussing about the service provided in a computer network, it is often useful to be able to describe the interactions between the users and the provider graphically. A frequently used representation is the time-sequence diagram. In this chapter and later throughout the book, we will often use diagrams such as the figure below. A time-sequence diagram describes the interactions between two users and a service provider. By convention, the users are represented in the left and right parts of the diagram while the service provider occupies the middle of the diagram. In such a time-sequence diagram, time flows from the top of the bottom of the diagram. Each primitive is represented by a plain horizontal arrow to which the name of the primitive is attached. The dashed lines are used to represent the possible relationship between two (or more) primitives. Such a diagram provides information about the ordering of the different primitives, but the distance between two primitives does not represent a precise amount of time. The

figure below provides a representation of the connectionless service. The user on the left, having address *S*, issues a *Data.request* primitive containing 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

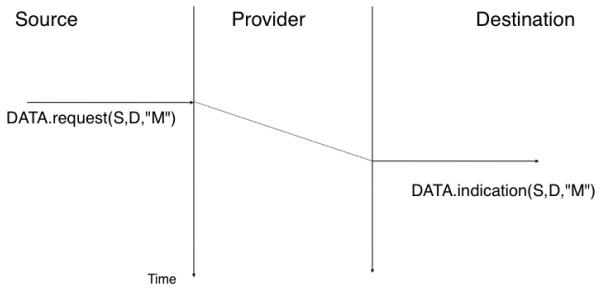


Figure 1.14: A simple connectionless service

of the connectionless service that we will discuss later in this book. 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 be eventually delivered to their destination. Such a service would be very useful for users, but guaranteeing perfect delivery is difficult in practice. For this reason, computer networks usually support an *unreliable connectionless service*.

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

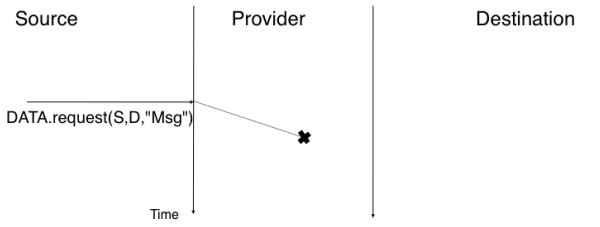


Figure 1.15: An unreliable connectionless service may loose SDUs

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 unreliable connectionless

service providers may deliver twice or even more a SDU sent by a user. This is illustrated by the time-sequence diagram below.

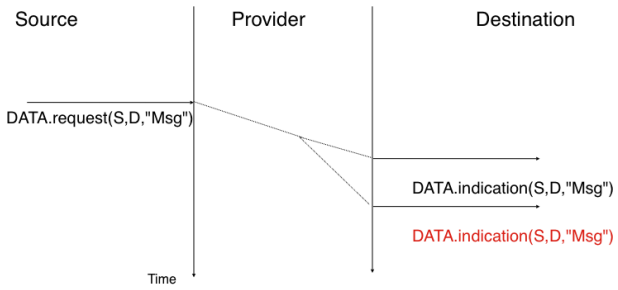


Figure 1.16: An unreliable connectionless service may duplicate SDUs

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

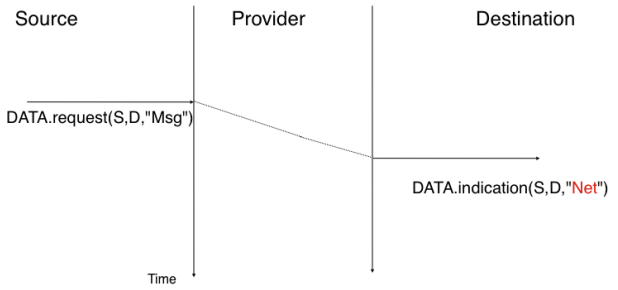


Figure 1.17: An unreliable connectionless service may deliver erroneous SDUs

When a user interacts with a service provider, it must know precisely the limitations of the underlying service to be able to overcome any problem that may arise. This requires a precise definition of the characteristics of the underlying service. Another important characteristic of the connectionless service is whether it preserves the ordering of the SDUs sent by one user. From the user's viewpoint, this is often a desirable characteristic. This is illustrated in the figure below.

However, many connectionless services, and in particular the unreliable connection services do not guarantee that they will always preserve the ordering of the SDUs sent by each user. This is illustrated in the figure below. The *connectionless service* is widely used in computer networks as we will see in the next chapter. Several variations to this basic service have been proposed. One of these is the *confirmed connectionless service*. This service uses a *Data.confirm* primitive in addition to the classical *Data.request* and *Data.indication* primitives. This primitive is

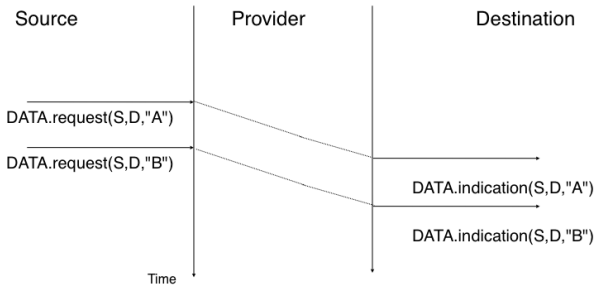


Figure 1.18: A connectionless service that preserves the ordering of SDUs sent by a given user

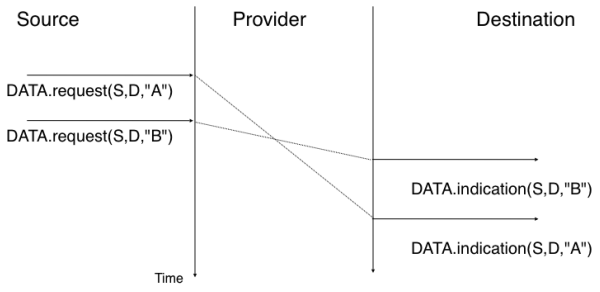


Figure 1.19: An connectionless service that does not preserve the ordering of SDUs sent by a given user

issued by the service provider to confirm to a user the delivery of a previously sent SDU to its recipient. Note that, like the registered service of the post office, the *Data.confirm* only indicates that the SDU has been delivered to the destination user. The *Data.confirm* primitive does not indicate whether the SDU has been processed by the destination user. This *confirmed connectionless service* is illustrated in the figure below. The *connectionless service* that we

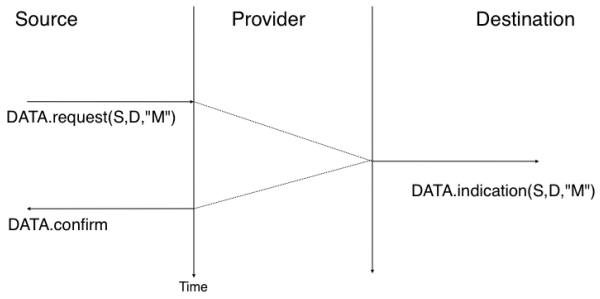


Figure 1.20: A confirmed connectionless service

have described earlier is frequently used by users who need to exchange small SDUs. Users who need to either send or receive several different and potentially large SDUs or who need structured exchanges often prefer the *connection-oriented service*. An invocation of the *connection-oriented service* is divided in 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 that supports the exchange of SDUs between the two users that are associated through the *connection*. This *data transfer* phase is the second phase of a connection. The third phase is the termination of the connection. Once the users have finished to exchange SDUs, they request the service provider to terminate the connection. As we'll see later, there are also some cases where the service provider may need to terminate itself a connection.

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 open and the destination user can start to send 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.

The example above shows a successful connection establishment. However, in practice not all connections are successfully established. A first 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. 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

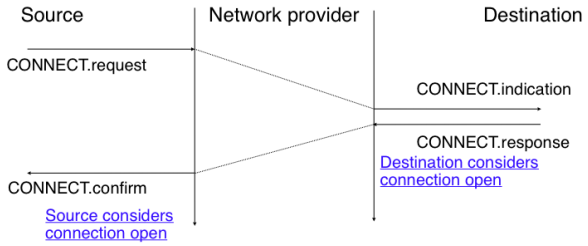


Figure 1.21: Connection establishment

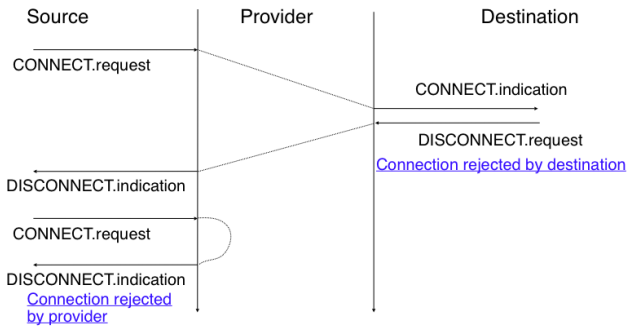


Figure 1.22: Two types of rejection for a connection establishment attempt

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. Unfortunately, the *message-mode* transfer is not widely used on the Internet. On the Internet, the

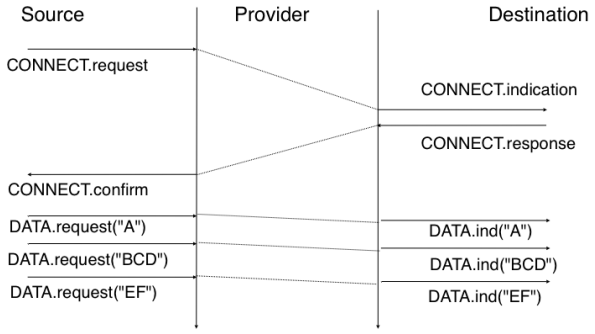


Figure 1.23: Message-mode transfer in a connection oriented service

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 user. The sending user sends bytes by using *Data.request* primitives that contain groups 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 ordering 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 competence 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 delineate these SDUs in the byte stream that it receives. The third phase of a connection is when it needs to be released. As a connection involves

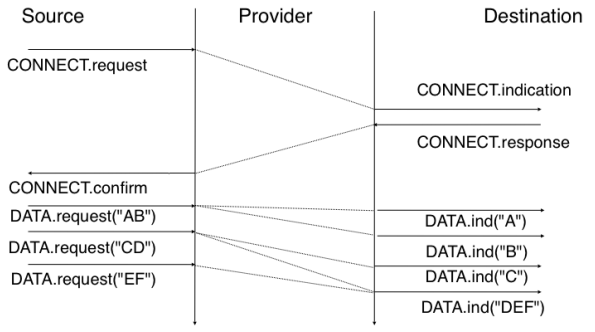


Figure 1.24: Stream-mode transfer in a connection oriented service

three parties (two users and a service provider), any of them can request the termination of the connection. Usually, connections are terminated upon request of one user. 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. 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.

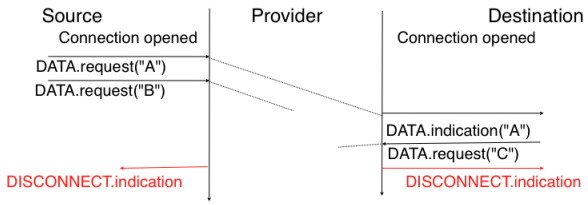


Figure 1.25: Abrupt connection release initiated by the service provider

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 issues a *Disconnect.request* primitive and requests an abrupt release. The service provider will process the request, stop the two data streams and deliver the *Disconnect.indication* 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

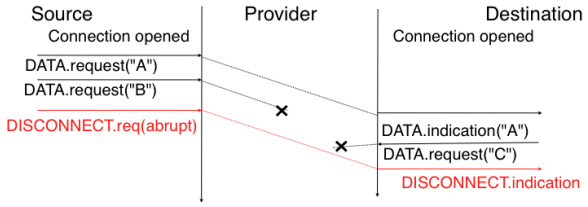


Figure 1.26: Abrupt connection release initiated by a user

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 once it has sent all the SDUs that it planned over this connection, but still continue to receive SDUs over the other stream. This *graceful* connection release is usually performed as shown in the figure below. One user issues a *Disconnect.request*

primitives to its provider once it has issued all its *Data.request* primitives. The service provider will wait until all *Data.indication* have been delivered to the receiving user before issuing the *Disconnnect.indication* primitive. This primitive informs the receiving user that he will not receive anymore SDUs over this connection, but he is still able to issue *Data.request* primitives on the stream in the opposite direction. Once the user has issued all his *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 *Data.indication* to the other user once it has delivered all the pending *Data.indication* primitives. At this point, the two streams have been released successfully and the connection is completely closed.

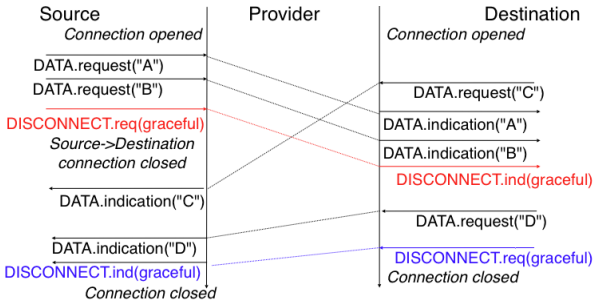


Figure 1.27: Graceful connection release

Reliability of the connection-oriented service

An important point to discuss about the connection-oriented service is its reliability. In practice, a *connection-oriented* can only guarantee the correct delivery of all SDUs if 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.

1.2 The reference models

Given the growing complexity of computer networks, network researchers proposed during the 1970s reference models that allow to describe network protocols and services. The Open Systems Interconnection (OSI) model [Zimmermann80] was probably the most influential one. It was 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 [2].

1.2.1 The five layers reference model

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

Starting from the bottom, the first layer is the Physical layer. Two communicating devices are linked through a physical medium. The physical medium is used to transfer and electrical or optical signal between the two devices. Different types of physical mediums are used in practice :

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

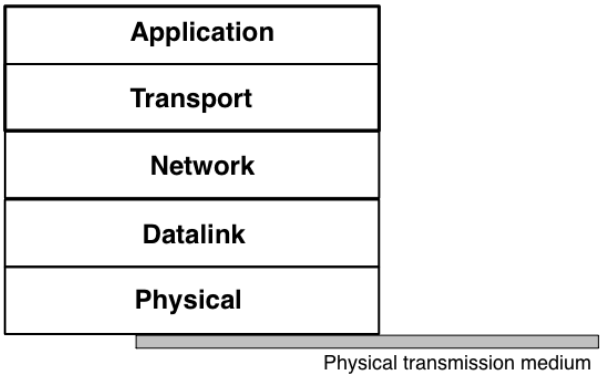


Figure 1.28: The five layers of the reference model

- *electrical cable*. Information can be transmitted over different types of electrical cables. The most common ones are twisted pairs that are used in the telephone network, but also in enterprise networks and coaxial cables. Coaxial cables are still used in cable TV networks, but not anymore in enterprise networks.
- *optical fiber*. Optical fibers are frequently used in public and enterprise networks with the distance is larger than one kilometer.
- *wireless*. In this case, a radio signal is used to encode the information being exchanged between the communicating devices.

Note: Additional information about the physical layer will be added later.

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

The last two points may seem strange at first glance. When two devices are attached through a cable, how is it possible for bits to be created or lost on such a cable ?

This is mainly due to the fact that the communicating devices use their own clock to transmit bits at a given bandwidth. Consider a sender having a clock that ticks one million times per second and sends one bit every tick. Every microsecond, the sender sends an electrical or optical signal that encodes one bit. The sender's bandwidth is thus 1 Mbps. If the receiver clock ticks exactly [3] every microsecond, it will also deliver 1 Mbps to its user. However, if the receiver's clock is slightly faster (resp. slower), than it will deliver slightly more (resp. less) than one million bits every second.

[3] Having perfectly synchronised clocks running at a high frequency is very difficult in practice. However, some physical layers introduce a feedback loop that allows the receiver's clock to synchronise itself automatically to the sender's clock. However, not all physical layers include this kind of synchronisation.

Bandwidth

In computer networks, the bandwidth achievable through the physical layer is always expressed in bits per second. A Mega bps is one million bits per second and a Giga bps 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 1.048 seconds.

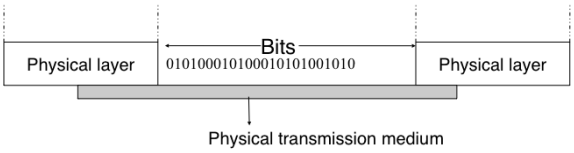


Figure 1.29: The Physical layer

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 because virtually any information can be encoded as a sequence of bits. Electrical engineers are used to process 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 in several layers as well. Hard-disks store sectors of 512 bytes or more. Unix filesystems groups sectors in larger blocks that can contain data or inodes that represent 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 and each layer provides a service that it built above the underlying layer and is closer to the needs of the applications. 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* user 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 a 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* 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 physical that will be sent over the physical medium. The *physical layer* on the right 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 not transmission errors, this entity will receive the frame sent earlier. 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*. The network layer entities exchange *packets*. A *packet* is a finite sequence of bytes. A packet usually contains information about its

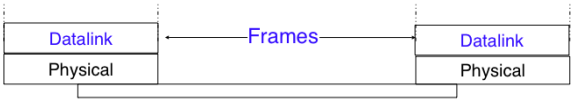


Figure 1.30: The Datalink layer

origin and its destination. A packet usually passes through several intermediate devices called routers on its way from its origin to its destination.

Different types of network layers can be implemented. The Internet uses an unreliable connectionless network layer service. Other networks have used reliable and unreliable connection-oriented network layer services. Most realisa-

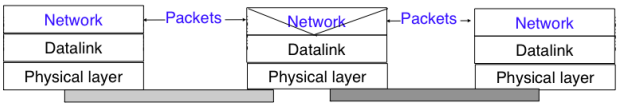


Figure 1.31: The network layer

tions, including the Internet, of the network layer do not provide a reliable service. However, many applications need to exchange information reliably and using the network layer service directly would be very difficult for them. Ensuring a 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. 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 on the Internet are *TCP* that provides a reliable connection-oriented bytestream transport service and *UDP* that provides an unreliable connection-less transport service. The upper layer of our architecture is the *Application layer*. It includes all the mechanisms and data structures that are necessary for the applications. We will use Application Data Unit (ADU) to indicate the data exchanged between two entities of the Application layer.

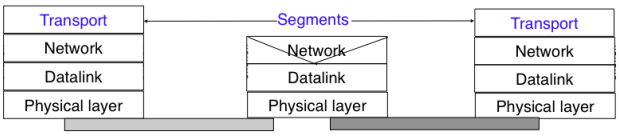


Figure 1.32: The transport layer

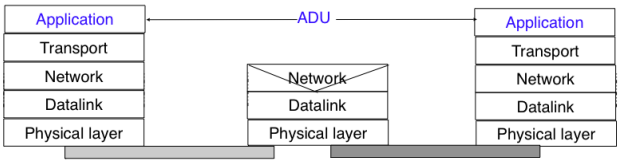


Figure 1.33: The Application layer

1.2.2 The TCP/IP reference model

In contrast with OSI, the TCP/IP community did not spend a lot of effort at defining a detailed reference model and in fact the goals of the Internet architecture were only documented after TCP/IP had been deployed [Clark88]. **RFC 1122** that 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.

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.

1.2.3 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 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 failure 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 **[#funicode]_**. The situation is even more complex when considering the exchange of structured information such as records. To solve this problem, the Presentation layer contains provides for common representation of the data transferred. The *ASN.1* notation was designed for the Presentation layer.
- the Application layers 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.

Where are the missing layers in TCP/IP reference model ?

The TCP/IP reference places the Presentation and the Session layers implicitly in the Application layer. The main motivations for simplifying the upper layers in the TCP/IP reference model were pragmatic. Most Internet applications started as prototypes that evolved and were standardised later. Many of these applications assumed that they would be used to exchange information written in American English and for which the 7 bits US-ASCII character code was sufficient. This was the case for email, but as we'll see in the next chapter email was able to evolve to support different characters encodings. Some applications considered the different data representations explicitly. For example, *ftp* contained mechanisms to convert a file from one format to another. On the other hand, many ISO specifications were developed by committees composed of people who did not all participate in actual implementations. ISO spent a lot of effort at analysing the requirements and defining a solution that meets all these requirements. Sometimes, the specification was so complex that it was difficult to implement it completely...

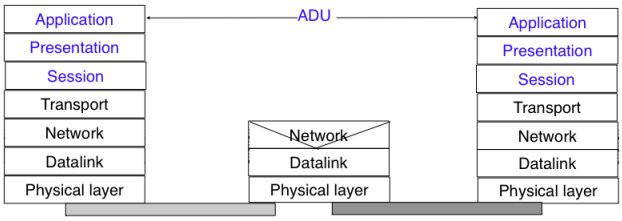


Figure 1.34: The seven layers of the OSI reference model

1.2.4 Organisation of the document

This document is organised according to the *TCP/IP* reference model and follows a top-down approach. Most of the first networking textbooks chose a bottom-up approach, i.e. they first explained all the electrical and optical details of the physical layer then moved to the datalink layer, ... This approach worked well during the infancy of computer networks and until the late 1990s. At that time, most students were not users of computer networks and it was useful to explain computer networks by building the corresponding protocols from the simplest in the physical layer up to the application layer. Today, all students are active users of Internet applications and starting to learn computer networking by looking at bits is not very motivating. Starting from [KuroseRoss09], many textbooks and teachers have chosen a top-down approach. This approach starts from the applications such as email and web that students already know and explores the different layers starting from the application layer. This approach works pretty well with today's students.

Top-down versus bottom-up

The traditional bottom-up approach could be in fact considered as an engineering approach since it starts from the simple network that allows to exchange bits and explains how to combine different protocols and mechanisms to build the most complex applications. The top-down approach could on the other hand be considered as a scientific approach. Like biologists, it starts from an existing (man-built) system and explores it layer by layer.

Besides the top-down versus bottom-up organisation, computer networking books can aim at having an in-depth coverage of a small number of topics or at having a limited coverage of a wide range of topics. Covering a wide range of topics is interesting for introductory courses or for students who do not need a detailed knowledge of computer networks. It allows the students to learn a *little about everything* and then start from this basic knowledge later if they need to understand computer networking in more details. This books chose to cover in details a smaller number of topics than other textbooks. This is motivated by the fact that computer networks are often pushed to their limits and understanding the details of the main networking protocols is important to be able to fully grasp how a network behaves or extend it to provide innovative services. As the popular quote says, *the devil is in the details* and this quote is even more important in computer networking where the change of a single bit may have huge consequences. In computer networks, understanding *all* the details is, unfortunately for some students, sometimes necessary.

The overall objective of the book is to explain the principles and the protocols used in computer networks and also provide the students with some intuition about the important practical issues that arise often. The course follows a hybrid problem-based learning (*PBL*) approach. During each week, the students follow a 2 hours theoretical course that describes the principles and some of the protocols. They also receive a set of small problems that they need to solve in groups. These problems are designed to reinforce the student's knowledge but also to explore the practical

problems that arise in real networks by allowing the students to perform experiments by writing prototype networking code. Most of the prototype code will be written in python by using the scapy packet injection/manipulation framework that will be described later.

Why open source ?

This book is being developed as an open-source book under a creative commons licence. This choice an an open-source license is motivated by two reasons. The first is that we hope that this will allow many students to use the book to learn computer networks and maybe other teachers will reuse, adapt and improve it. The second reason is that that the computer networking community heavily relies on open source implementations. In fact, there are high-quality and widely used open-source implementations for most of the protocols described in this book. This includes the TCP/IP implementations that are part of linux, freebsd or the uIP stack running on 8bits controllers, but also servers such as bind, unbound, apache or sendmail and implementations of routing protocols such as xorp or quagga . Furthermore, the official specifications of most of the protocols that are described in this book have been developed within the IETF in an almost open-source manner. The IETF publishes its protocols specifications in the publicly available RFC and new proposals are described in Internet drafts.

The book is organised as follows. We first describe the application layer. Given the large number of Internet-based applications, it is of course impossible to cover them all in details. Instead we focus on three types of Internet-based applications. We first study the Domain Name System (DNS) and then explain some of the protocols involved in the exchange of electronic mail. The discussion of the application layer ends with a description of the key protocols of the world wide web and a brief explanation of peer-to-peer applications. All these applications rely on the transport layer. This is a key layer in today's networks as it contains all the mechanisms that are necessary to provide a reliable delivery of data over an unreliable network. We cover the transport layer by first developing a simple reliable transport layer protocol and then explain the details of the TCP and UDP protocols used in TCP/IP networks. After the transport layer, we focus on the network layer. This is also a very important layer as it is responsible for the delivery of packets from any source to any destination through intermediate routers. In the network layer, we describe the two possible organisations of the network layer and the routing protocols based on link-state and distance vectors. Then we explain in details the IPv4, IPv6, RIP, OSPF and BGP protocols that are actually used in today's Internet. The last part of the course is devoted to the datalink layer. More precisely, our focus in this part is on the Local area networks. We first describe the Medium Access Control mechanisms that allow multiple hosts to share a given transmission medium. We consider both opportunistic and deterministic techniques. We explain in details two types of LANs that are important from a deployment viewpoint today : Ethernet and WiFi.

CHAPTER

TWO

THE APPLICATION LAYER

The Application Layer is the most important and most visible layer in computer networks. Applications reside in this layer and human users interact via those applications through the network.

In this chapter, we first briefly describe the main principles of the application layer and focus on the two most important models : the client-server model and the peer-to-peer models. Then, we review in details two families of protocols that have proved to be very useful in networks such as the Internet : electronic mail and the protocols that allow to access information on the world wide web. We also describe the Domain Name System that allows humans to use user-friendly names while the hosts use IP addresses.

2.1 Principles

The are two important models 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 provide responses. It is illustrated in the figure below.

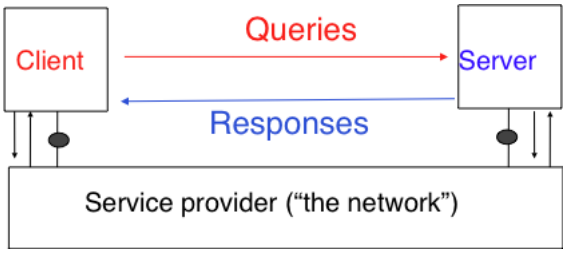


Figure 2.1: The client-server model

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 was used by tens or more users at the same time. Each user was interacting via the minicomputer by using a terminal. Those terminals, were mainly a screen, a keyboard and a cable connected to the minicomputer.

There are various types of servers and various types of clients. A web server provides information in response to the query sent by the client. 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 in the above figure), but in practice these messages are exchanged thanks to the

underlying protocols (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. To ensure that the server is able to understand the queries sent by the client and that the client is able to understand the responses sent by the server, they must agree on a set of syntactical and semantic rules that define the format of the messages that they exchange and 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*

This conversation can succeed provided that 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 who's 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.*

Application-layer protocols can exchange two types of messages. Some protocols such as those used to support electronic mail exchange messages that are 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. The most common ones are *CR* and *LF* that are used to terminate a line or the *Bell* character that 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 set to *0*. Bytes are always transmitted starting from the high order or most significant bit.

Besides characters, some applications also need to exchange 16 bits and 32 bits fields such as IPv4 addresses. A naive solution would have been to send the 16- or 32-bits field as it was encoded in 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** and refer to this encoding as the *network-byte order*. Most libraries [1] used to write networked applications contain functions to convert multibyte fields from memory to the network byte order and vice versa.

Besides 16 and 32 bits words, some applications need to exchange that contain 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 as 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*.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       First field  (16 bits)  |A|B|C|D|E|F|G|H|   Second      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       field (24 bits)         |  First Byte   | Second Byte   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 and 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.

We will discuss several examples of application-level protocols in this chapter.

2.1.1 The transport services

Networked applications are built on top of the transport service. As explained in the previous chapter, there are two main types of transport services :

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

[1] 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 programming languages.

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 an SDU that has been modified by transmission errors.

The connectionless transport service allows networked application to exchange messages. Several networked applications may be running at the same time on a given 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. As we will explain in chapter *xx* 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. 130.104.32.107. IPv6 addresses are usually represented as a set of hexadecimal numbers separated by by semicolons, e.g. 2001:6a8:3080:2:217:f2ff:fed6:65c0. Today, most Internet hosts have an 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. Another possibility to identify an Internet host is by using its *fully qualified domain name* (usually summarised as *name*). The figure below illustrates two that are using the datagram service provided by UDP on hosts with IPv4 addresses.

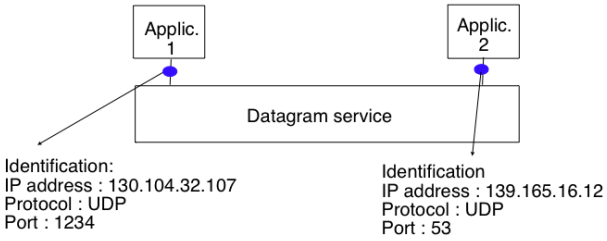


Figure 2.2: The connectionless or datagram service

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. As for the datagram service, the networked applications that are using the byte-stream service are identified by the host where they run and a port number. The hosts can be identified by an IPv4 address, an IPv6 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. Some applications use SCTP instead of TCP for the byte-stream service.

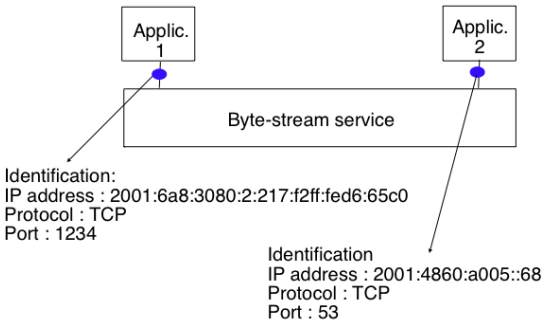


Figure 2.3: The connection-oriented or byte-stream service

2.2 Application-level protocols

Many protocols have been defined for networked applications. In this section, we describe some of the important applications that are used on the Internet. We first explain the domain name systems that enables hosts to be identified by human-friendly names instead of the IPv4 or IPv6 addresses that are used by the network. Then we describe the operation of electronic mail, one of the first killer applications on the global Internet and the main protocol used on world wide web. In the last section, we show how simple networked clients and servers can be written in python .

2.2.1 The Domain Name System

In the early days of the Internet, there were only few hosts (mainly minicomputers) connected to the network. The most popular applications were remote login and file transfer. In 1983, there were already five hundred hosts attached to the network. Each of these hosts was 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 us them when needed. Programming languages use named variables that allow programmers to ignore their exact location in memory. Networked applications must also be able to use names instead of IP addresses.

A first solution to allow applications to use names was the *hosts.txt* file. This file contained the mapping between the name of each Internet host and its associated IPv4 address(s) [2]. It was maintained by SRI International that coordinated the Network Information Center. When a new host was connected to the network, the system administrator had to register the name of the host and its IP address at the NIC. The NIC updated the *hosts.txt* file on its server. All Internet hosts retrieved regularly the updated *hosts.txt* from the server maintained by SRI. This file was stored at a well-known location on each Internet host **RFC 952** and networked applications could use it to find the IP address corresponding to a name.

The *hosts.txt* file can be used when there a 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 that usually contained the name of the institution and a suffix to identify the host inside the institution. On the ARPANet few institutions had several hosts connected to the network.

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

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 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* [3]. 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 letters ISO-3166 country codes as top-level domain names. Since ISO-3166 defines a two letters 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 or *.tv* for Tuvalu, a group of small islands in the Pacific and *.tm* for Turkmenistan. 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 Catalonia region in Spain. 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 first-come first served, an allow anyone to register domain names, but there are some exceptions. For example, *.gov* is reserved for the US government.

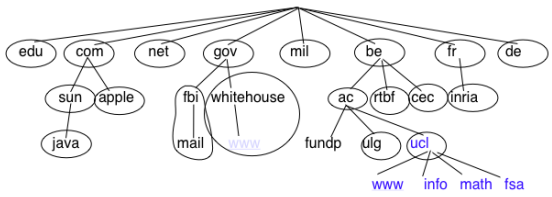


Figure 2.4: The tree of domain names

RFC 1035 clarified the definition of the fully qualified domain names by using the following *BNF*

```
<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 low
<digit> ::= any one of the ten digits 0 through 9
```

This grammar speficies that a domain name is an ordered list of labels separated by the dot (.) character. Each label can contain letters, numbers and the hyphen character (-) but must start with a letter [4]. 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

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

[4] This specification evolved later to support domain names written by using other character sets than us-ASCII **RFC 3490**. This extension is important to support other languages than English, but a detailed discussion is outside the scope of this document.

ending with the root implicitly at the right. The top-level domain name must be one of the registered TLDs [5]. 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 answer to the queries sent by the client. 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* and three hosts inside domain *a.sdom1.dom*.

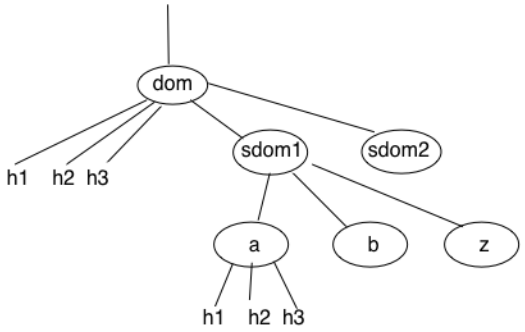


Figure 2.5: A simple tree of domain names

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 DNS server(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*)

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

[5] 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

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 [6].

Each root nameserver maintains the list [7] of all the nameservers that are responsible for each of the top-level domain names and their IP addresses [8]. 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, ...

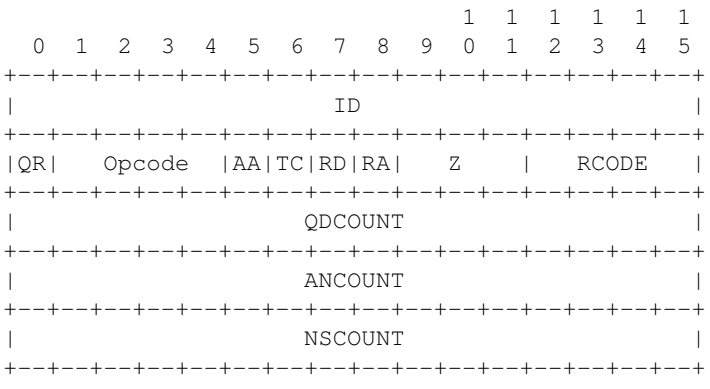
To be able to contact the root nameservers, each DNS client must know their IP addresses. This implies, that DNS clients must maintain an up-to-date list of the IP addresses of the root nameservers [9]. Without this list, it is impossible to contact the root nameservers. Forcing all Internet hosts to maintain the most recent version of this list would be difficult from an operational viewpoint. 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 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 find the *answer*.

DNS resolvers have several advantages over letting each Internet host query directly nameservers. First, regular Internet hosts do not need to maintain the up-to-date list of the IP addresses of the root servers. Second, 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.

The last component of the Domain Name System is the DNS protocol. The DNS protocol runs both above the datagram service and the bytestream service. In practice, the datagram service is used when short queries and responses are exchanged and the bytestream is used when longer responses are expected. In this section, we will only discuss the utilisation of the DNS protocol above the datagram service.

DNS messages are composed of five parts that are named sections in **RFC 1035**. The first three sections are mandatory and the last two 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 the servers that can provide authoritative answers if required. The last section contains addition information that 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.



[6] There are currently 13 root servers. In practice, some of these root servers are themselves implemented as a set of distinct physical servers. See http://www.root-servers.org/ for more information about the physical location of these servers.

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

[8] 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 2010, several DNS root servers are still not reachable by using IPv6.

[9] The current list of the IP addresses of the root nameservers is maintained at http://www.internic.net/zones/named.root . These IP addresses are stable and root nameservers seldom change their IP addresses. DNS resolvers must however maintain an up-to-date copy of this file.

```
|                      ARCOUNT                  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The *ID* (identifier) is a 16-bits 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. One utilisation of this field is to distinguish between a *standard query* in which a client sends a *name* and the server returns the corresponding *address* and an *inverse query* in which the client sends an *address* and the server returns the corresponding *name*.

The *AA* bit is set when the server that sent the response is an *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 that are responsible for a given domain. They always store the most recent information about a domain. *Non-authoritative* servers on the other are not directly managed by the owners of a domain. They may thus provide answers that are out of date. From a security viewpoint, the *authoritative* bit is not an indication about the validity of an answer. Securing the Domain Name Systems 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**. These extensions are outside the scope of this chapter and will be discussed later.

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*. 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 [10] 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 addition details. The last four field 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*.

All RRs have the same top level format shown below :

::

```
                                1 1 1 1 1 1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
/ / / NAME / | | +–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+ | TYPE | +–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+
+ | CLASS | +–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+ | TTL | | | +–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+
| RDLENGTH | +–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+| / RDATA / / / +–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+–+
+–+–+
```

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 environment 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 when load must be spread among several servers.

The *RDLength* field is the size 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 *NS*

[10] Some DNS resolvers allow any host to send queries. OpenDNS and GoogleDNS are example of open resolvers.

record contains the name of the DNS server that is responsible for a given domain. *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.

Reverse DNS and in-addr.arpa

The DNS is mainly used to find the IP 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, see **RFC 3596**.

2.2.2 Electronic mail

Electronic mail or email is a very popular application in computer networks such as the Internet. Email appeared in the early 1970s. It allows users to exchange messages. Initially, Email was mainly used to exchange short messages, but over the years its usage has grown. Email is now used to exchange very long messages but also messages 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 one of the available 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 a webmail.

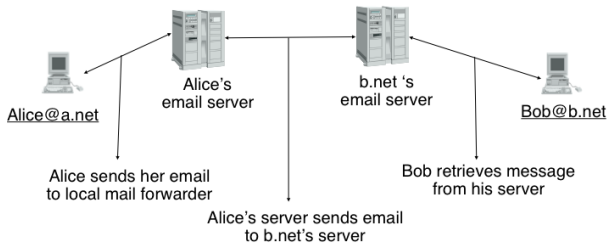


Figure 2.6: Architecture of the Internet email

In practice, an email system is composed of four components :

- a precise format to encode email messages
- protocols that allow 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 first discuss the format of email messages and then the protocols that are used on today's Internet to exchange and retrieve emails. Other email systems have been developed in the past, 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 for both email clients and email servers.

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

- the *header* that contains control information that is used by the email servers to deliver the email message to its recipient
- 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. 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.

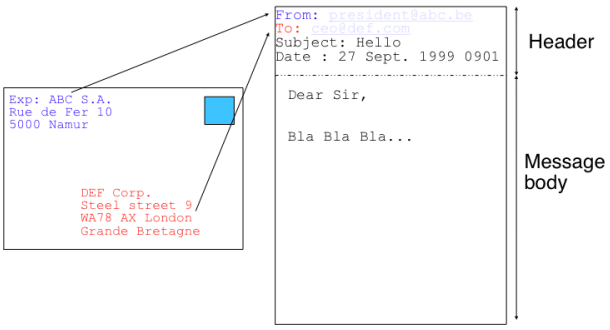


Figure 2.7: The structure of email messages

The email header contains several lines that all begin by a keyword followed by 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:*. It contains the (optional) name of the sender followed by its 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.

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

- the *To:* header line contains the list of the email addresses of the primary recipients of the message. Several addresses are 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 marks the boundary between the header and the body of the message.

Several other header lines are defined in **RFC 5322** and other optional header lines have been defined elsewhere [11]. Furthermore, many email software define their own header lines starting from *X-*. Several of these header lines 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 as *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.
- 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 was originated at a host named *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>
```

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

```
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, this became a severe limitation for two reasons. First, non-English speakers wanted to write emails in their mother language that often requires more characters than those of the ASCII character table. Second, many users wanted to send other content than 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 where 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.

Backward compatibility and the evolution of the Internet

The Internet protocols such as eBackward compatibility Although backward compatibility increases

RFC 2045 defines three new types of header lines that can appear inside the headers of email messages.

- 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 might be defined in the future. Thanks to this header line, software that process 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 rfc:*822* specification.
- The *Content-Type:* header line indicates that type of data that is carried inside the message.
- 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. Some servers were unable to process 8 bits ASCII characters and dropped such characters if they appeared. To solve this problem, several techniques were designed to map

The *Content-Type:* header line is used for two different purposes. When used inside the email header, it 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 an email message can naturally contain an empty line. Another possible option would be to define a special line, e.g. *-*-*-*-*-*-*-*-*-* 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 them 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 that contains two parts that are both in plain text and encoded by using the ASCII character set. Note that the string *simple boundary* is defined in the *Content-Type:* header as the string that marks the boundary between the header and the first part and also between the first and the second part and at the end of the message. Other 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 that may be found 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:* 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. 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.
- *audio*. The message part contains an audio clip. The subtype indicates the format of the audio clip.
- *video*. The message part contains a video clip. The subtype indicates the format of the video clip.
- *application*. The message part contains binary information that was produced by a particular application that is listed as the subtype. Email clients may use the subtype to launch the application that is able to decode the received binary information.

From ASCII to Unicode

The first computers used different techniques to represent characters in memory and on disk. During the 1950s, most computers were isolated. During the 1960s, computers became less and less isolated and started 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 sometimes difficult. The 7 bits ASCII character **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 move was the definition of ISO-8859 by **ISO_** This family of standards specified various character sets that allow to represent many European written languages by using 8 bits characters. Unfortunately, ISO-8859 was not able 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 allows to support all written languages that are used on Earth today. The Unicode standard [Unicode] has now been adopted by most computer and software vendors. It defines the standard way to encode characters. It can be expected that all applications, notably on the Internet, will support Unicode.

The last MIME header line is *Content-Transfer-Encoding:*. This header line is used after the *Content-Type:* header line in a message part. It 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*. They both allow to encode a sequence of bytes in 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 in groups of three bytes (with the last group being possibly partially filled). Each group of three bytes is divided in four six-bits fields and each six bits 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 sequence of bytes to be encoded are 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 = as a padding character. This character is used twice when the last group contains two bytes and once when the

last group of bytes 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. Do 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 responsible for the 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 ... *Bob*'s SMTP server will store the email sent by *Alice* until *Bob* retrieves the message by using a webmail interface or protocols such as the Post Office Protocol (*POP*) or the Internet Message Access Protocol (*IMAP*).

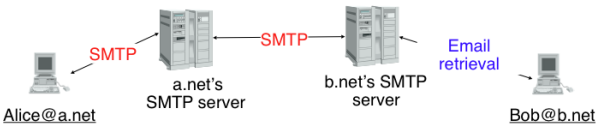


Figure 2.8: Email delivery protocols

The Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (*SMTP*) defined in **RFC 5321** is a client-server protocol. The SMTP specification distinguishes five types of hosts that are involved in the delivery of email messages. Email messages are composed on a Mail User Agent (MUA). In practice, the MUA is 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 [12], MSA-MTA and MTA-MTA.

SMTP is a text-based protocol like many other application-layer protocols on the Internet. SMTP use the byte-stream service and servers listen on port *21*. SMTP clients sends commands that are each composed of one line of ASCII text terminated by *CR+LF*. SMTP servers reply by sending ASCII lines that contain a three digits numerical error/success code and additional 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 following BNF rules.:

```
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
```

In this BNF, *atext* corresponds to the printable ASCII characters. This BNF rule is defined in **RFC 5322**. The five main commands are *HELO*, *MAIL FROM:*, *RCPT TO:*, *DATA* and *QUIT*. *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 returned by the SMTP server are defined by the following BNF rules

```
Greeting        = "220 " Domain [ SP textstring ] CRLF
textstring      = 1*(%d09 / %d32-126)
Reply-line      = *( Reply-code "-" [ textstring ] CRLF )
                  Reply-code [ SP textstring ] CRLF
Reply-code      = %x32-35 %x30-35 %x30-39
```

SMTP servers use structured reply codes. 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. For some reasons, indicated by the other digits or the comment, the command cannot be processed immediately, but there is some hope that the problem will be transient. This is a hint to the client that it should try again the same command 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 later 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**.

Example 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
```

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

```
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 the commands sent by the client. The fourth reply code would be sent by the server when the client sends command in an incorrect order (e.g. the client tries to send an email before providing the destination 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 TCP 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 TCP 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 is temporarily unavailable, for different reasons while reply code *550* indicates that the mailbox does no exist or cannot be used for policy reasons. Reply code *354* is sent to allow the client to transmit its email message.

The transfer of an email message is performed in three phases. During the first phase, the client opens a TCP connection with the server. Then the client and the server exchange greetings messages. Most servers insist on receiving valid greeting messages and some of them drop the underlying TCP connection if they do not receive valid greetings. 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, the email address of the recipient followed by the headers of the body of the email message. Once the client has sent all the email messages to the SMTP server, it terminates the SMTP association.

A successful transfer of an email message is shown below

```
S: 220 smtp.example.com ESMTP MTA information
C: HELO 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 this example, 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 *HELO* command with its fully qualified domain name. The server replies with reply-code *250* and sends its greetings. To send an email, the client must issue three commands : *RCPT TO:* that provides the address of the recipient of the email, *MAIL FROM:* that indicates the address of the sender of the email and *DATA* that starts the

actual transfer of the email message. The *MAIL FROM:* and *RCPT TO:* must be issued before the *DATA* command, but the former does not need to be sent before the former. 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 [13]. 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.

Open SMTP relays and spam

Since its creation in 1971, email was a very useful tool that was used my 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 occured in 1978 when a marketer for a computer vendor sent a *marketing email <http://www.templetons.com/brad/spamreact.html#msg>* 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, due to to low cost of sending emails, the problem of unsollicitated emails or spams has not stopped. 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 in Internet Service Providers and large companies that need to process many useless messages. SMTP servers are not anymore open **RFC 5068**. Several extensions to SMTP have been developed during the 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**.

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 a SMTP server. On such hosts, email destined to local users was delivered by placing the email in a directory owned by the user. However, the introduction of the personal computers in the 1980s, changed the 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 email client applications running 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. While POP is mainly used to download email messages, 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 and by *-ERR* to indicate errors.

[13] 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.

When a client opens a connection with the server, the latter sends as banner and 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 returns *+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 indicates that the server can delete all 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 [14]. 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.

SMTP versus POP

Both SMTP and POP are involved in the delivery of email messages. They are thus complimentary protocols. However, there are two important differences between these two protocols. First, POP forces the client to be authenticated, usually by providing a username and a password. SMTP was designed without any authentication. Second, the POP client downloads email messages from the server, while the SMTP client sends email messages.

[14] **RFC 1939** defines another authentication scheme that is not vulnerable to such attackers.

2.2.3 The HyperText Transfer Protocol

In the early days of the Internet, the network 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 client offer a user interface similar to a Unix shell and allows the client to browse the file system on the server and 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 userid 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. Anonymous access is nowadays 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 that were available then [B1989]. As none of the existing solutions met CERN's requirements, they choose 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* is the hypertext. An hypertext is a text that contains references (hyperlinks) to other text that the reader can immediately access. Compared to the hypertexts 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.

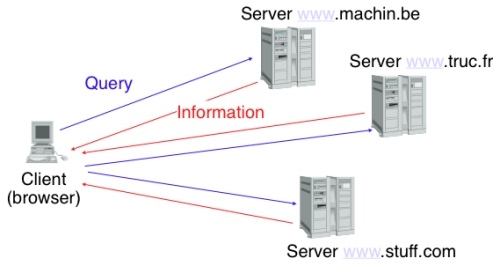


Figure 2.9: World-wide web clients and servers

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

1. A standardised addressing scheme that allows to unambiguously identify documents
2. A standard document format. html http://www.w3.org/MarkUp
3. A standardised protocol that allows to efficiently retrieve documents stored on a server

Open standards and open implementations

Open standards have and are still playing a key role in the success of the *world wide web* as we know it today. However, open and efficient implementations of these standards have greatly contributed to the success of the *web*. 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 <http://www.w3.org/Library/Activity.html>*. 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 component 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 the URIs

```
URI         = scheme ":" "//" authority path [ "?" query ] [ "#" fragment ]
scheme      = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
authority   = [ userinfo "@" ] host [ ":" port ]
query       = *( pchar / "/" / "?" )
fragment    = *( pchar / "/" / "?" )
```

The first component of a URI is its *scheme*. In practice, the *scheme* identifies the application-layer protocol that must used by the client to retrieve the document. The most frequent scheme is *http* that will be described later, but a URI scheme can be defined for almost any application layer protocol [15]. The characters *:* and *//* follow the *scheme* of any URI.

The second part of the URI is the *authority*. It includes the DNS name or the IP address on which the document can be retrieved by using the protocol specified in the *scheme*. This name can be preceded by some information about the user (e.g. a username) who is requesting the information. Earlier definitions of the URI allowed to specify a username 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 each *scheme* and the port number should only be included in the URI is a non-default port number is used.

The third part of the URI is the path to the document. This path is structured as filenames on a Unix host. If the path is not specified, the server will provide 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.BaseHTTPRequestHandl
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 i sidelined in **RFC 2368**. 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* by using *http*. The query *highlight=http* is associated to this URI. 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 username 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 username set *cnn.example.com&story=breaking_news*.

[15] The list of standard URI schemes is maintained by IANA at http://www.iana.org/assignments/uri-schemes.html

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 that remain readable and editable for tens of years and insisted on a standardised format supported by multiple vendors. Today, SGML is not widely used anymore besides specific applications, but children like *HTML* and *XML* are now widespread.

HTML is a markup language that contains several markers. Most markers are very simple HTML document such as the one shown in the figure below is delineated by the *<HTML> 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>* indicates an unnumbered list while the *<LI>* marker indicates a list item. The *<A HREF="URI">text</A>* indicates an hyperlink. The *text* will be rendered in the web page and client will fetch the URI if the user clicks on the link.

Figure 2.10: 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*. 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 indicate optional parameters for each 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 header ends with an empty line.
- a MIME document

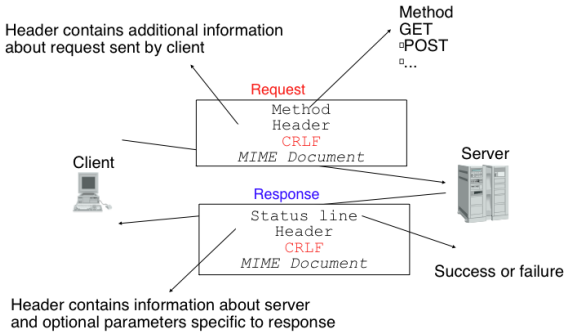


Figure 2.11: HTTP requests and responses

There are three types of methods in HTTP requests :

- the *GET* method is the most popular one. It is used to retrieve a document from a server. It should be noted that the client only provides the path of URI of the requested document after the *GET* keyword. For example, if a client requests the http://www.w3.org/MarkUp/ URI, it will open a TCP on port *80* with host *www.w3.org*. The first line of its HTTP request will contain

  GET /MarkUp/ HTTP/1.0

- the *HEAD* method is a variant of the *GET* method that allows to retrieve the header lines for a given URI without retrieving the entire document. It can be used by a client that wants to verify whether a document has changed compared to a previous version.

- the *POST* method is less popular. It can be used by a client to send a document to a server. The document sent is attached to the HTTP request.

HTTP clients and servers can include many different HTTP headers in the HTTP requests and responses. Each 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. This header would be set to *x-gzip* for a document compressed by using the gzip software.

RFC 1945 and **RFC 2616** also 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 the software release and optional modules that is uses. 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:* indicates the last modification date and time 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 the 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 inside its cache. If yes, it sends an 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 measurement the impact of advertisements containing hyperlinks placed on websites.
- the *Host:* header contains the fully qualified domain name of the URI being requested.

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 server both *web.example.com* and *www.dummy.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.dummy.net/index.html*, The HTTP request contains the following line :

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

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 digits status code and additional information in English. The 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 not available anymore on the server. *301 Moved Permanently* indicates that the requested document is not anymore 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 that indicate the type of document and its size.

As an illustration of HTTP/1.0, here are an HTTP request for http://www.ietf.org and the corresponding HTTP response. The H

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 included modules. The *Last-Modified:* header indicates that the requested document was modified about one week before the request. An 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.

```
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 PHP/5.2.6 with Suhosin-Patch mod_pytho
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 text documents that were self-contained. For this reason, and to ease the implementation of clients and servers, the designers of HTTP choose 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. On a web containing only text documents this was a reasonable design choice as the client remains usually idle while the (human) user is reading the retrieved document.

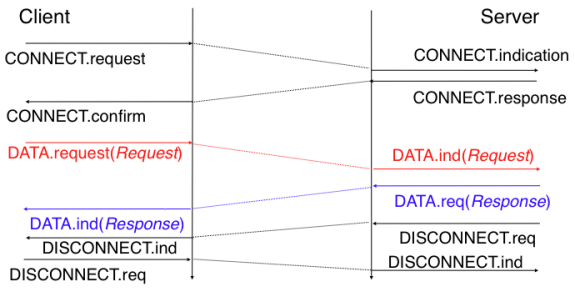


Figure 2.12: HTTP 1.0 and the underlying TCP connection

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 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 to completely retrieve 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.

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.

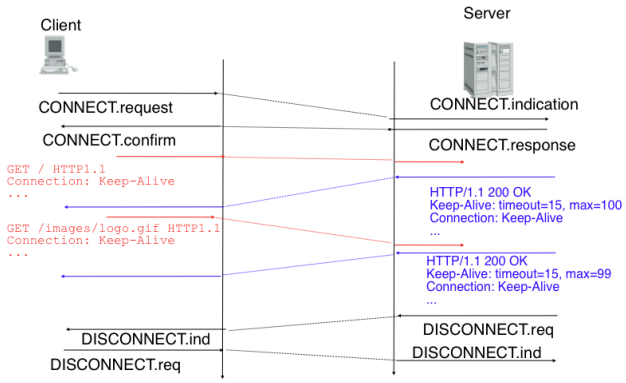


Figure 2.13: HTTP 1.1 persistent connections

- 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) AppleWebKit/531.22.7 (KHTML, li
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 the 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) AppleWebKit/531.22.7 (KHTML, li
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

...
```

The last request sent by the client is for the webserver's icon [16] 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 immediately closed.

```
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) AppleWebKit/531.22.7 (KHTML, li
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"> ...
```

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

However, in practice, some servers want to provide content that is 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. For this, servers need to maintain some information about the preferences of each user and use to information to produce the content that matches their user's preferences. Several solutions have been tested in HTTP to solve this problem and it is interesting to discuss their advantages and drawbacks.

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, usernames and password could be included inside URIs **RFC 1738**. However, placing passwords in clear in a potentially publicly visible URI is completely insecure and this usage is now deprecated **RFC 3986**. HTTP supports several extension headers **RFC 2617** that can be used by a server to request the client to be authenticated and by the client to provide his/her credentials. However, usernames and passwords have not been popular on web servers because they force the human users to remember one username and one password per server. Remembering a password is acceptable when a user needs to access protected content, but users will not accept to pick a username and password to receive targeted advertisements from the web sites that they visit.

[16] 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.

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 that language that it prefers to use by selecting options on each visited web server.

The third, and widely adopted, solution are the HTTP cookies. HTTP cookies were initially developed as a private extension by Netscape. They are now part of the standard **RFC 2965**. 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 an 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.

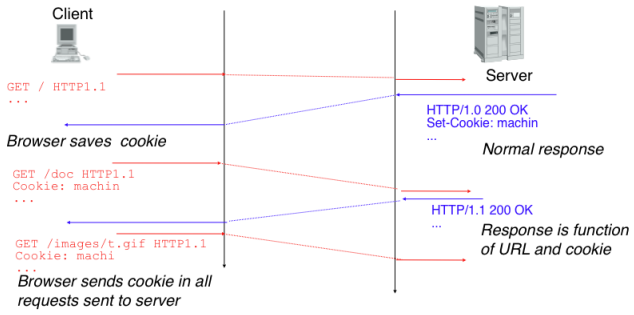


Figure 2.14: HTTP cookies

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

2.2.4 Writing simple networked applications

CHAPTER

THREE

THE TRANSPORT LAYER

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. There are two types of network layer services : connectionless and connection-oriented. The connectionless network layer service is the most widespread. Its main characteristics are :

- the connectionless network layer service can only transfer SDUs of *limited size* [1]
- 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

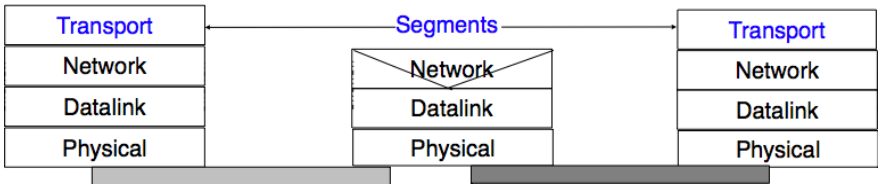


Figure 3.1: The transport layer in the reference model

These imperfections of the connectionless network layer service will be better understood once we have explained the network layer in the next chapter. At this point, let us simply assume that these imperfections occur without trying to understand why they occur.

Some transport protocols can be used on top of a connection-oriented network service, such as class 0 of the ISO Transport Protocol (TP0) defined in [X224] , but they have not been widely used. We do not discuss such utilisation of a connection-oriented network service in more details in this book.

This chapter is organised as follows. We first explain how it is possible to provide a reliable transport service on top of an unreliable connectionless network service. For this, we explain the main mechanisms found in such protocols. Then, we study in details the two transport protocols that are used in the Internet. We begin with the User Datagram Protocol (UDP) that provides a simple connectionless transport service. Then, we describe the Transmission Control Protocol (TCP) in details, including its congestion control mechanism.

3.1 Principles of a reliable transport protocol

In this section, we design a reliable transport protocol running above a connectionless network layer service. For this, we first assume that the network layer provides a perfect service, i.e. :

- the connectionless network layer service never corrupts SDUs

[1] Many network layer services are unable to carry SDUs that are larger than 64 KBytes.

- the connectionless network layer service never discards SDUs
- the connectionless network layer service never delays, reorders nor duplicate SDUs
- the connectionless network layer service can support SDUs of *any size*

We will remove these assumptions one after the other in order to better understand the mechanisms that are used to solve each imperfection.

3.1.1 Reliable data transfer on top of a perfect network service

The transport layer entity interacts with a user in the application layer and also with an entity in the network layer. According to the reference model, these interactions will be performed by using *DATA.req'and DATA.ind* primitives. However, to simplify the presentation and avoid a confusion between a *DATA.req* primitive issued by the user of the transport layer entity and a *DATA.req* issued by the transport layer entity itself, we use the following terminology :

- the interactions between the user and the transport layer entity are represented by using the classical *DATA.req*, *DATA.ind*, ... primitives
- the interactions between the transport layer entity and the network layer service are represented by using *send* instead of *DATA.req* and *recvd* instead of *DATA.ind*

This is illustrated in the figure below.

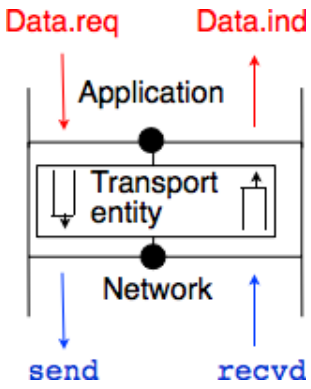


Figure 3.2: Interactions between the transport layer and its user and its network layer provider

When running on top of a perfect connectionless network service, a transport level entity can simply issue a *send(SDU)* upon arrival of *DATA.req(SDU)*. Similarly, the receiver issues a *DATA.ind(SDU)* upon reception of a *recvd(SDU)*. Such a simple protocol is sufficient when a single SDU is sent.

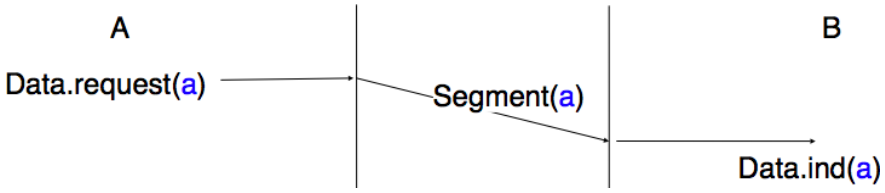


Figure 3.3: The simplest transport protocol

Unfortunately, this is not sufficient to always ensure a reliable delivery of the SDUs. Consider the case of a client that 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 could arise. The transport entity contains buffers to store that SDUs that have been received as *Data.request*

from the application and not yet sent via the network service. If the application is faster than the network layer, the buffer becomes full and the operating system suspends the application to let the transport entity empty its transmission queue. The transport entity also uses a buffer to store the segments received from the network layer that have not yet been processed by the application. If the application is slow to process the data, this buffer becomes full and the transport entity is not able to accept anymore the segments from the network layer. The buffers of the transport entity have a limited size [2] and if they overflow, the transport entity is forced to discard received segments.

To solve this problem, we need to introduce inside our transport protocol, and despite the fact that the network layer provides a perfect service, a feedback mechanism that allows the receiver to inform the sender that it has processed a segment and that another one can be sent. For this, our transport protocol must process two types of segments :

- data segments carrying a SDU
- control segments carrying an acknowledgment that indicates that the previous segment was processed correctly

These two types of segments can be distinguished by using a segment composed of two parts :

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

The transport 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.

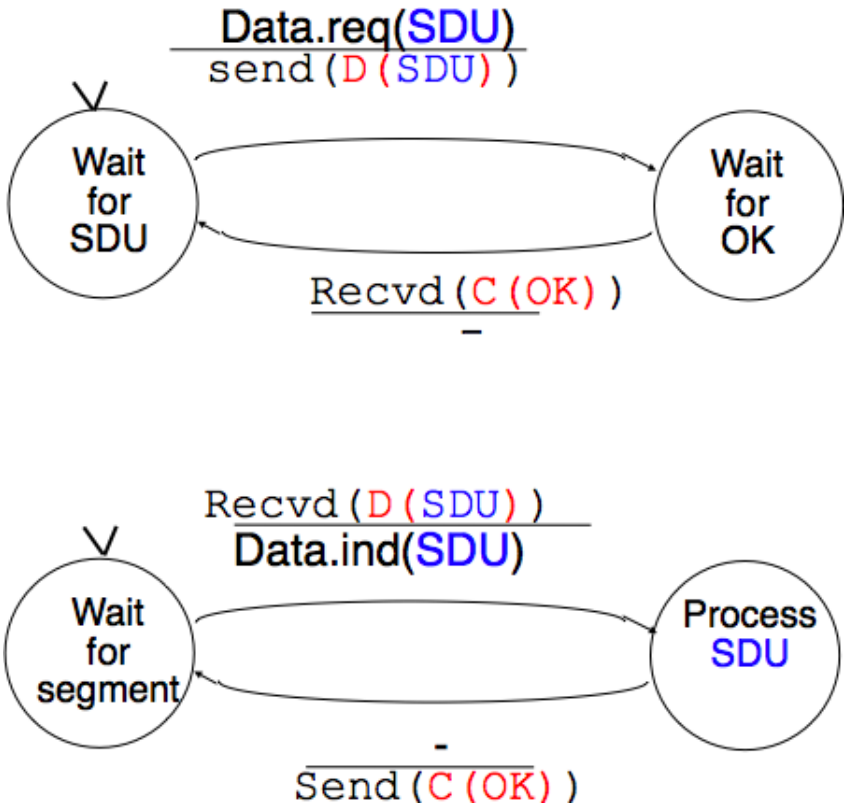


Figure 3.4: Finite state machine of the simplest transport protocol

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 segments between two hosts.

[2] In the application layer, most servers are implemented as processes. The network and transport layer on the other hand are usually implemented inside the operating system and the amount of memory that they can use is limited by the amount of memory allocated to the entire kernel.

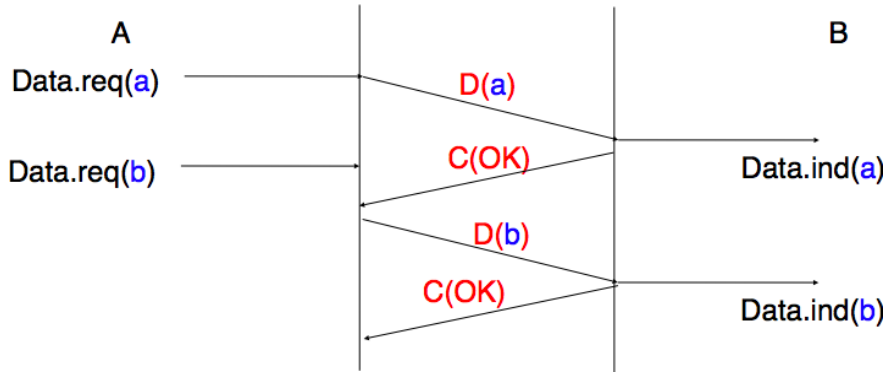


Figure 3.5: Time sequence diagram illustrating the operation simplest transport protocol

3.1.2 Reliable data transfer on top of an imperfect network service

The transport layer must deal with the imperfections of the network layer service. There are three types of imperfections that must be considered by the transport layer :

1. Segments can be corrupted by transmission errors
2. Segments can be lost
3. Segments can be reordered or duplicated

To deal with these four types of imperfections, transport protocols rely on different types of mechanisms. The first problem are the transmission errors. The segments sent by a transport entity is processed by the network and datalink layers and finally transmitted by the physical layer. All these layers are imperfect. For example, the physical layer may be affected different types of errors :

- random isolated errors where the value of single bit has been modified changed 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 segments that are sent. *Information Theory* defines two mechanisms that can be used to transmit information over a transmission channel that is affected by random errors. These two mechanisms add redundancy to the information sent 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.

Information theory defines *coding schemes*. There are different types of coding schemes, but let us focus on coding schemes that operate on binary strings. A coding scheme is a function that maps information encoded as a string of *m* bits into a string of *n* bits. The simplest coding scheme is the even parity coding. This coding scheme takes a *m* bits source string and produces a *m+1* bits coded string where the first *m* bits of the coded string are the bits of the source string and the last bit of the coded string is always chosen such that the coded string always contains an even number of bits set to *1*. For example :

- *1001* is encoded as *10010*
- *1101* is encoded as *11011*

This parity scheme has been used in some RAMs and to encode characters sent over a serial line. It is easy to show that this coding scheme allows the receiver to detect a single transmission error, but it cannot correct it. If two or more bits are in error, the receiver may not always be able to detect the error.

Some coding schemes allow the receiver to correct some transmission errors. For example, consider the coding scheme that encodes each source bit as follows :

- *1* is encoded as *111*
- *0* is encoded as *000*

This simple coding scheme forces the sender to transmit three bits for each source bit. However, it allows the receiver to correct single bit errors. More advanced coding systems that allow to recover from errors are used in several types of physical layers.

Transport protocols use error detection schemes, but none of the widely used transport protocols relies on error correction schemes. For this, a segment is usually divided in two parts :

- a *header* that contains the fields used by the transport protocol to ensure a 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 passed by the application layer.

Some segment header also include a *length* that indicates the total length of the segment 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 compose a segment. There are different types of checksums. For example, an eight bits checksum can be computed as the arithmetic sum of all the bytes of (the header and trailer of) the segment. The checksum is computed by the sender before sending the segment and the receiver verifies the checksum upon reception of each segment. The receiver discards the segments 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.

Checksums, CRCs, ...

Most of the protocols in the TCP/IP protocol suite rely on the simple Internet checksum to verify that the received segment has not been affected by transmission errors. Despite its popularity and ease of implementation, the Internet checksum is not the only available checksum mechanism. The 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 be easily implemented efficiently in hardware and have better error-detection capabilities that Internet checksum [SGP98] . However, when the first transport protocols were designed the CRCs were considered to be too CPU-intensive for software implementations and other checksum mechanisms were chosen. The TCP/IP community chose the Internet checksum, the OSI community chose the Fletcher checksum [Sklower89] . There are now efficient techniques to quickly compute CRCs in software [Feldmeier95] . The SCTP protocol initially chose the Adler-32 checksum but replaced it recently with a CRC (see **RFC 3309**).

The second imperfection of the network layer is that it may lose segments. As we will see later, the main cause of packet losses in the network layers is the lack of buffers in intermediate routers. Since the receiver sends an acknowledgement segment after having received each segment, the simplest solution to deal with losses is to use a retransmission timer. When the sender sends a segment, it starts a retransmission timer. The value of this retransmission timer should be larger than the *round-trip-time*, i.e. the delay between the transmission of a data segment and the reception of the corresponding acknowledgement. When the retransmission timer expires, the sender assumes that the data segment has been lost and retransmits it. This is illustrated in the figure below.

Unfortunately, retransmission timers alone are not sufficient to recover from segment losses. Let us for example consider 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, transport protocols associate a *sequence number* to each data segment. This *sequence number* is one of the fields found in the header of the data segments. We use the notation *D(S,...)* to indicate a data segment whose sequence number field is set to *S*. The acknowledgements also contain a sequence number that indicates the data segments that it acknowledges. We use *OKS* to indicate an acknowledgement segment that confirms the reception of *D(S,...)*. The sequence number is encoded as a bit string of fixed length. The simplest transport is the Alternating Bit Protocol (ABP). The Alternating

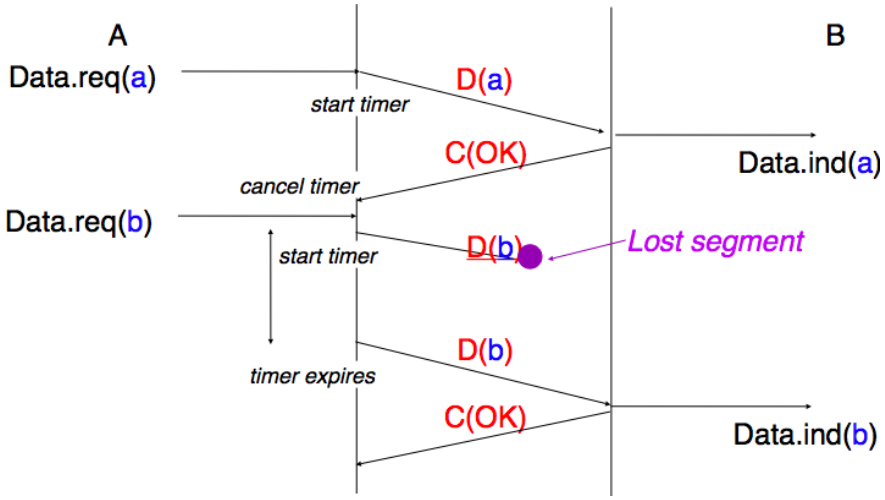


Figure 3.6: Using retransmission timers to recover from segment losses

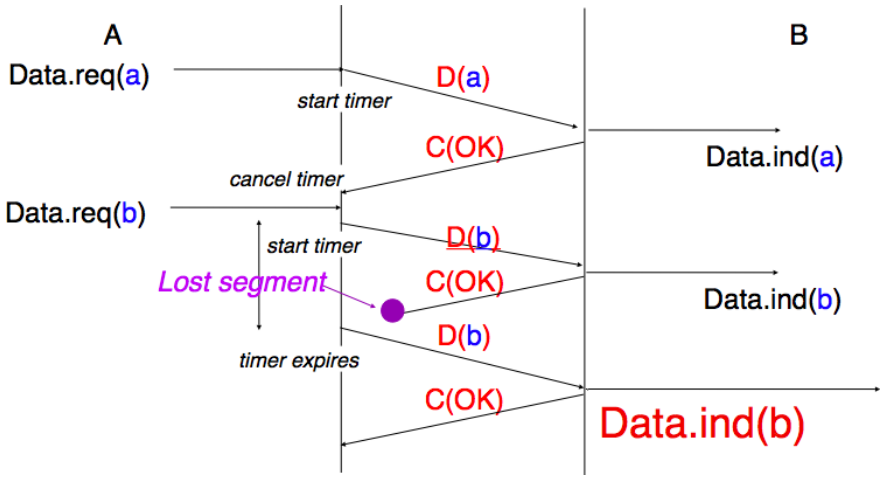


Figure 3.7: Limitations of retransmission timers

Bit Protocol uses a single bit to encode the sequence number. It can be implemented by using a simple Finite State Machine.

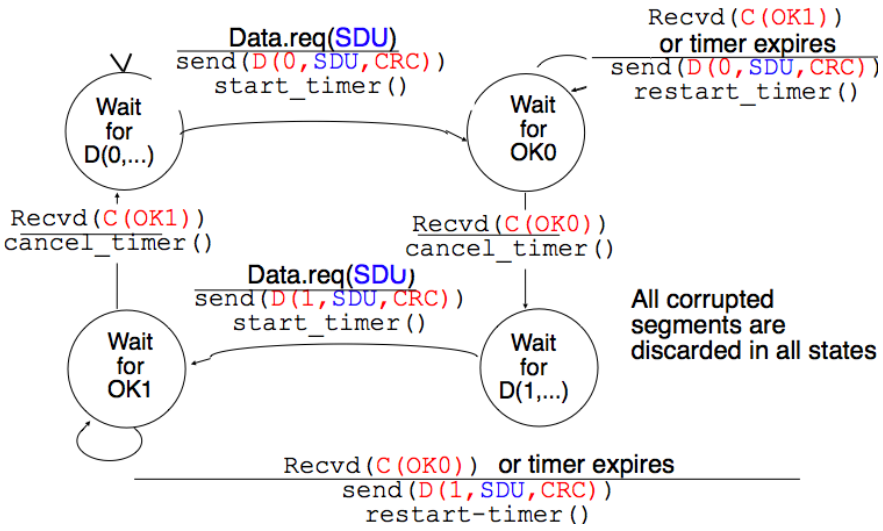


Figure 3.8: Alternating bit protocol : Sender FSM

The initial state of the sender is *Wait for D(0,...)*. In this state, the sender waits for a *Data.request*. The first data segment that it sends uses sequence number *0*. After having sent this segment, the sender waits for an *OK0* acknowledgement. A segment 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 segment has a correct *CRC*, it passes the SDU to its user and sends *OK0*. Then, the receiver waits for *D(1,...)*. In this state, it may receive a duplicate *D(0,...)* or a data segment with an invalid CRC. In both cases, it returns an *OK0* segment to allow the sender to recover from the possible loss of the previous *OK0* segment.

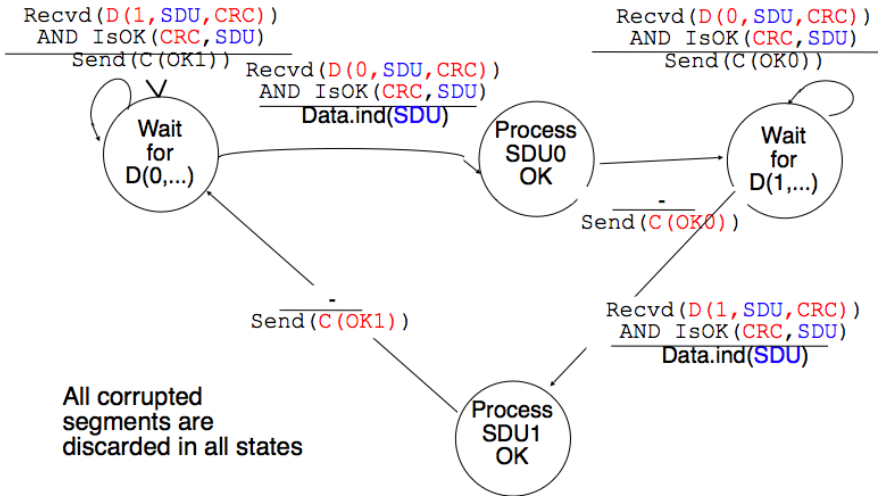


Figure 3.9: Alternating bit protocol : Receiver FSM

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

The Alternating Bit Protocol can recover from the transmission errors and the segment 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

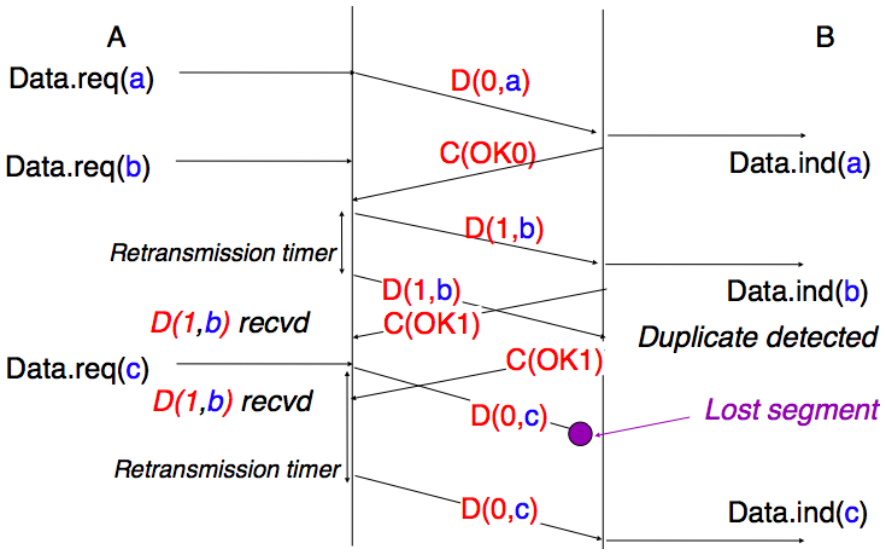


Figure 3.10: Operation of the alternating bit protocol

milliseconds propagation delay. If these hosts send 1000 bits segments, then the maximum throughput that can be achieved by the alternating bit protocol is one segment every $20 + 250 + 250 = 520$ milliseconds if we ignore the transmission time of the acknowledgement. This is less than 2 Kbits/sec !

3.1.3 Go-back-n and selective repeat

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

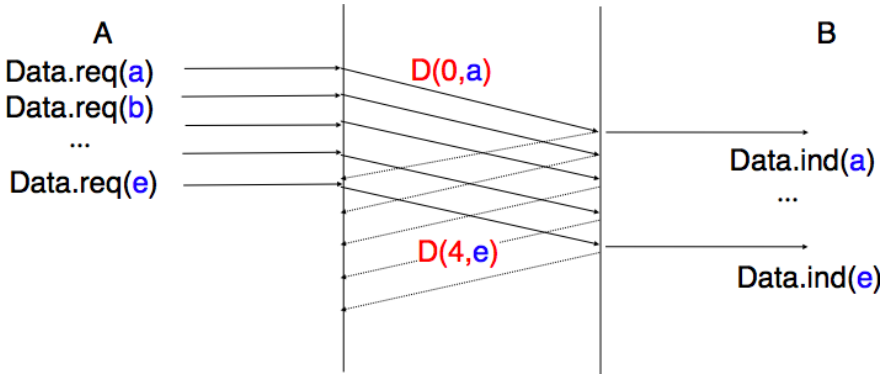


Figure 3.11: Pipelining to improve the performance of transport protocols

Pipelining allows the sender to transmit segments faster, but we need to ensure that the receiver does not become overloaded. Otherwise, the segments sent by the sender are not correctly received by the destination. The transport protocols that rely on pipelining allow the sender to transmit W unacknowledged segments 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 segments without being forced to wait for an acknowledgement. The figure below shows a sliding window that contains five segments (*6,7,8,9* and *10*). Two of these sequence numbers (*6* and *7*)

have been used to send segments 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.

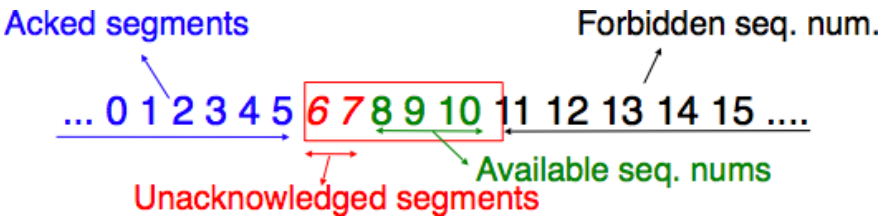


Figure 3.12: The sliding window

The figure below illustrates the operation of the sliding window. The sliding window contains three segments. The sender can thus transmit three segments before being forced to wait for an acknowledgement. The sliding window moves to the higher sequence numbers upon reception of acknowledgements. 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 SDU *d*.

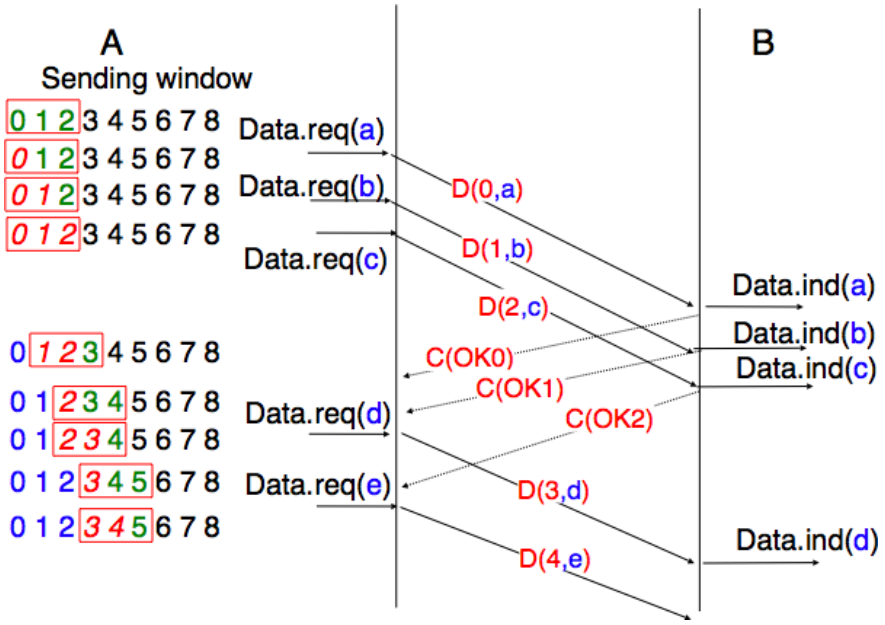


Figure 3.13: Utilisation of the sliding window

In practice, as the segment header encodes the sequence number in a *n* bits string, only the sequence numbers between 0 and $2^n - 1$ can be used. This implies that the same sequence number is used for different segments and that 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 segment header. Note that upon reception of *OK1*, the sender slides its window and can reuse sequence number *0*. Unfortunately, segment losses do not disappear because a transport protocol is using a sliding window. To recover from segment losses, a sliding window protocol must define :

- a heuristic to detect segment losses
- a *retransmission strategy* to retransmit the lost segments.

The simplest sliding window protocol uses *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 segments that arrive in-sequence. A *go-back-n* receiver discards any out-of-sequence segment that it receives. When a *go-back-n* receives a data segment, it always returns an

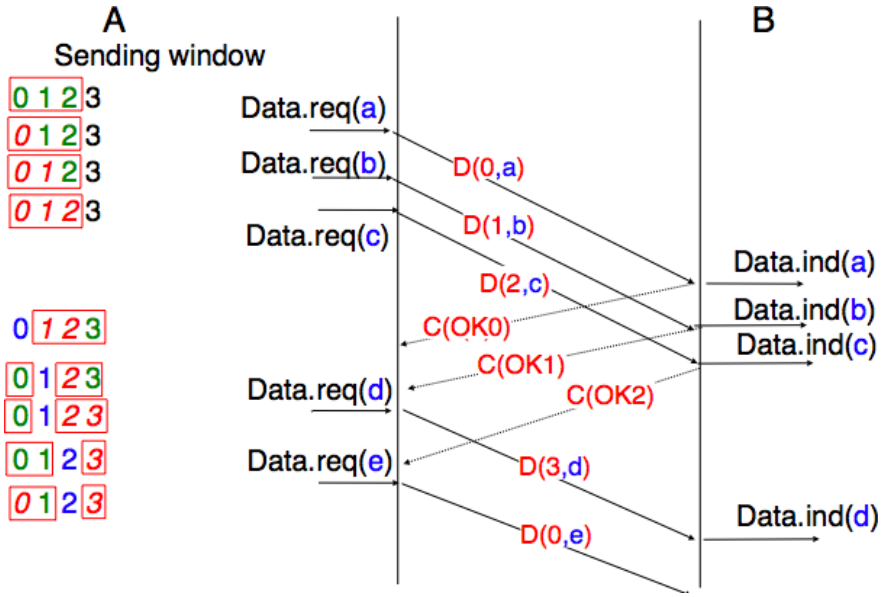


Figure 3.14: Utilisation of the sliding window with modulo arithmetic

acknowledgement that contains the sequence number of the last in-sequence segment that it received. This acknowledgement is said to be *cumulative*. When a *go-back-n* receiver send an acknowledgement for sequence number *x*, it implicitly acknowledges the reception of all segments whose sequence number is earlier than *x*. A key advantage of these cumulative acknowledgements is that it is easy to receove from the loss of an acknowledgement. Consider for example a *go-back-n* receiver that received segments *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 segments 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 segment that has been acknowledged. The receiver only accepts the segments that are received in sequence. *maxseq* is the number of different sequence numbers (2^n).

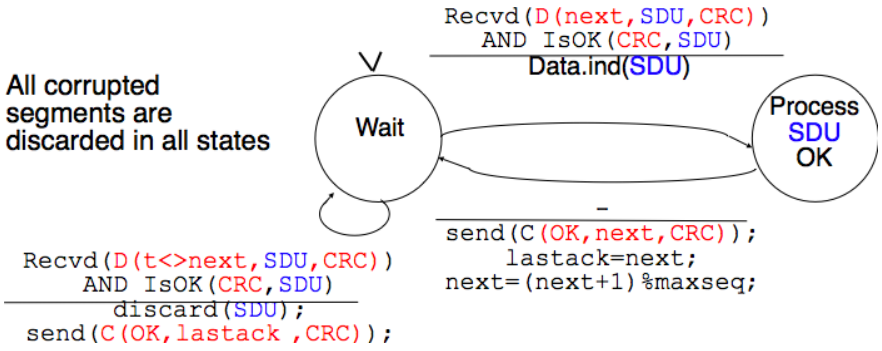


Figure 3.15: Go-back-n : receiver

A *go-back-n* sender is also very simple. It uses a sending buffer that can store an entire sliding window of segments [3] . The segments are sent with increasing sequence number (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

[3] The size of the sliding window can be either fixed for a given protocol or negotiated during the connection establishment phase. We'll see later that it is also possible to change the size of the sliding window during the connection's lifetime.

the sending buffer all the acknowledged segments. It uses a retransmission timer to detect segment losses. A simple *go-back-n* sender maintains one retransmission timer per connection. This timer is started when the first segment is sent. When the *go-back-n sender* receives an acknowledgement, it restarts the retransmission timer only if there are still unacknowledged segments. When the retransmission timer expires, the *go-back-n* sender assumes that all the unacknowledged segments that are stored in its sending buffer have been lost. It thus retransmits all the unacknowledged segments and restarts its retransmission timer.

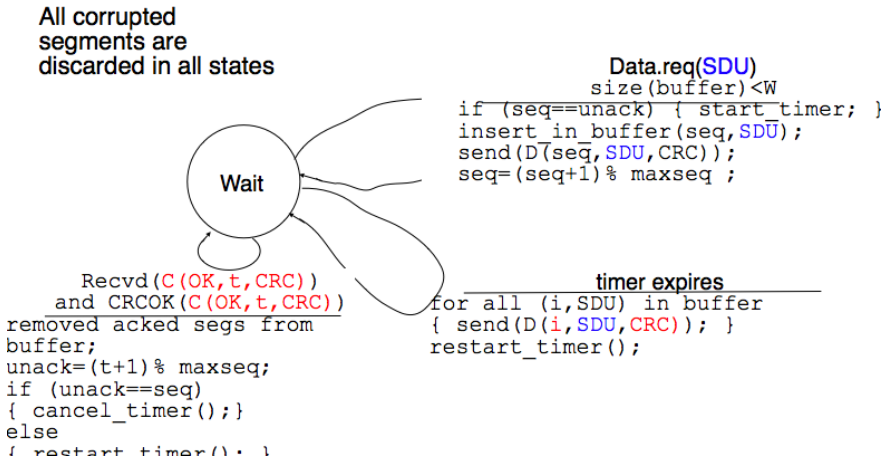


Figure 3.16: Go-back-n : sender

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

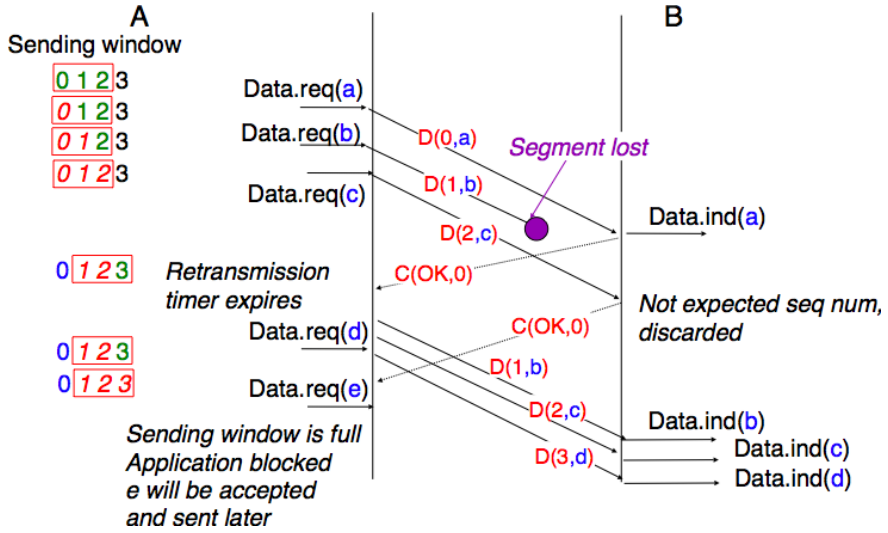


Figure 3.17: Go-back-n : example

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

- the *go-back-n* receiver does not accept out-of-sequence segments

- the *go-back-n* sender retransmits all unacknowledged segments once its has detected a loss

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

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

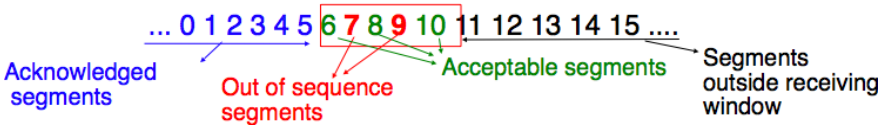


Figure 3.18: The receiving window with selective repeat

A *selective repeat* receiver discards all segments having an invalid CRC. The receiver maintains (variable *lastack*) the sequence number of the last in-sequence segment that it has received. It 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 that it has received. This can be done for example by placing the list of the sequence numbers of the correctly received out-of-sequence segments in the acknowledgements together with the *lastack* value.

When a *selective repeat* receivers receives a data segment, it first verifies whether the segment is inside its receiving window. If yes, the segment is placed in the receive buffer. Otherwise, it is discarded and an acknowledgement containing *lastack* is sent. Then the receiver removes from the receive buffer all consecutive segments starting at *lastack* (if any). The payloads of these segments are delivered to the user, *lastack* and the receiving window are updated and an acknowledgement that acknowledges the last segment received in sequence is sent.

The *selective repeat* sender maintains a sending buffer that can store up to *W* unacknowledged segments. The segments are sent as long as the sending buffer is not full. Several implementations of a *selective repeat* sender are possible. A simple implementation is to associate a retransmission timer to each segment. The timer is started when the segment is sent and cancelled upon reception of an acknowledgement that covers this segment. When a retransmission timer expires, the corresponding segment is retransmitted and this retransmission timer is restarted. When an acknowledgement is received, all the segments 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 segments are lost. In this figure, *C(OK,x)* is used to indicate that all segments, up to and including sequence number *x* have been received correctly. Pure cumulative acknowledgements work well with the *go-back-n* strategy. However, with only cumulative acknowledgements a *selective repeat* sender cannot easily determine which data segments have been correctly received after a lost data segment. 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 segment although it has already been received. A possible solution to improve the performance of *selective repeat* is to provide additional information about the received segments 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 segments that have already been received. Such acknowledgements are sometimes called *selective acknowledgements*. This is illustrated in the figure below.

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

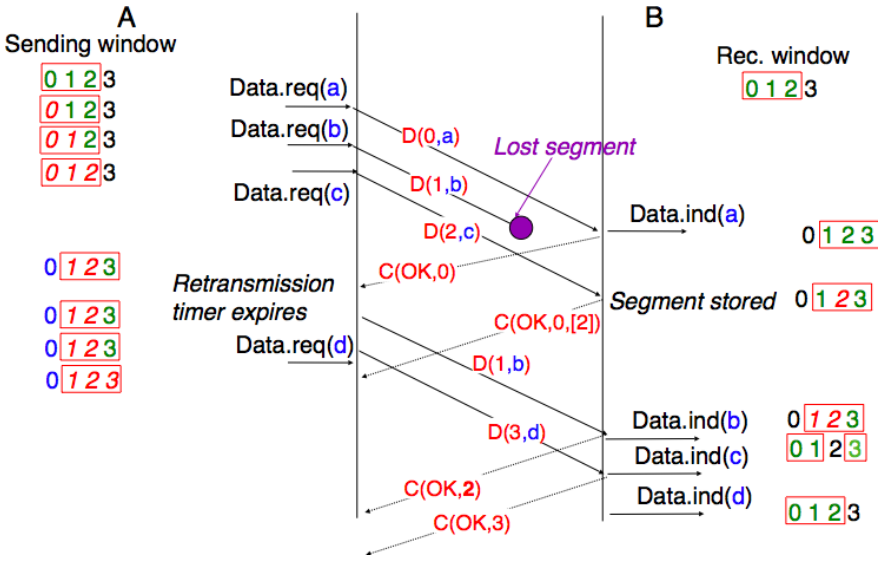


Figure 3.19: Selective repeat : example

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

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

Go-back-n or *selective repeat* are used by transport protocols to provide a reliable data transfer above an unreliable network layer service. Until now, we have assumed that the size of the sliding window was fixed for the entire lifetime of the connection. In practice a transport layer entity is usually implemented in the operating system and shares memory with other parts of the system. Furthermore, a transport layer entity must support at the same time several (possibly hundreds or thousands) of transport connections. This implies that the memory that can be used to support the sending or the receiving buffer of a transport connection may change during the lifetime of the connection [4] . 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 adjuster 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)* [5] . The utilisation of dynamic windows is illustrated in the figure below.

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 because the application is not able to process the received data quickly enough. In this case, the receive buffer may be completely full and the advertised

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

[5] 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.

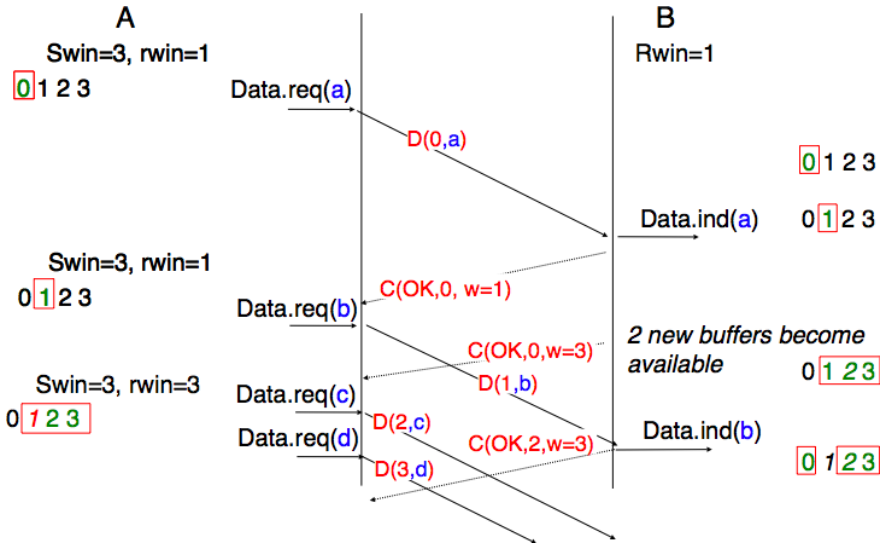


Figure 3.20: Dynamic receiving window

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

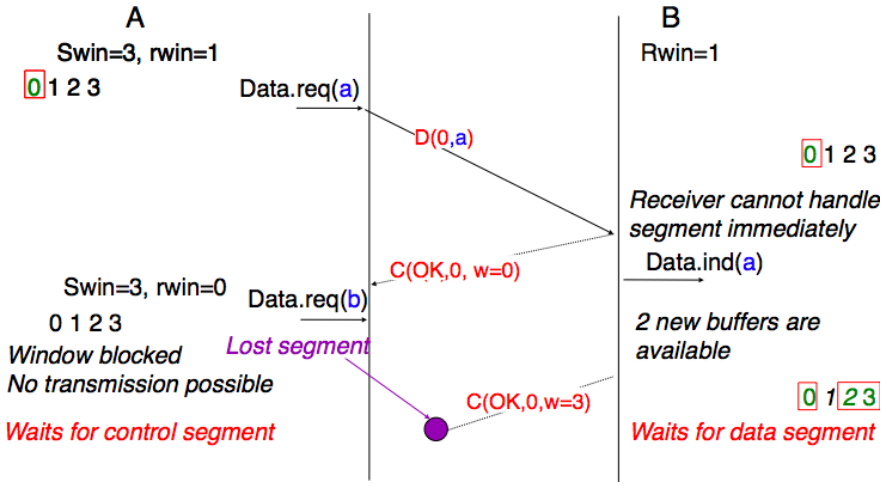


Figure 3.21: Risk of deadlock with dynamic windows

persistence timer. This timer is started by the sender when it receives an acknowledgement that advertises a *0* window. When the timer expires, the sender retransmits an old segment to force the receiver to send a new acknowledgement.

To conclude our description of the basic mechanisms found in transport protocols, we still need to discuss the impact of segments reordering. 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 too long 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

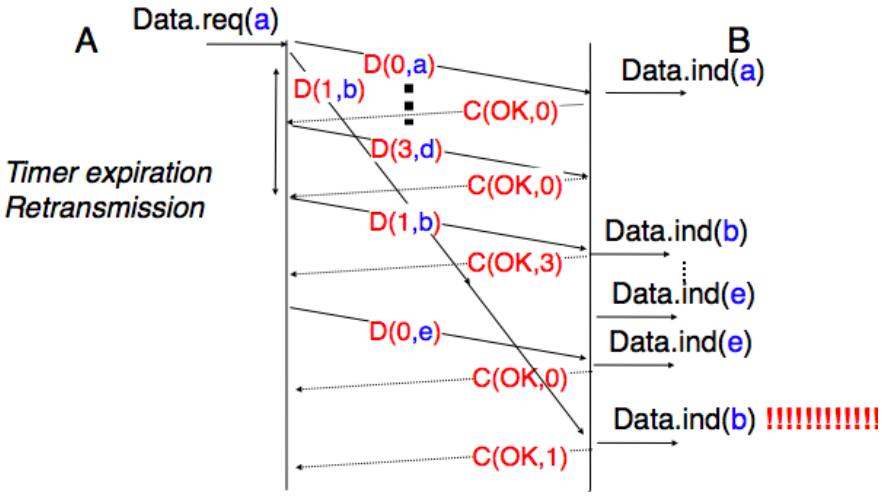


Figure 3.22: Ambiguities caused by excessive delays

layer must ensure that no packet remains in the network during more than MSL seconds. In the Internet the MSL is assumed [6] 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. Transport protocols often need to send data in both directions. To reduce the overhead caused by the acknowledgements, most transport protocols use *piggybacking*. Thanks to this technique, a transport entity can place inside the header of the data segments that it sends the acknowledgements and the receive window that it advertises for the opposite direction of the data flow. The main advantage of piggybacking is that it reduces the overhead as it is not necessary to send a complete segment to carry an acknowledgement. This is illustrated in the figure below where the acknowledgement number is underlined in the data segments. 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. The last point to be discussed about the data transfer mechanisms used by transport protocols is the

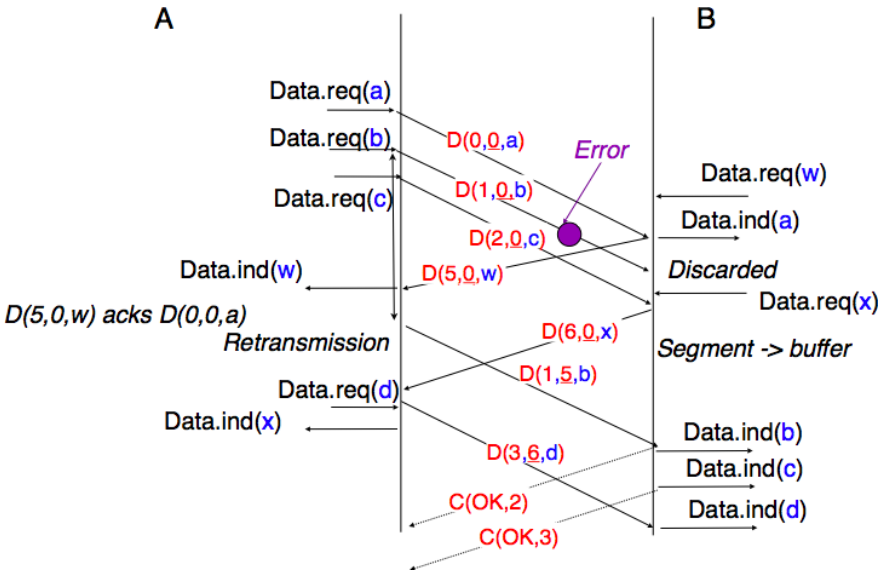


Figure 3.23: Piggybacking

[6] As we will see in the next chapter, 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.

provision of a byte stream service. As indicated in the first chapter, the byte stream service is widely used in the transport layer. The transport protocols that provide a byte stream service associate a sequence number to all the bytes that are sent and place the sequence number of the first byte of the segment in the segment's header. This is illustrated in the figure below. In this example, the sender choose to put two bytes in each of the first three segments. This is due to graphical reasons, a real transport protocol would use larger segments in practice. However, the division of the byte stream in segments combined with the losses and retransmissions explain why the byte stream service does not preserve the SDU boundaries.

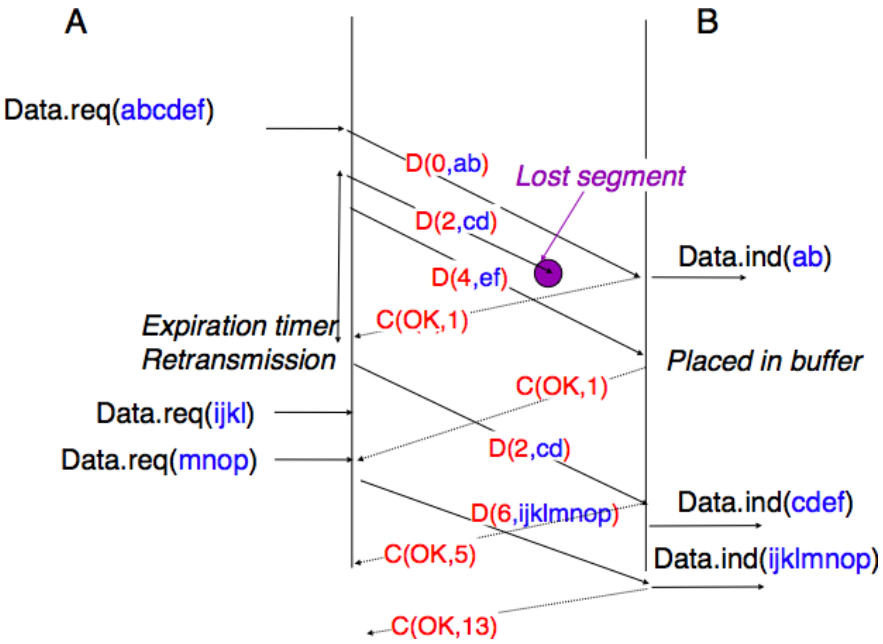


Figure 3.24: Provision of the byte stream service

3.1.4 Connection establishment and release

The last points to be discussed about the transport protocol are the mechanisms used to establish and release a transport connection.

We explained in the first chapters the service primitives that are used to establish a connection. 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 transport connection is considered to be established once the *CA* segment has been received and data segments can be sent in both directions.

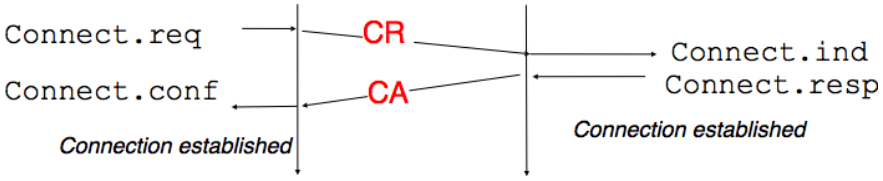


Figure 3.25: Naive transport connection establishment

Unfortunately, this scheme is not sufficient for several reasons. First, a transport entity usually needs to maintain several transport connections with remote entities. Sometimes, different users (i.e. processes) running above a given

transport entity request the establishment of several transport connections to different users attached to the same remote transport entity. These different transport connections must be clearly separated to ensure that data from one connection is not passed to the other connections. This can be achieved by using a connection identifier that is chosen by the transport entities and placed inside each segment to allow the entity that receives a segment to easily associate it to one established connection.

Second, as the network layer is imperfect, the *CR* or *CA* segment 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. Furthemore, 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 (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 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

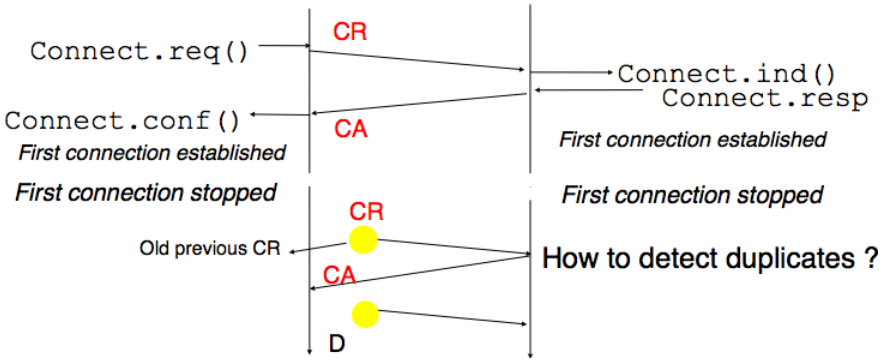


Figure 3.26: Duplicate transport connections ?

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. On today's Internet, *MSL* is expected to be 2 minutes. To avoid duplicate transport connections, transport protocols 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.
- the *transport clock* must continue to be incremented even if the transport entity stops or reboots

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

The *transport clock* is combined with an exchange of three segments that is 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 connection identifier (not shown in the figure) and a sequence

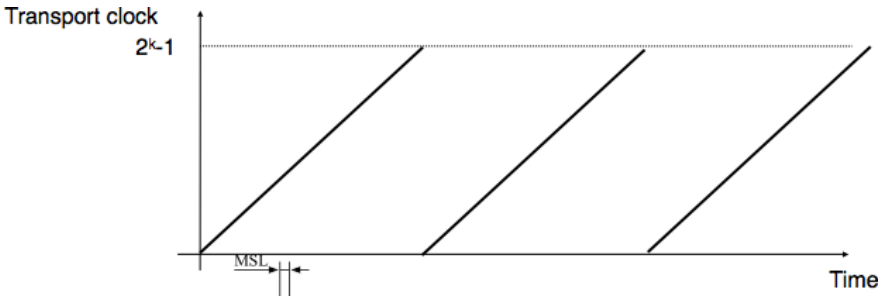


Figure 3.27: Transport clock

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 *CA* 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.

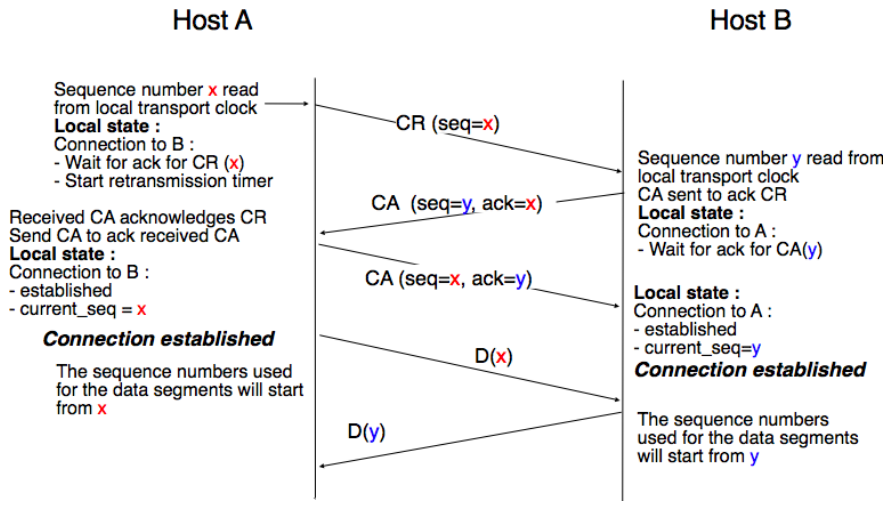


Figure 3.28: Three-way handshake

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

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.

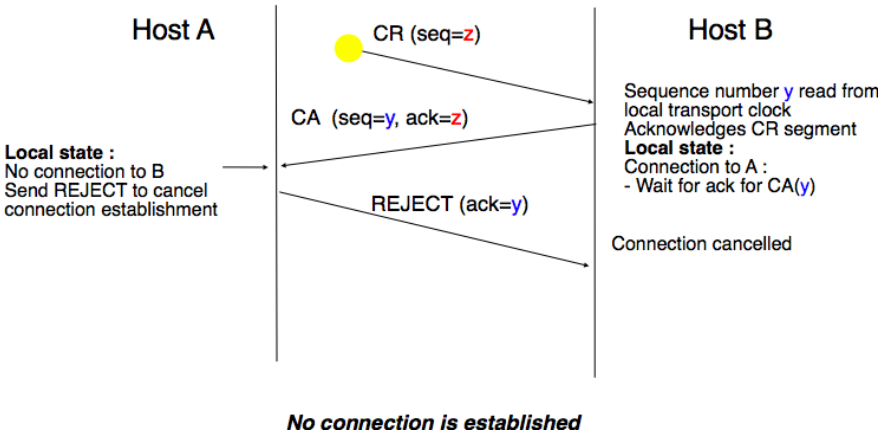


Figure 3.29: Three-way handshake : recovery from a duplicate *CR*

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.

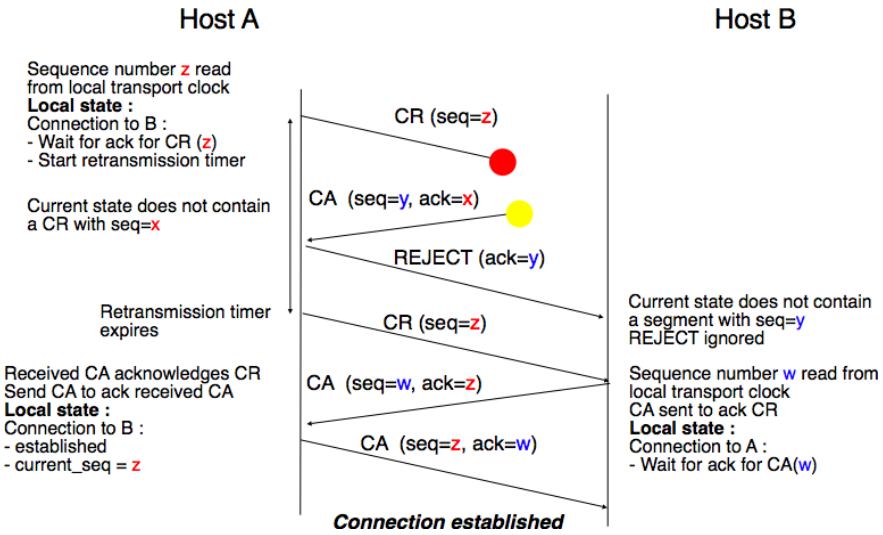


Figure 3.30: Three-way handshake : recovery from a duplicate *CA*

The last scenario is less likely, but it it important to consider it as well. The remote entity receives and 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. When we discussed the connection-oriented service, we mentionned that there are two types of

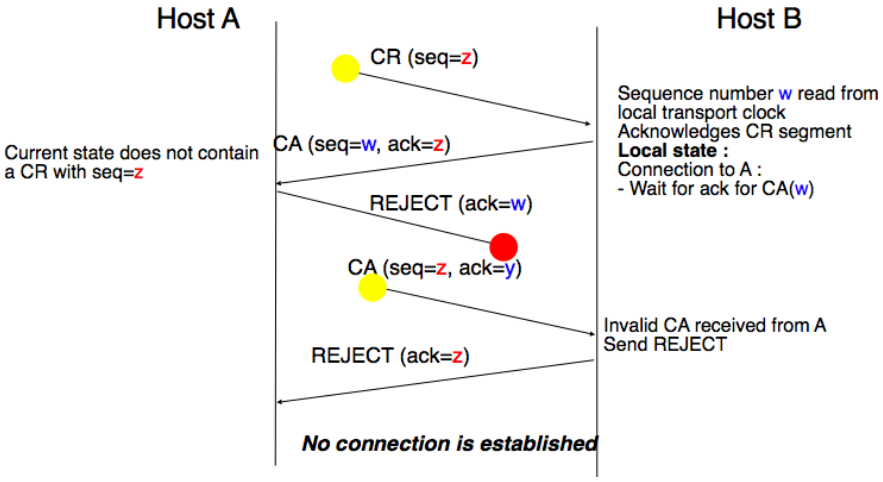


Figure 3.31: Three-way handshake : recovery from duplicates *CR* and *CA*

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.

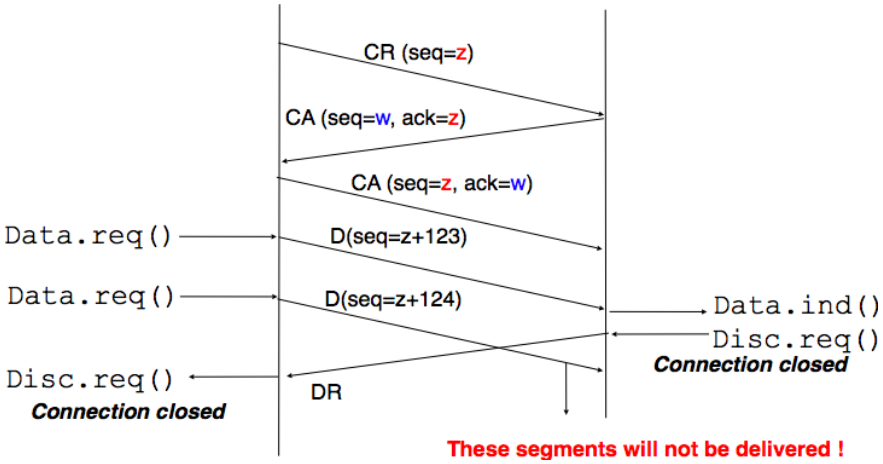


Figure 3.32: Abrupt connection release

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.

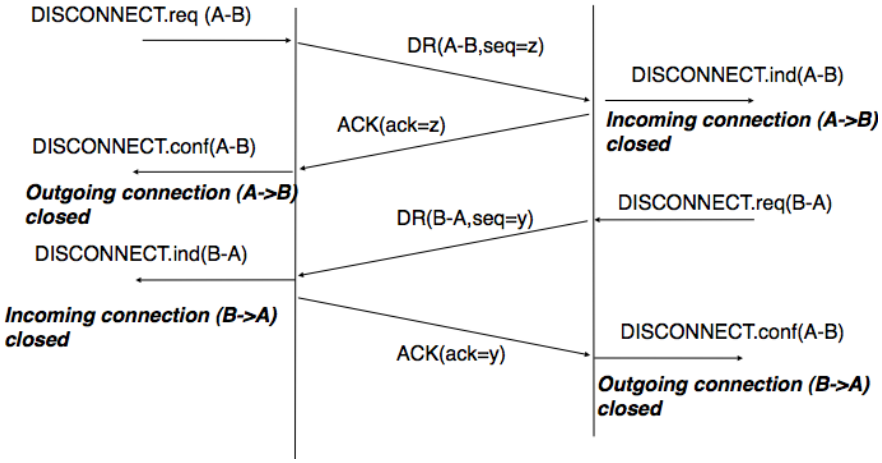


Figure 3.33: Graceful connection release

3.2 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 65507 bytes [7]
- 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 besides the IP address that identifies a host to differentiate the application running on a host. This additional addressing is provided by the *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. UDP uses a single segment format shown below. The UDP header contains four fields :

- a 16 bits source port
- a 16 bits destination port
- a 16 bits length field
- a 16 bits checksum

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

[7] This limitation is due to the fact that the network layer (IPv4 and IPv6) 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.

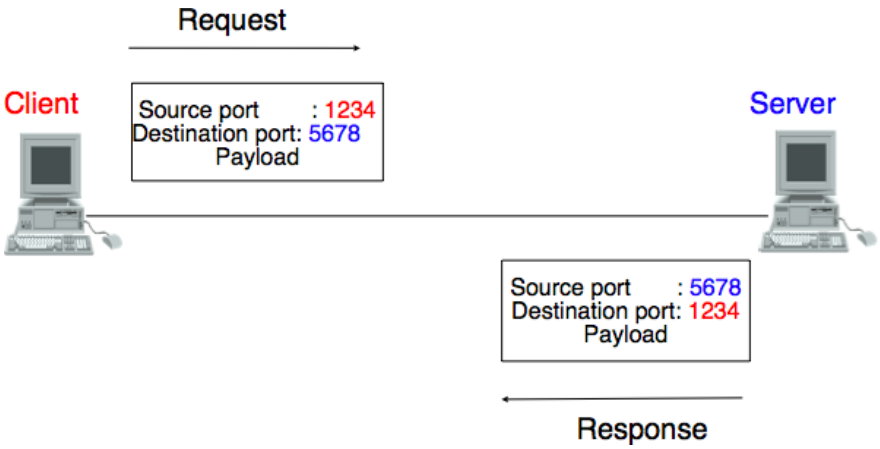


Figure 3.34: Usage of the UDP port numbers

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Length                 |         Checksum              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

UDP Header Format
```

As the port numbers are encoded as a 16 bits field, there can be only up to 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 to notice that most implementations divide the range of allowed UDP port numbers in three different ranges :

- the privileged port numbers (1 < port < 1024)
- the ephemeral port numbers (officially [8] 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 registered port numbers range should be used by servers. In theory, developers of network servers should register their port number officially through IANA, but few developers do this.

[8] 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

Computation of the UDP checksum

The checksum of the UDP segment is computed over :

- a pseudo header containing the source IP address, the destination IP address and a 32 bits bit field containing the most significant byte set to 0, the second set to 17 and the length of the UDP segment in the lower two bytes
- the entire UDP segment, including its header

This pseudo-header allows the receiver to detect errors that affect the IP source or destination addresses that are 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, but this caused problems that were difficult to diagnose. In practice, there are rarely good reasons to disable UDP checksums. 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 quickly a 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 exchange frequently 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 the 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.

3.3 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 [9]. However, the basics of the protocol remain and an implementation that only supports **RFC 793** should interoperate with today's implementation.

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 :

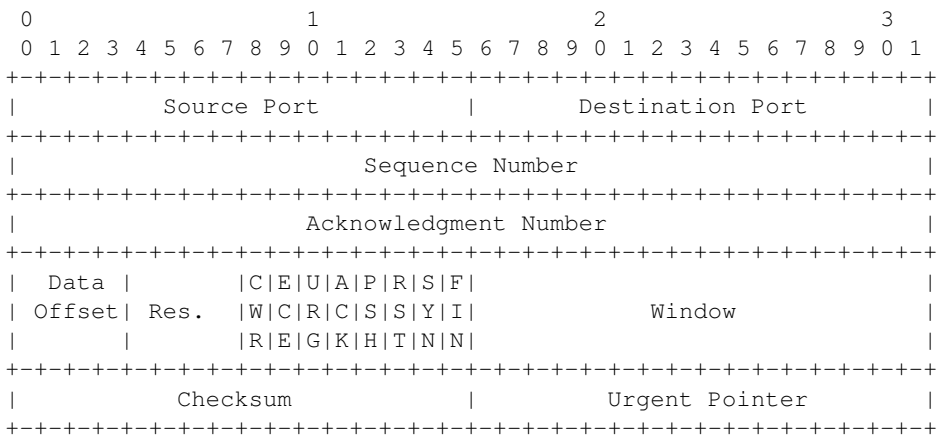
- Email (*SMTP*, *POP*, *IMAP*)
- World wide web (*HTTP*, ...)
- Most file transfer protocols (*ftp*, peer-to-peer file sharing applications , ...)
- remote computer access : *telnet*, *ssh*, :term:'X11, VNC, ...
- non-interactive multimedia applications : flash

On the global Internet, most of the applications used in the wide area rely on TCP. Many studies [10] have reported that TCP was responsible for more than 90% of the data exchange'd in the global Internet. To provide this service, TCP

[9] A detailed presentation of all standardisation documents concerning TCP may be found in **RFC 4614**

[10] 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

relies on a simple segment format. 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.



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 to identify 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 informations :
  - the IP address of the client
  - the IP 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 by 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 details 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 [StevensTCP]
- the flags field contain 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 bits words. The maximum size of the TCP header is thus 64 bytes.
- the *Optional header extension* is used to add optional information in the TCP header. Thanks to this header extension, it is possible to add new fields in 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*.

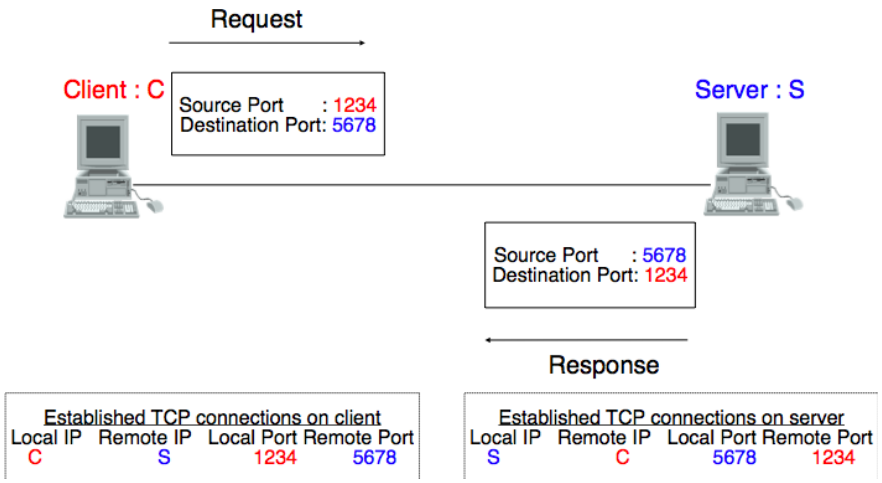


Figure 3.35: Utilization of the TCP source and destination ports

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.

3.3.1 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 used on both directions of the connection. For this, each TCP entity maintains a 32 bits counter that is supposed to be incremented by one at least every 4 microseconds and after each connection establishment [11]. 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

[11] 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**).

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 client 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.

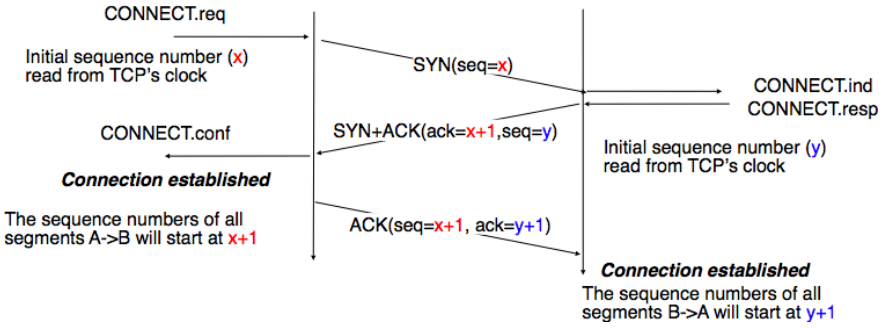


Figure 3.36: Establishment of a TCP connection

In the figure above, the connection is considered 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*).

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 [a]. 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 is chosen by the server, he can send a fake *SYN* segment and shortly after the fake *ACK* segment that confirms 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 on 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* 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 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*).

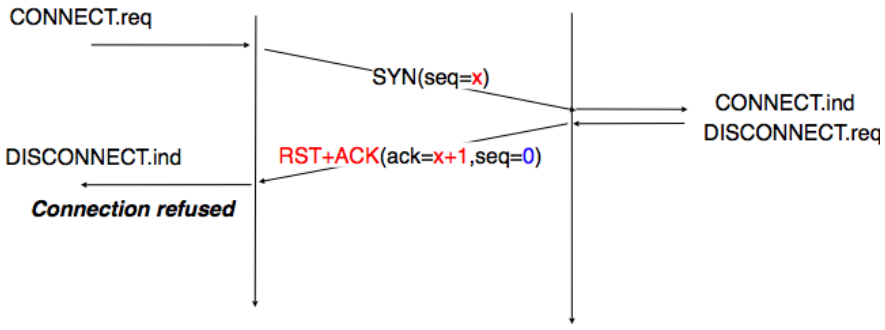


Figure 3.37: TCP connection establishment rejected by peer

The TCP connection establishment can be described as the four states 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.

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.

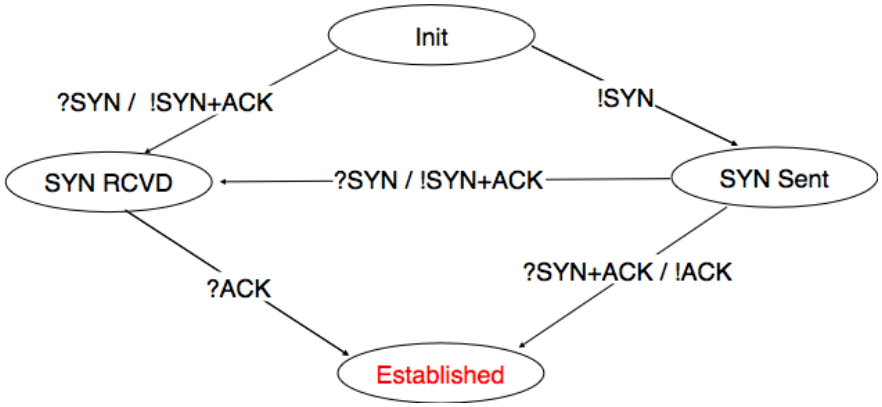


Figure 3.38: TCP FSM for connection establishment

Besides 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 [12]. 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.

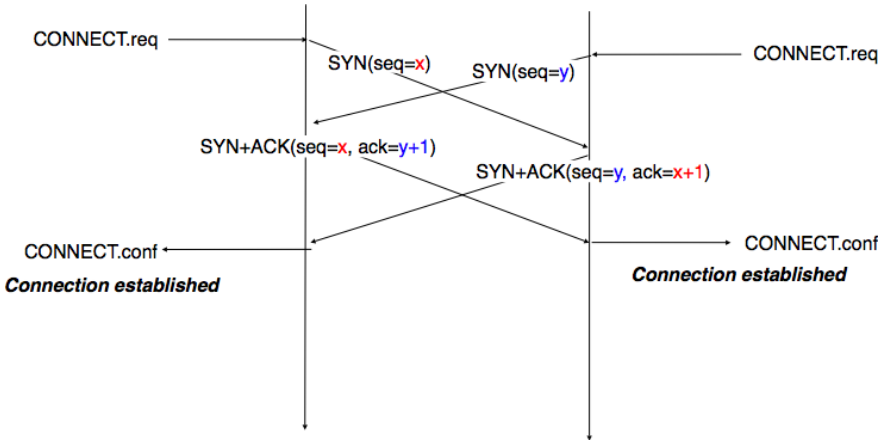


Figure 3.39: Simultaneous establishment of a TCP connection

[12] 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 all the 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 discard 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 the 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 100s of *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 firsts to be affected by this attack.

To avoid the *SYN flood* attacks, recent TCP implementations do not anymore 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 acknowedges 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 a 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*.

[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 then send the first *SYN* segment. This timer is often set to a 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 that 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 the TCP options during connection establishment is to enable TCP extensions. For example, consider **RFC 1323** (that 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 to extend 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 bits boundaries. All other options [13] are encoded by using the TLV format.

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.

3.3.2 TCP connection release

TCP, like most connection-oriented transport protocols, supports two types of connection release :

- graceful connection release where each TCP user can release its own direction of data transfer
- abrupt connection release where either one user closes both directions of data transfert 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**

[13] The full list of all TCP options may be found at http://www.iana.org/assignments/tcp-parameters/

- 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* was 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.

TCP *RST* wars

TCP implementers should ensure that two TCP entities never enter in 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 that is used when a TCP connection is released.

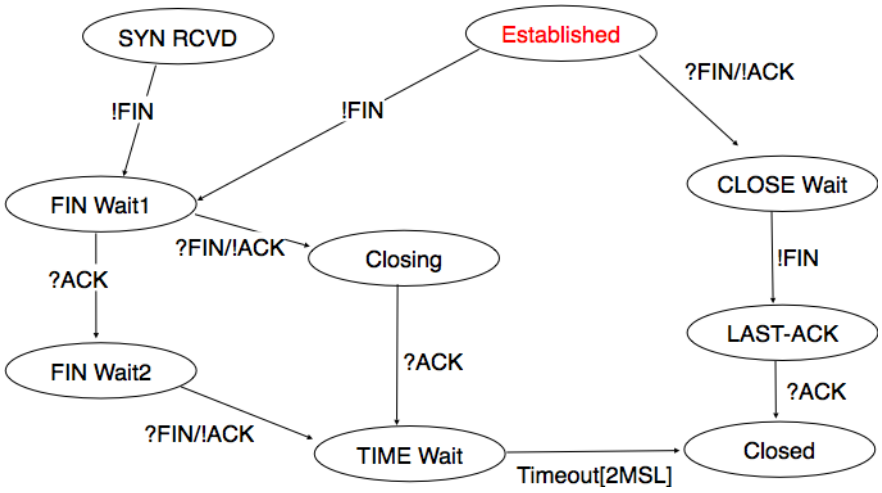


Figure 3.40: FSM for TCP connection release

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 data have 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 decides first to send a *FIN* segment. In this case, it 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, 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 during $2 * MSL$ 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 a *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.

TIME_WAIT on busy TCP servers

The $2 * MSL$ seconds delay in the *TIME_WAIT* state is an important operationnal 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 remains in the *TIME_WAIT* state during 4 minutes, this implies that the server would have to maintain more than 2 millions 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 that does not exist anymore. This optimisation reduces the number of TCBs maintained by the host sending the *RST* segment but at the cost of possibly more processing on the remote host when the *RST* segment is lost.

3.3.3 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-directionnal 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 [14] TCP segments have their *ACK* flag set.

[14] 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.

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 [a] :

- 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 use 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

[a] 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.

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 remove 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 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 woulld place each user byte in a segment containing 20 bytes of TCP header [15]. 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/IPv4 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/IPv4 networks, most of the bytes are exchanged in long packets, usually packets containing up to 1460 bytes of payload which is the default MSS for hosts attached to an Ethernet network, the most popular type of LAN

The figure below provides a distribution of the packet sizes measured on a link. It shows a three-modal distribution of the packet size. 50% of the packets contain pure TCP acknowledgements and occupy 40 bytes. About 20% of the packets contain about 500 bytes [16] of user data and 12% of the packets contain 1460 bytes of user data. However, most of the user data is transported in large packets. This packet size distribution has implications on the design of routers as we discuss in the next chapter.

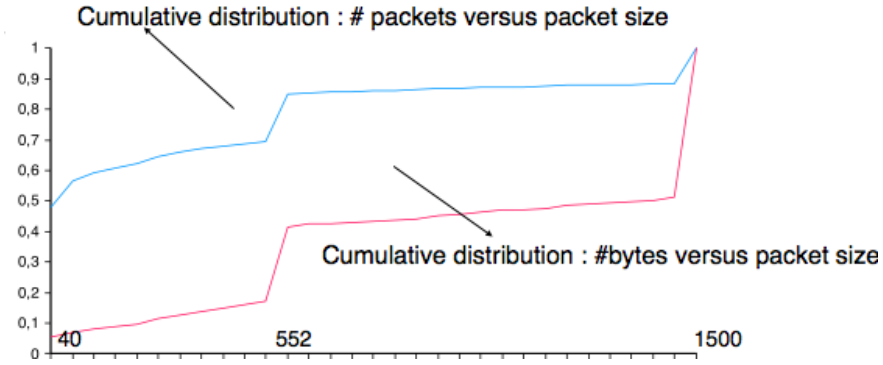


Figure 3.41: Packet size distribution in the Internet

[15] This TCP segment is then placed in an IP header. We describe IPv4 and IPv6 in the next chapter. The minimum size of the IPv4 (resp. IPv6) header is 20 bytes (resp. 40 bytes).

[16] When these measurements were taken, some hosts had a default MSS of 552 bytes (e.g. BSD Unix derivatives) or 536 bytes (the default MSS specified in **RFC 793**). Today, most TCP implementation derive the MSS from the maximum packet size of the LAN interface they use (Ethernet in most cases).

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 (40-64 bytes depending on the utilisation of TCP options) and large 1460-bytes packets carrying most of the user data.

TCP windows

From a performance viewpoint, 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 [17] 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 ($0 \leq S \leq 14$) 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 [18]. 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]

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 bandwith is wasted by retransmitting segments that have been already correctly received. 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 on each TCP connection. The round-trip-time differs between TCP connections, but may also change during the lifetime of a

[17] 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.

[18] See http://fasterdata.es.net/tuning.html for more information on how to tune a TCP implementation

single connection. For example, the figure below shows the evolution of the round-trip-time between two hosts during a period of 45 seconds.

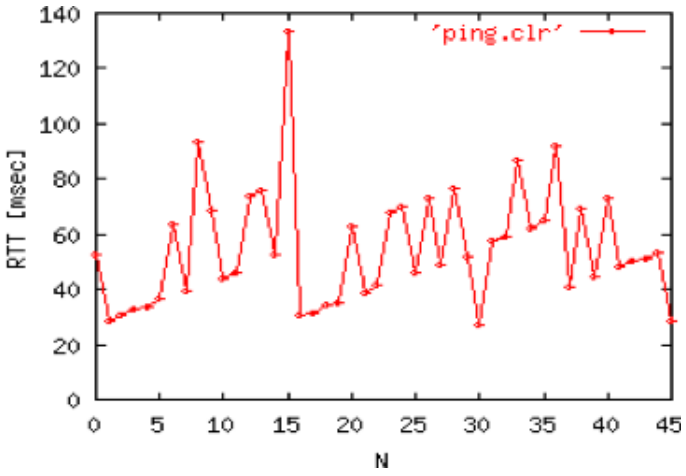


Figure 3.42: Evolution of the round-trip-time between two hosts

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 [19]. As illustrated in the figure below, this measurement works well when there are no segment losses.

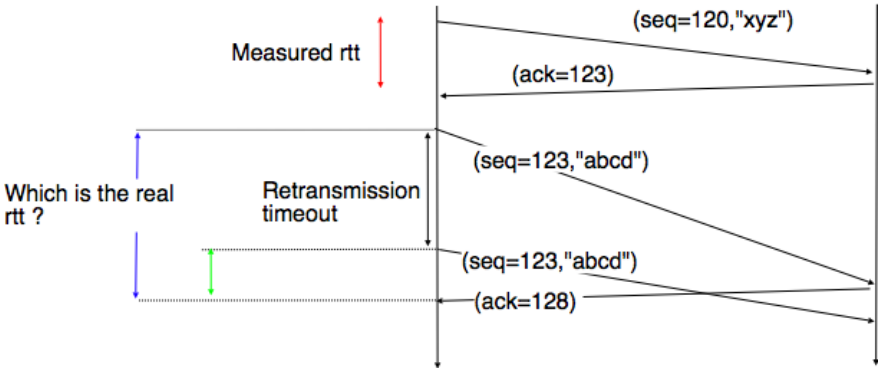


Figure 3.43: How to measure the round-trip-time ?

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 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 bits 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 [20]. The second value, TS Echo Re-

[19] 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**

[20] 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 uptime. Solutions proposed to solve this problem may be found in [CNPI09]

ply (*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 the disembiguate the round-trip-time measurement when there are retransmissions.

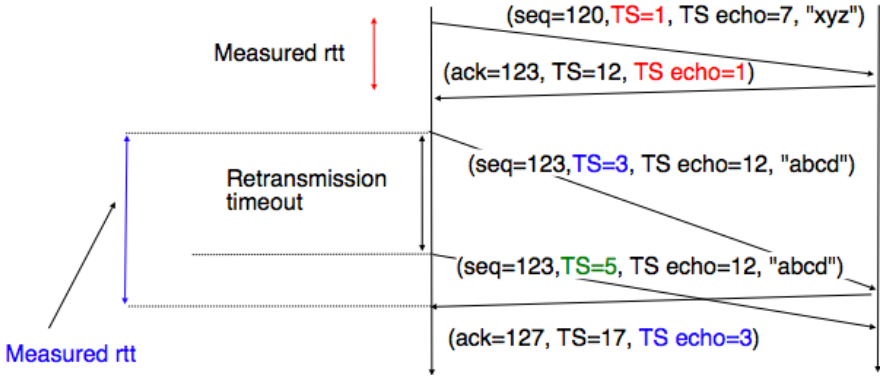


Figure 3.44: Disembiguating round-trip-time measurements with the **RFC 1323** timestamp option

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 [21] , 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 rount-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.

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

```
srtt=rtt
rttvar=rtt/2
rto=srtt+4*rttvar
```

Then, when other rtt measurements are collected, *srtt* and *rttvar* are updated as follows :

$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt|$$

$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt$$

$$rto = srtt + 4 \times rttvar$$

[21] 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.

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.

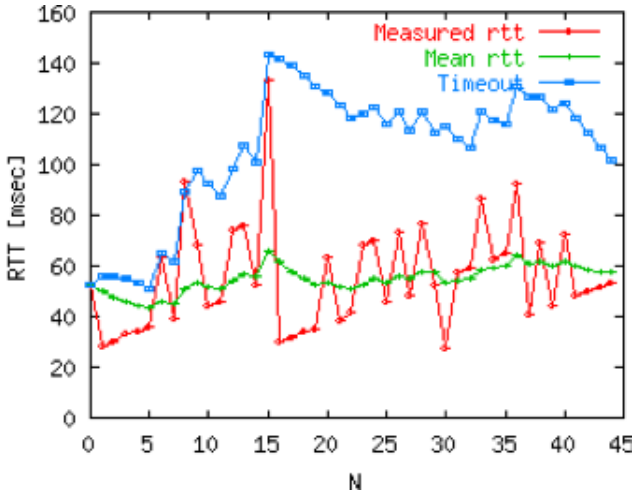


Figure 3.45: Example computation of the *rto*

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 timout. This is called an *exponential backoff*. This doubling of the retransmission timeout after a retransmission was include in TCP to deal with issues including 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 when possible and 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 received TCP segment.

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 developpers 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 heuristics. This extension can be implemented on TCP senders and thus 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 viewpoint, one issue with the 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.

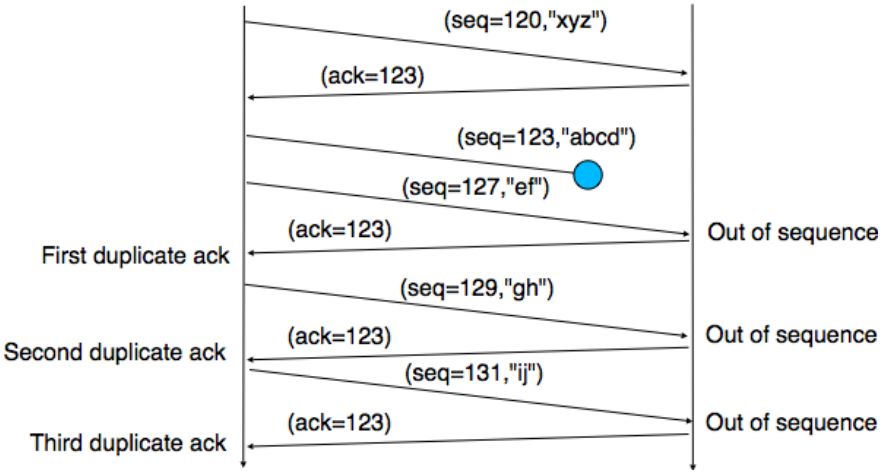


Figure 3.46: Detecting isolated segment losses

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* heuristics has been included in most TCP implementations. It can be implemented as follows

```
ack arrival:
    if tcp.ack==snd.una:    # duplicate acknowledgement
       dupacks++
       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* heuristics 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. When losses are not isolated or when the

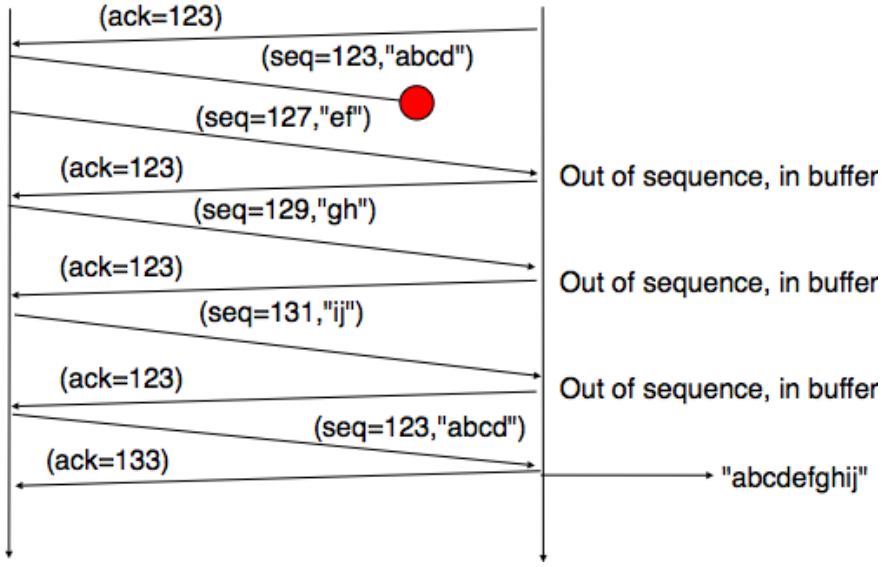


Figure 3.47: TCP fast retransmit heuristics

windows are small, the performance of the *fast retransmit* heuristics 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.

A 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 bits numbers (the same size as the TCP sequence number) in a 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

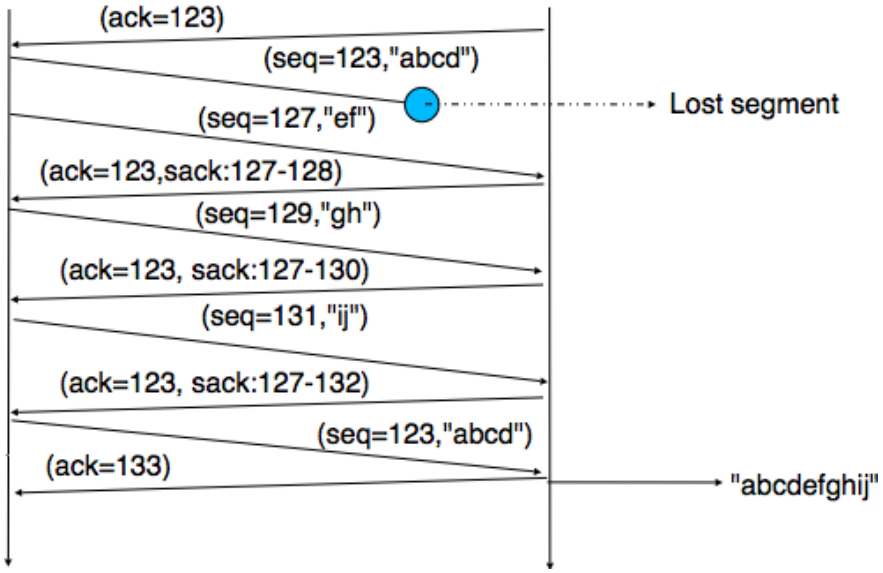


Figure 3.48: TCP selective acknowledgements

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 that has currently more than 3 blocks inside its receiving buffer must select the blocks that it places in the SACK option. A good heuristic is to put in the SACK option the blocks that have changed the most recently as the sender is likely already aware of the older blocks.

When a sender receives a SACK option that indicates a new block and thus a new possible segment loss, it usually does not retransmit the missing segment(s) 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.

3.3.4 TCP congestion control

In the previous sections, we have explained the mechanisms that TCP uses to deal with transmission errors and segment losses. In an heterogeneous network such as the Internet or enterprise IP networks, endsystems have very different performances. Some endsystems are highend servers attached to 10 Gbps links while others are mobile devices attached to a very low bandwidth wireless link. Despite of this huge difference in terms of performance, the mobile device should be able to efficiently exchange segments with the highend server. To better understand this problem, let us consider the scenario shown in the figure below where a server (*A*) attached to a *10 Mbps* link is sending TCP segments to a laptop (*C*) attached to a *2 Mbps* link.

In this network, the TCP segments sent by the server reach router *R1*. *R1* forward the segments towards router *R2*. Router *R2* can potentially receive segments at *10 Mbps*, but it can only forward them at *2 Mbps* to host *C*. Router *R2* contains buffers that allow it to store the packets that cannot be immediately forwarded to their destination. To understand the operation of TCP 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 *R2*. This queue is emptied at a rate of *2*

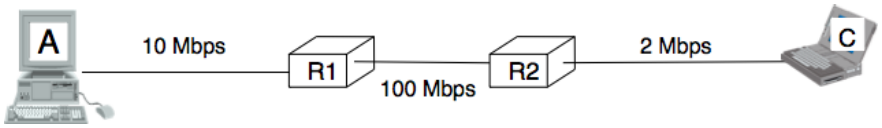


Figure 3.49: TCP over heterogenous links

Mbps.

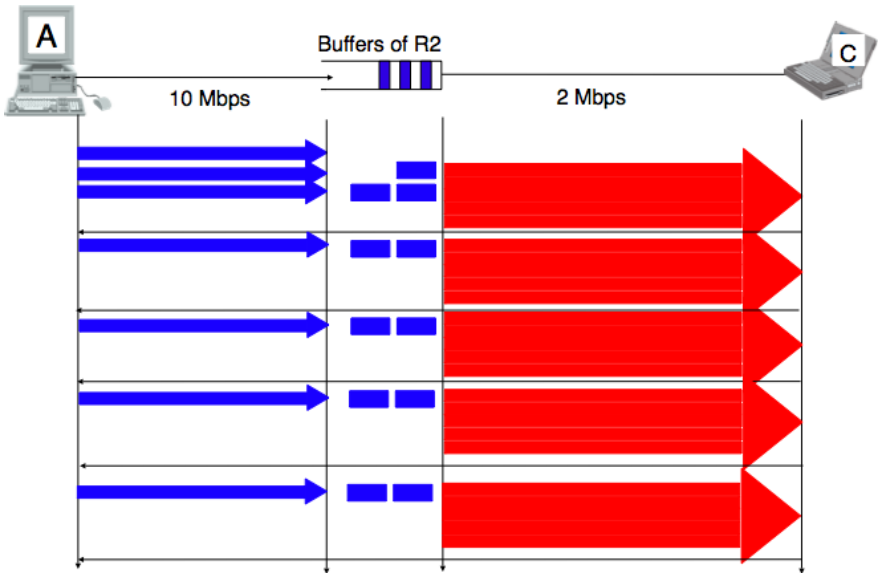


Figure 3.50: TCP self clocking

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, TCP sends one data segment after the reception of each acknowledgement returned by the destination [22] . 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 *TCP self-clocking* is the first mechanism that allows TCP 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, TCP is not always used in this environement. In the global Internet, TCP is used in networks where a large number of hosts send segments to a large number of receivers. For example, let us consider the network depicted below that 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.

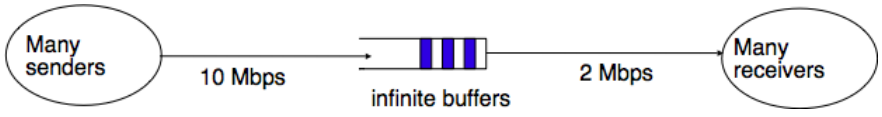


Figure 3.51: The congestion collapse problem

[22] If the destination is using delayed acknowledgements, the sending host sends two data segments after each acknowedgement.

If many TCP 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 TCP, 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 round-trip-time measured by TCP is roughly 6 milliseconds if we ignore the propagation delay on the links. Most routers manage their buffers as a FIFO queue [23]. If the buffer contains 100 segments, the round-trip-time becomes $1+100\times5+5$ milliseconds as a new segment are only transmitted on the *2 Mbps* link once all previous segments have been transmitted. Unfortunately, TCP uses a retransmission timer and performs *go-back-n* to recover from tranmission errors. If the buffer occupancy is high, TCP assumes that some segments have been lost and retransmits 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 occured several times in the late 1980s. For example, [Jacobson1988] notes that in 1986, the useable bandwidth of a 32 Kbits link dropped to 40 bits per second due to congestion collapse [24] !

The *congestion collapse* is a problem that faces all heterogenous networks. 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 details, 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.

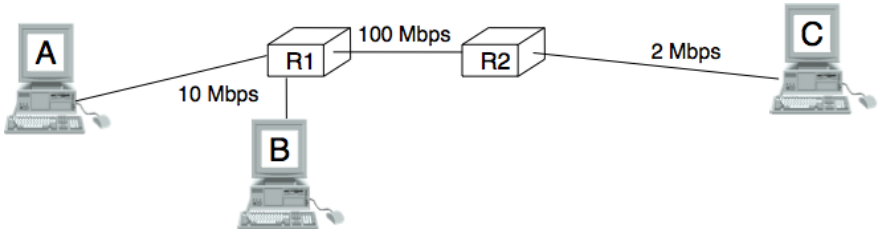


Figure 3.52: The congestion problem

To avoid *congestion collapse*, the hosts must regulate their transmission rate [25] 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 Transfert Mode (ATM)* or *Frame Relay* include congestion control mechanisms in lower layers. 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 $\forall t \sum r_i(t) \approx R$.

[23] We discuss in another chapter other possible organisations of the router's buffers.

[24] At this time, TCP implementations were mainly following **RFC 791**. The round-trip-time estimations and the retransmission mechanisms were very simple. TCP was improved after the publication of [Jacobson1988]

[25] 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.

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 blottlneck 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 in two regions. The lower left part of the graph contains all allocations (r_B, r_A) that are 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.

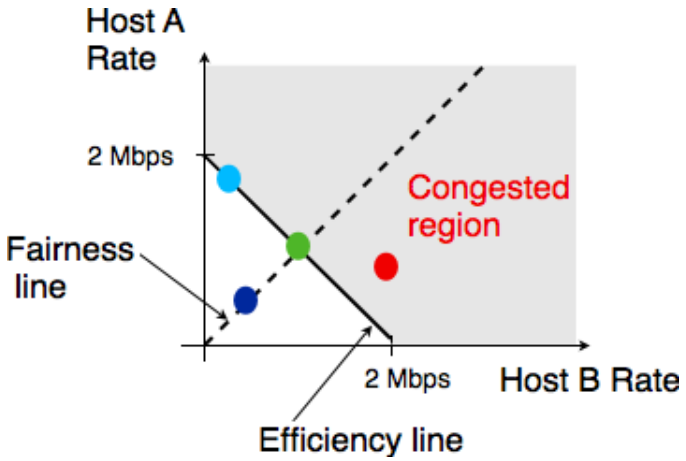


Figure 3.53: Possible allocated transmission rates

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, might be several thousands of TCP connections or more that pass through the same link [26].

To deal with these fluctuations in the demand that result in fluctuations in the available bandwidth, computer networks use a congestion control scheme. This congestion control scheme should achieve the three objectives 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 today the most widely used. Intuitively, the congestion control scheme should decrease the transmission rate of a host when congestion has been detected in the network to

[26] 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/

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.

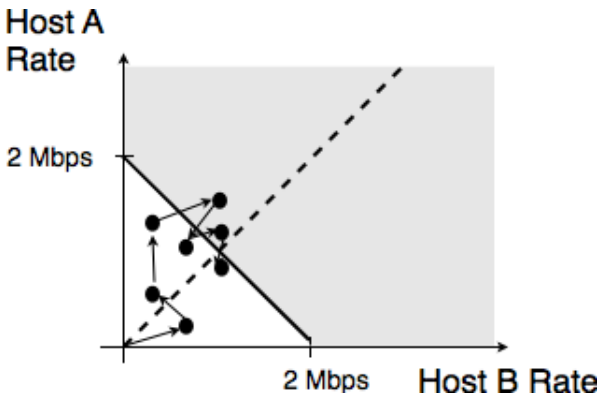


Figure 3.54: Evolution of the transmission rates

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 slowy 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 adapation algorithm can be implemented by the pseudocode below

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

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. A 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.

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 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 a 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* 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 : *sshtresh* (*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.

In practice, a TCP implementation considers the network to be congested once its needs to retransmit a segment. 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 theshold 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 at the sender performs again slow-start until the congestion window reaches the *sshtresh*. Then, it switches to congestion avoidance and the congestion window increases linearily until segments are lost and the retransmission timer expires ...

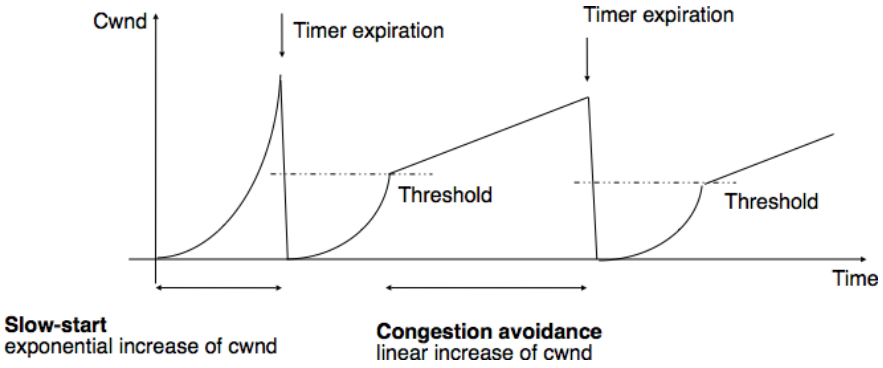


Figure 3.55: Evaluation of the TCP congestion window with severe congestion

The figure below illustrates the evolution of the congestion window when the network is lightly congested and all lost segments can be retransmitted by 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 senders immediately enters congestion avoidance as this was a mild congestion.

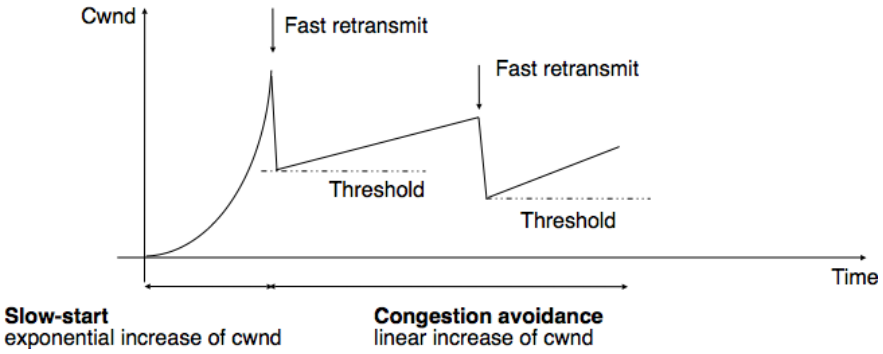


Figure 3.56: Evaluation of the TCP congestion window when the network is lightly congested

Most TCP implementations update the congestion window when they receive an acknowledgement. If we assume that the receiver acknowledges each received segment and the the sender only sends MSS sized segments, the TCP congestion control scheme can be implemented by using the simplified pseudocode [27] below

```
# Initialisation
cwnd = MSS;
ssthresh= swin;
```

[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**.

```
# Ack arrival
if tcp.ack > snd.una :  # new ack, no congestion
   if  cwnd < ssthresh :
     # slow-start : increase quickly cwnd
     # double cwnd  every rtt
     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 does not know anymore the current congestion state of th e network.

Initial congestion window

The original TCP congestion control mechanism proposed in [Jacobson1988] recommended that each TCP connection begins 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. Since the publication of **RFC 3390**, TCP hosts are allowed to use an initial congestion window of about 4 KBytes, which corresponds to 3 segments in many environments.

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 to predict 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 realistics, but it gives us a good intuition without requiring complex mathematics.

This model considers an 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.

As the losses are equally spaced, the congestion window always starts at some value ($\frac{W}{2}$), be 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.

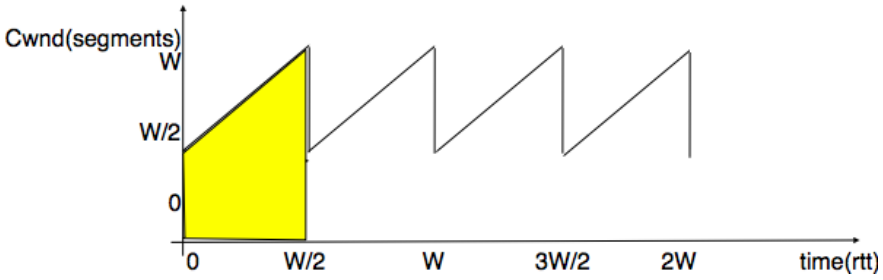


Figure 3.57: Evolution of the congestion window with regular losses

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 :

$$area = (\frac{W}{2})^2 + \frac{1}{2} \times (\frac{W}{2})^2 = \frac{3 \times W^2}{8}$$

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 that 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 that is 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.

$$Throughput < min(\frac{window}{rtt}, \frac{k \times MSS}{rtt \times \sqrt{p}})$$

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**) that was proposed in as an improvement over the fast recovery mechanism in the *Reno* implementation
- *TCP Vegas* that uses changes in the round-trip-time to estimate congestion in order to avoid it [BOP1994]
- *CUBIC* that was designed for high bandwidth links and is the default congestion control scheme in the Linux 2.6.19 kernel [HRX2008]
- *Compound TCP* that was designed for high bandwidth links is the default congestion control scheme in several Microsoft operating systems [STBT2009]

A search of the scientific literature 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...

CHAPTER

FOUR

THE NETWORK LAYER

The network layer is a very important layer in computer networks as it is the glue that allows the applications running above the transport layers to use a wide range of different and interconnected networks built with different datalink and physical layers. The network layer enables applications to run above networks built with very different network technologies.

In this chapter, we first explain the principles of the network layer. These principles include the datagram and virtual circuit modes, the separation between the data plane and the control plane and the algorithms used by routing protocols. Then, we explain in more details the network layer in the Internet, starting with IPv4 and IPv6 and then moving to the routing protocols (RIP, OSPF and BGP).

4.1 Principles

The main objective of the network layer is to allow endsystems connected to different networks to exchange information through intermediate systems that are called *router‘s. The unit of information in the network layer is called a :term:‘packet*.

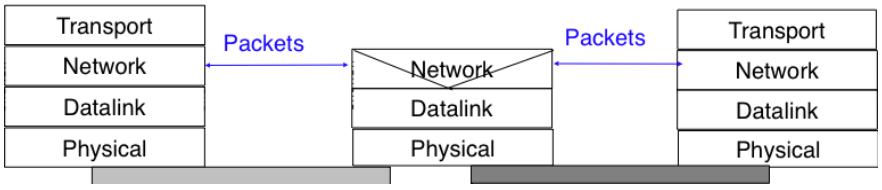


Figure 4.1: The network layer in the reference model

Before explaining the network layer in details, it is useful to first analyse the service provided by the *datalink* layer. There are many variants of the datalink layer. Some provide a connection-oriented service while others provide a connectionless service. In this section, we focus on connectionless datalink layer services that 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.

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** in 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

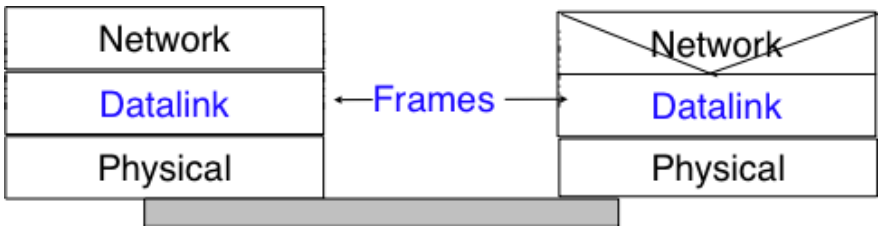


Figure 4.2: The point-to-point datalink layer

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 may connect hundreds or even thousands of devices.

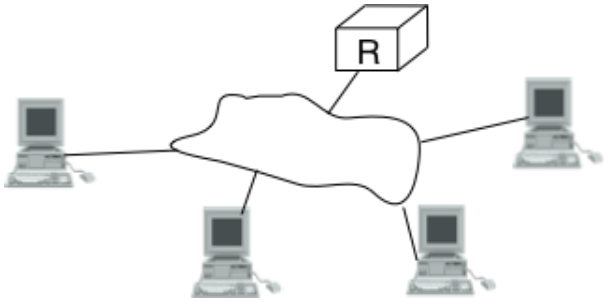


Figure 4.3: A local area network

We describe in the next chapter 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. 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 traditionnal LAN, the NBMA service only supports unicast. The datalink layer service provided by an NBMA network does not support 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 heterogeinity of the datalink layer.

The network layer itself relies on the following principles :

1. Each network layer entity is identified by a *network layer address*. This address is independent of the datalink layer addresses that it may use.
2. The service provided by the network layer does not depend on the service or the internal organisation of the underlying datalink layers.

3. The network layer is conceptually divided in two planes : the *data plane* and the *control plane*. The *data plane* contains the protocols and mechanisms that allow hosts and routers to exchange packets carrying user data. The *control plane* contains the protocols and mechanisms that enable routers to efficiently learn how to forward packets towards their final destination.

The independance of the network layer from the underlying datalink layer is a key principle of the network layer. It ensures that the network layer can be used to allow hosts attached to different types of datalink layers to exchange packets through intermediate routers. Furthermore, this allows the datalink layers and the network layer to evolve independently from each other. This enables the network layer to be easily adapted to a new datalink layer every time a new datalink layer is invented.

There are two types of services that can be provided by the network layer :

- an *unreliable connectionless* service
- a *connection-oriented*, reliable or unreliable, service

Connection-oriented services have been popular with technologies such as *X.25* and *ATM* or *frame-relay*, but nowadays most networks use an *unreliable connectionless* service. This is our main focus in this chapter.

4.1.1 Organisation of the network layer

There are two possible internal organisations of the network layer :

- datagram
- virtual circuits

The internal organisation of the network is orthogonal to the service that it provides, but most of the time a datagram organisation is used to provide a connectionless service while a virtual circuit organisation is used in networks that provide a connection-oriented service.

Datagram organisation

The first and most popular organisation of the network layer is the datagram organisation. This organisation is inspired from 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

The network layer limits the maximum packet size. Thus, the information must have been divided in packets by the transport layer before being passed to the network layer.

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 lookups the destination address of the packet in its *routing table*. A *routing table* is a data structure that maps each destination address (or set of destination addresses) on the outgoing interface over which a packet destined to this address must be forwarded to reach its final destination.

The main constraint imposed on the routing tables is that they must allow any host in the network to reach any other host. This implies that each router must know a route towards each destination but also that the paths composed from the information stored in the routing tables cannot not contain loops. Otherwise, some destinations would be unreachable.

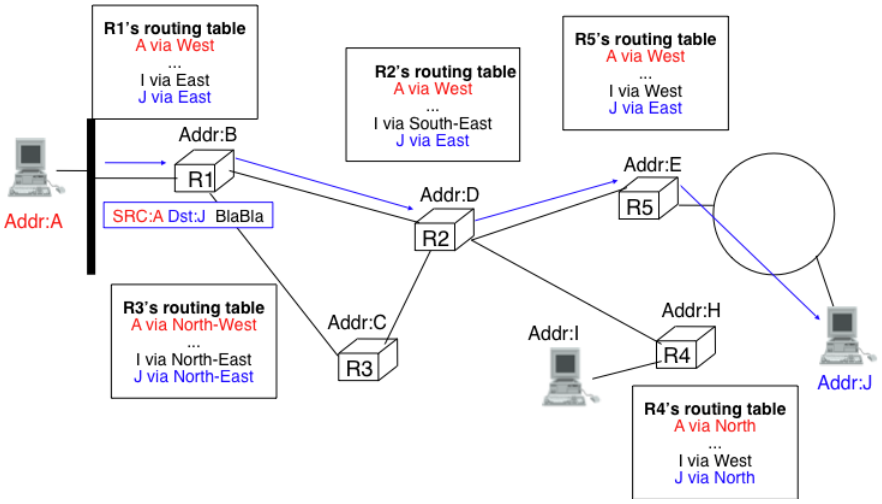


Figure 4.4: A simple internetwork

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.

To allow hosts to exchange packets, a network relies on two different types of protocols and mechanisms. First, there must be a precise definition of the format of the packets that are sent by hosts and processed by routers. Second, the algorithm used by the routers to forward these packets must be defined. This protocol and this algorithm are part of the *data plane* of the network layer. The *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.

The *data plane* and in particular the forwarding algorithm used by the routers depends on the routing tables that are maintained on reach router. These routing tables can be maintained by using various techniques (manual configuration, distributed protocols, centralised computation, ...). These techniques are part of the *control plane* of the network layer. The *control plane* contains all the protocols and mechanisms that are used to compute and install routing tables on the routers.

The datagram organisation has been very popular in computer networks. Datagram based network layers include IPv4 and IPv6 in the global Internet, CLNP defined by the ISO, IPX defined by Novell or XNS defined by Xerox.

Virtual circuit organisation

The main advantage of the datagram organisation is its simplicity. The principles of this organisation can be easily understood. Furthermore, it allows a host to easily send a packet towards any destination at any time. However, as each packet is forwarded independently by intermediate routers, packets sent by a host may not follow the same path to reach a given destination. This may cause packet reordering which may be annoying for transport protocols. Furthermore, as a router using *hop-by-hop forwarding* always forwards over the same outgoing interface the packets sent towards the same destination, this may cause congestion over some links.

The second organisation of the network layer, called *virtual circuits* has been inspired from the organisation of the 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-directionnal 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 the 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 identified with a network layer address. However, a host must explicitely request the establishment of a *virtual circuit before being able to send packets to a destination host. The request to establish a virtual circuit is processed by the 'control plane* that installs state to create the virtual circuit between the source and the destination through intermediate routers. All the packets that are sent on the virtual circuit contain a virtual circuit identifier that allows the routers to determine to which virtual circuit each packet belongs. This is illustrated in the figure below with one virtual circuit between host *A* and host *I* and another one between host *A* and host *J*.

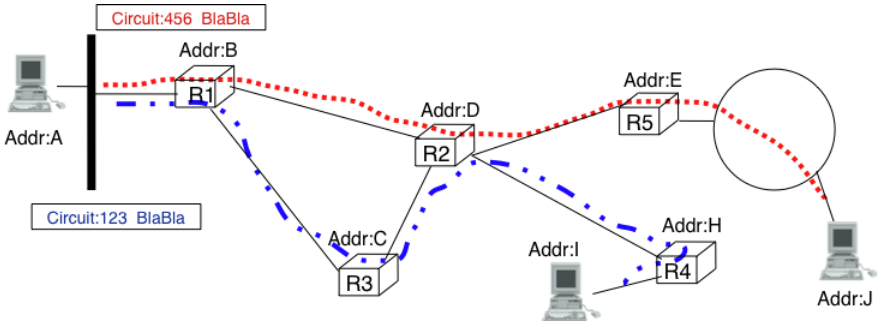


Figure 4.5: A simple internetwork using virtual-circuits

The establishment of a virtual circuit is performed by using a *signalling protocol* in the *control plane*. Usually, the source host sends a signalling message to indicate to its router the address of the destination and possibly some performance characteristics of the virtual circuit to be established. The first router can process the signalling message in two different ways.

A first solution is for the router to consult its routing table, remember the characteristics of the requested virtual circuit and forward it over its outgoing interface towards the destination. The signalling message is thus forwarded hop-by-hop until it reaches the destination and the virtual circuit is opened along the path followed by the signalling message. This is illustrated with the red virtual circuit in the figure below. A second solution can be used if the routers know the

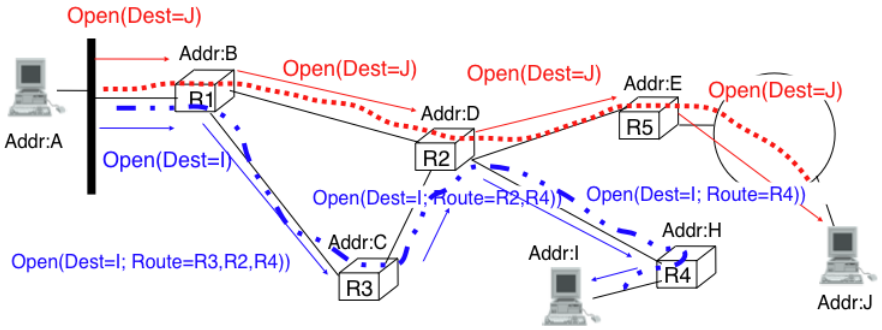


Figure 4.6: Virtual circuit establishment

entire topology of the network. In this case, the first router can use a technique called *source routing*. Upon reception of the signalling message, the first router chooses the path of the virtual circuit in the network. This path is encoded as the list of the addresses of all intermediate routers to reach the destination. It is included in the signalling message and intermediate routers can remove their address from the signalling message before forwarding it. This technique enables routers to better spread the virtual circuits throughout the network. If the routers know the load of remote links, they can also select the less loaded path when establishing a virtual circuit. This solution is illustrated with the blue circuit in the figure above.

The last point to be discussed about the virtual circuit organisation is its *data plane*. The *data plane* defines mainly the format of the data packets and the algorithm used by routers to forward packets. The data packets contain a virtual circuit identifier that is encoded as a fixed number of bits. These virtual circuit identifiers are usually called *labels*.

Each host maintains a flow table that associates a label with each virtual circuit that is has established. When a router receives a packet that contains a label, it extracts the label and consults its *label forwarding table*. This table is a data structure that maps each couple *(incoming interface, label)* to the outgoing interface to be used to forward the packet and the label that must be placed in the outgoing packets. In practice, the label forwarding table can be implemented as a vector and the couple *(incoming interface, label)* is the index of the entry in the vector that contains the outgoing interface and the outgoing label. Thus a single memory access is sufficient to consult the label forwarding table. The utilisation of the label forwarding table is illustrated in the figure below.

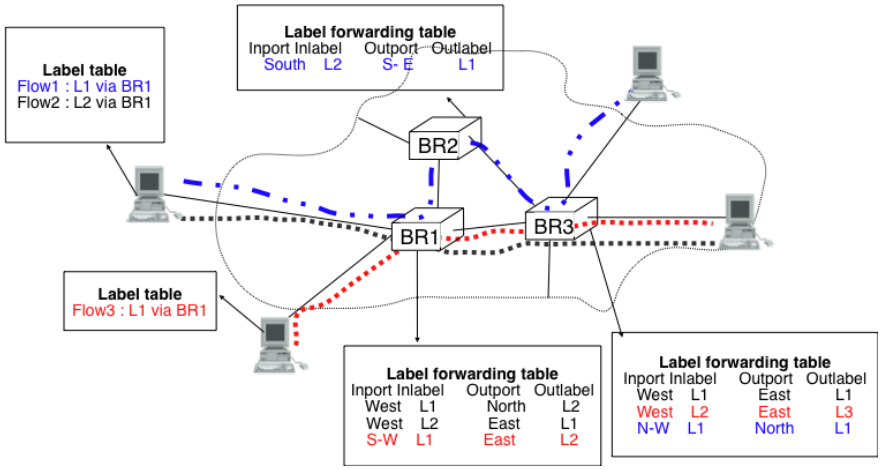


Figure 4.7: Label forwarding tables in a network using virtual circuits

The virtual circuit organisation has been mainly used in public networks, starting from X.25 and then Frame elay and Asynchronous Transfert Mode (ATM) network.

Both the datagram and virtual circuit organisations have advantages and drawbacks. The main advantage of the datagram organisation is that hosts can easily send packets to any number of destinations while the virtual circuit organisation requires the establishment of a virtual circuit before the transmission of data packet. This solution can be costly for hosts that exchange small amounts of data. On the other hand, the main advantage of the virtual circuit organisation is that the forwarding algorithm used by routers is simpler than when using the datagram organisation. Furthermore, the utilisation of virtual circuits may allow the load to be better spread through the network thanks to the utilisation of multiple virtual circuits. The MultiProtocol Label Switching (MPLS) technique that we discuss in chapter ??? can be considered as a good compromise between datagram and virtual circuits. MPLS uses virtual circuits between routers, but does not extend them to the endhosts.

4.1.2 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 three main techniques that can be used to maintain the routing tables in a network.

Static routing

The simplest solution is to pre-compute all the routing tables of all routers solution and to install them on each router. Several algorithms can be used to compute these tables. A simple solution is to use shortest path routing and

to minimise the number of intermediate routers to reach each destination. More complex algorithms can take into account the expected load on the links to ensure that congestion does not occur for a given traffic demand. Those algorithms must all ensure that :

- all routers are configured with a route to reach each destination
- the paths composed with the entries found in the routing tables do not cause forwarding loops

The figure below shows sample routing tables in a five routers network.

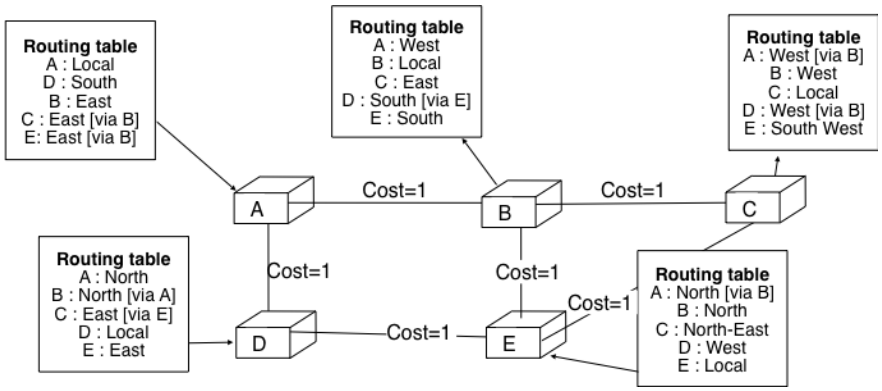


Figure 4.8: Routing tables in a simple network

The main drawback of static routing is that it does not adapt to the evolution of the network. When a new router or link is added, all routing tables must be recomputed. Furthermore, when a link or router fails, the routing tables must be updated as well.

Distance vector routing

Distance vector routing is a simple distributed routing protocol. Distance vector routing allows the routers to automatically discover the destinations that are reachable inside the network and 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 routinng table by using the pseudo-code below

```
Every N seconds:
 v=Vector()
 for each destination=d in R[]
 {
  v.add(Pair(d,R[d].cost));
 }
 for each interface
 {
```

```
  Send(v,interface)  # send vector v on this 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

```
Received(Vector V[],link l)
{ # received vector from link l
 for each destination=d in V[]
 {
  if not (d isin R[])
   { # new route
     R[d].cost=V[d].cost+l.cost;
     R[d].link=l;
     R[d].time=now;
   }
   else
   {
    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 five routers network shown below.

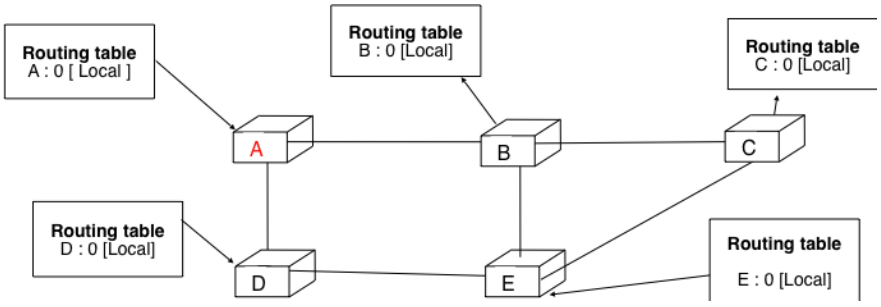


Figure 4.9: Operation of distance vector routing in a simple network

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=2]* 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.

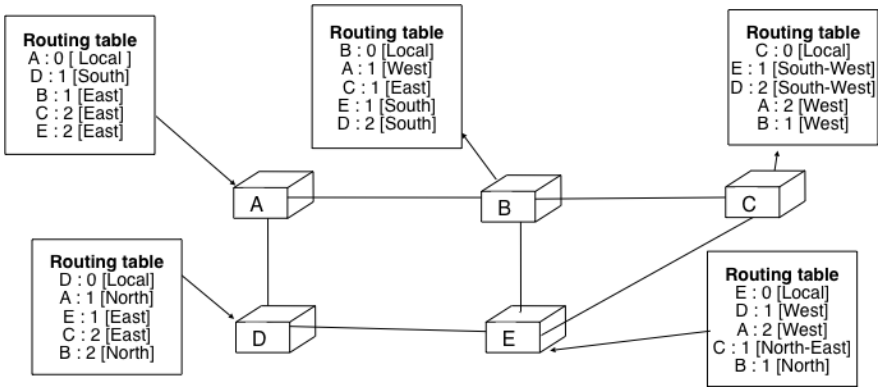


Figure 4.10: Routing tables computed by distance vector in a simple network

To deal with link and router failures, routers use the timestamp that is 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 every *N* seconds the timestamp of the routes stored in their routing table 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 neighbour 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 that allows them to reach all another routers, except router *A* that 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* that shows the routing tables on all routers. Consider now that the link between *D* and *E* fails. The network is now

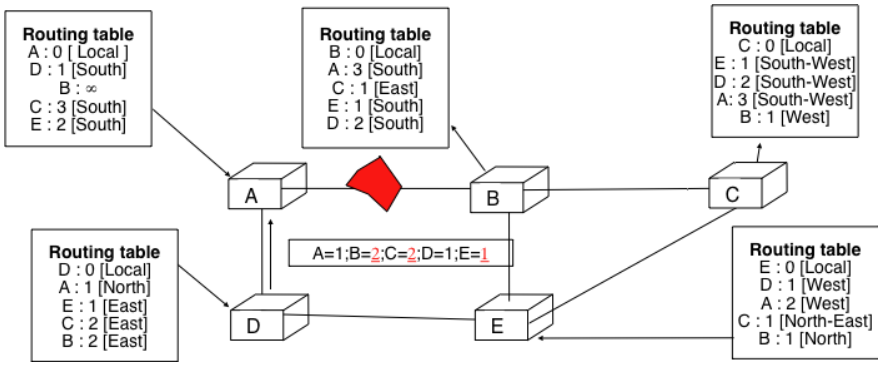


Figure 4.11: Routing tables computed by distance vector after a failure

partitionned in 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 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 $[A = 0, D = 1, B = 5, C = 5, E = 4]$ ... This problem is known as the *count to infinity problem* in the 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 avod 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:
 for each link=l
 { /* one different vector for each link */
  Vector=null;
  for each destination=d in R[]
  {
   if (R[d].link<>l)
     Vector=Vector+Pair(d,R[d].cost);
  }
  Send(Vector);
 }
```

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 pseudocode below

```
Every N seconds:
 for each link=l
 { /* one different vector for each link */
  Vector=null;
  for each destination=d in R[]
  {
   if (R[d].link<>l)
     Vector=Vector+Pair(d,R[d].cost);
   else
     Vector=Vector+Pair(d,infinity);
  }
  Send(Vector);
 }
```

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 four routers network below.

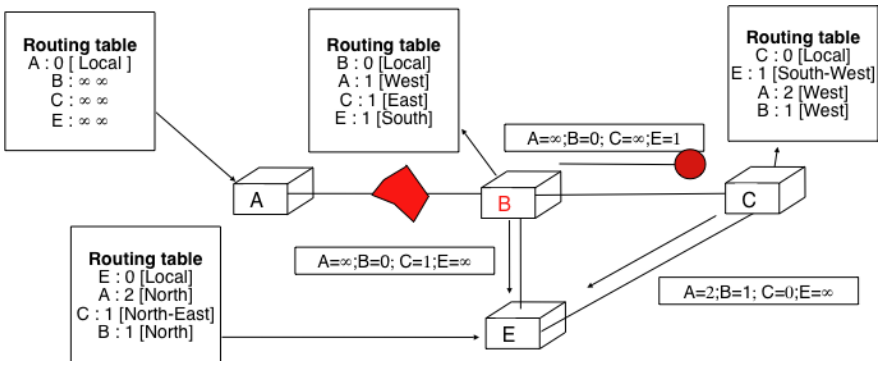


Figure 4.12: Count to infinity problem

After having detected the failure, router *A* 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*...

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 weights can be associated to each directed edge :

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

- weight proportionnal to the propagation delay on the link. If all links 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

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]. Usually, the same weight is associated to the two directed edges that correspond to a physical link.

However, in some cases, these weights can differ (e.g. because the uplink and downlink bandwidths are different or because one direction is overloaded and fewer destinations should be reached over this directed link). When a link-state router boots, it first needs to discover to which routers it is directly connected. For this, each router sends every *N* seconds a HELLO message on all 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 the direct neighbour. 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 neighboring router during a period of $k \times N$ seconds.

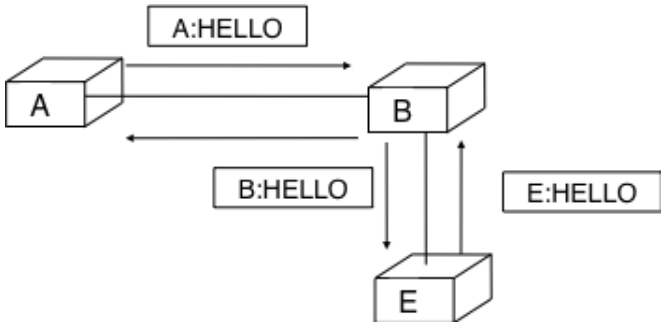


Figure 4.13: The exchange of HELLO messages

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) that contains 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 starting at this is represented with the following information : - LSP.Links[i].Id : identification of the neighbour - LSP.Links[i].cost : cost of the link

As the LSPs are used to distribute the network topology that allows routers to compute their routing tables, routers cannot rely on their non-existing routing tables to distribute the LSPs. *Flooding* is used to efficiently distribute the LSPs of all routers. Each router that implements *flooding* maintains a *link state database* (LSDB) that contains the most recent LSP sent by each router. When a router receives a 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 expect the link over which the LSP was received. Reliable flooding can be implemented by using the pseudo-code below

```
# 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 pseudocode, *LSDB(r)* returns the most recent *LSP* originated by router *r* that is stored in the *LSDB*. *newer(lsp1,lsp2)* returns true if *lsp1* is more recent than *lsp2*. See the sidebar below for a discussion on how *newer* can be implemented.

Which is the most recent LSP ?

A router that implements flooding must be able to detect whether a received LSP is newer than the received 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 arithemtic used to increment the sequence numbers. Intuitively, the comparaison divides the circle of all sequence numbers in 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* that is newer than *8*, but unfortunately *8* was considered as 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 number 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 a 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 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 a LSP to all routers in the network increases. In practice, routers flood immediately 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 originated by this router) [FFEB2005].

To ensure that all routers receive all LSPs even when there are transmissions errors, link state routing protocols use

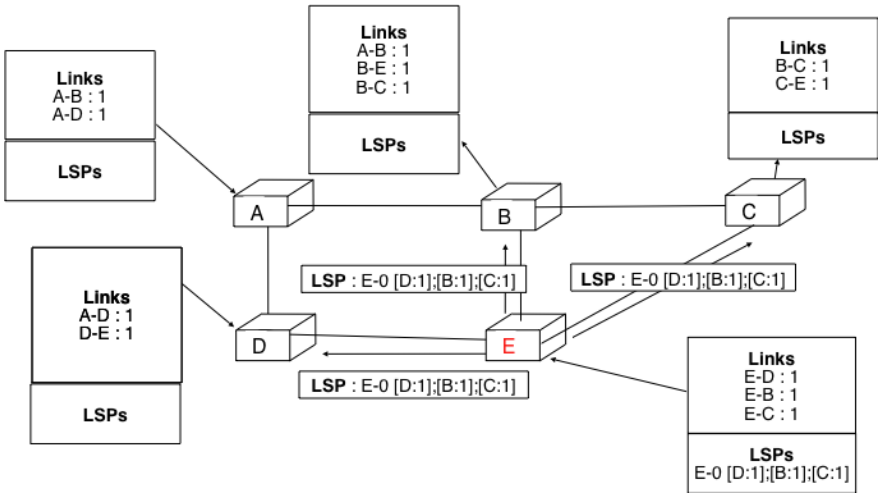


Figure 4.14: Flooding : example

reliable flooding. With *reliable flooding*, routers use acknowledgements and if necessary retransmissions to ensure that all link state packets are successfully transferred to all neighboring 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.

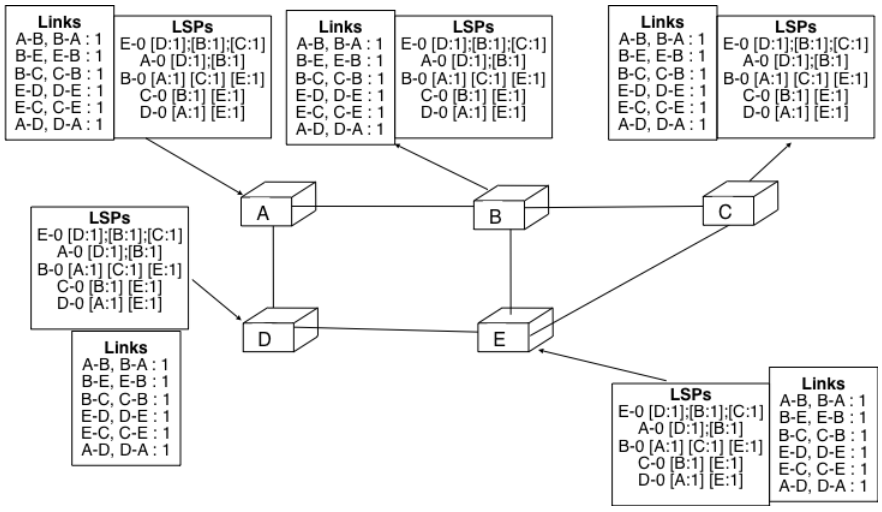


Figure 4.15: Link state databases received by all routers

Static or dynamic link metrics ?

As link state packets are flooded regularly, routers could measure the quality (e.g. delay or load) and their links and adjust the metric of each link according to its current quality. Such dynamic adjustements were included in the ARPANET routing protocol [MRR1979] . However, experience showed that it was difficult to tune the dynamic adjustements and ensure that no forwarding loops happen 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 during the

last $k \times N$ seconds. Once a router has detected a local link failure, it generates and floods a new LSP that does not contain anymore the failed link. 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.

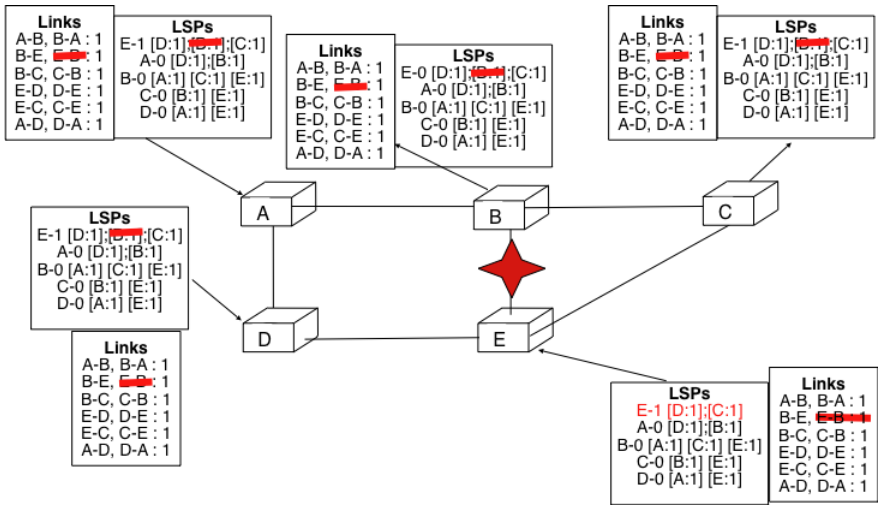


Figure 4.16: The two-way connectivity check

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 a 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 to deal with router failures. When a router fails, all its links fail by defintion. 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 [1]. 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 routing table, each router computes the spanning rooted at itself by using Dijkstra's shortest path algorithm [Dijkstra1959]. The routing table can be derived automatically from the spanning as shown in the figure below.

4.2 Internet Protocol

The Internet Protocol (IP) is the network layer protocol of the TCP/IP protocol suite. IP allows the applications running above the transport layer (UDP/TCP) to use a wide range of heterogeneous datalink layers. IP was designed when most point-to-point links were telephone lines with modems. Since then, IP has been able to use Local Area Networks (Ethernet, Token Ring, FDDI, ...), new wide area data link layer technologies (X.25, ATM, Frame Relay, ...) and more

[1] 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.

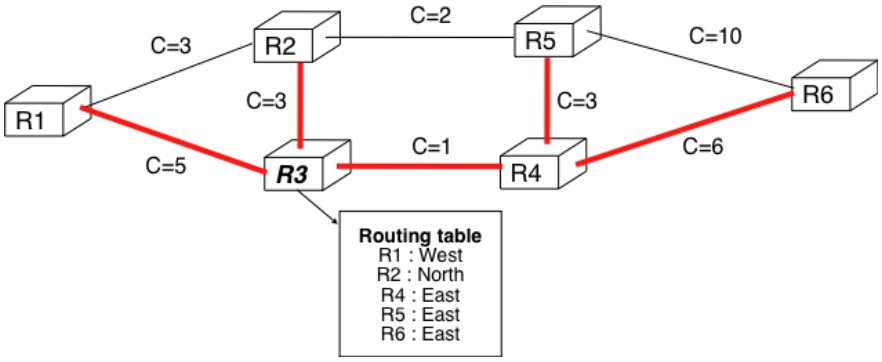


Figure 4.17: Computation of the routing table

recently wireless networks (802.11, 802.15, UMTS, GPRS, ...). The flexibility of IP and its ability to use various types of underlying data link layer technologies is one of its key advantages.

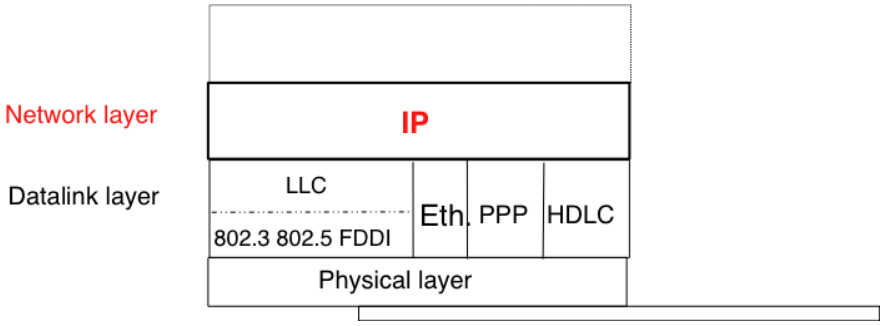


Figure 4.18: IP and the reference model

The current version of IP is version 4 specified in **RFC 791**. We first describe this version and later explain IP version 6 that is expected to replace IP version 4 in the future.

4.2.1 IP version 4

IP version 4 is the data plane protocol of the network layer in the TCP/IP protocol suite. The design of IP version 4 was based on the following assumptions :

- IP should provide an unreliable connectionless service (TCP provides reliability when required by the application)
- IP operates with the datagram transmission mode
- IP addresses have a fixed size of 32 bits
- IP must be useable above different types of datalink layers
- IP hosts exchange variable length packets

The addresses are an important part of any network layer protocol. In the late 1970s, the developers of IPv4 designed IPv4 for a research network that would interconnect some research labs and universities. For this utilisation, 32 bits wide addresses were much larger than the expected number of hosts on the network. Furthermore, 32 bits was a nice address size for software-based routers. None of the developers of IPv4 were expecting that IPv4 would become as widely used as it is today.

IPv4 addresses are encoded as a 32 bits field. IPv4 addresses are often represented in *dotted-decimal* format as a sequence of four integers separated by a *dot*. The first integer is the decimal representation of the most significant byte of the 32 bits IPv4 address, ... For example,

- 1.2.3.4 corresponds to 00000001000000100000001100000100
- 127.0.0.1 corresponds to 01111111000000000000000000000001
- 255.255.255.255 corresponds to 11111111111111111111111111111111

An IPv4 address is used to identify an interface on a router or a host. A router has thus as many IPv4 addresses as the number of interfaces that it has in the datalink layer. Most hosts have a single datalink layer interface and thus have a single IPv4 address. However, with the growth of wireless, more and more hosts have several datalink layer interfaces (e.g. an Ethernet interface and a WiFi interface). These hosts are said to be *multihomed*. A multihomed host with two interfaces has thus two IPv4 addresses.

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 IPv4 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 2.3.4.5 while another host located in Africa would use address 2.3.4.6. Unfortunately, this would force all routers to maintain a specific route towards each host. The figure below shows a simple enterprise network with two routers and three hosts and the associated routing tables if such isolated addresses were used. To preserve the scalability of the routing system, it is important

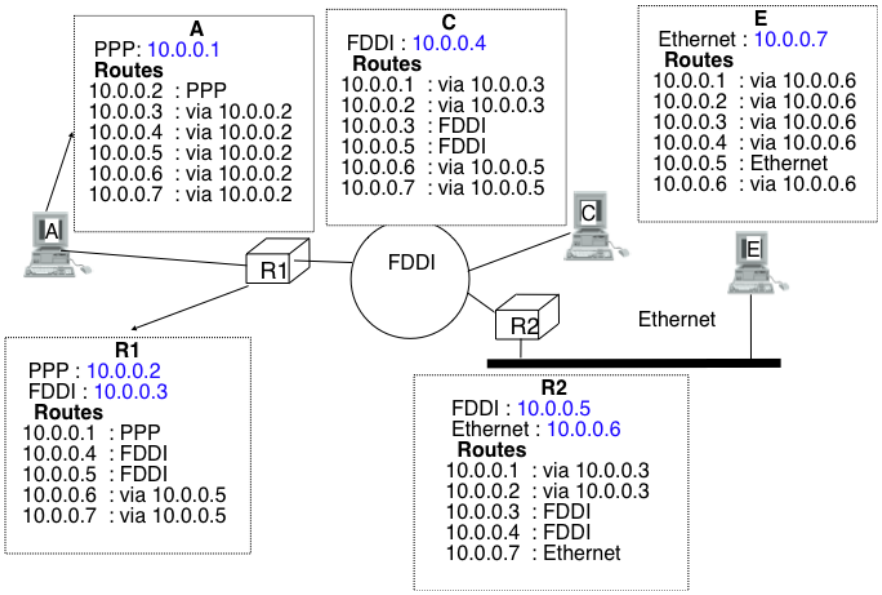


Figure 4.19: Scalability issues when using isolated IP addresses

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. An IPv4 address is composed of two parts : a *subnetwork identifier* and a *host identifier*. The *subnetwork identifier* is composed of the high order bits of the address and the host identifier is encoded in the low order bits of the address. This is illustrated in the figure below. When a router needs to forward a packet, it must know the *subnet* of the destination address to be able to consult its forwarding table to forward the packet. **RFC 791** proposed to use the high-order bits of the address to encode the length of the subnet identifier. This lead to the

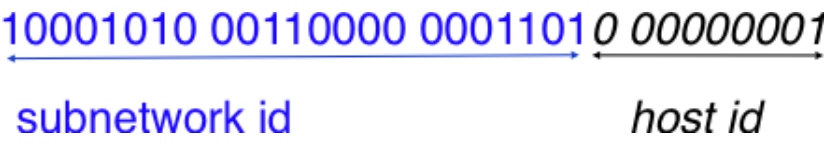


Figure 4.20: The subnetwork and host identifiers inside an IPv4 address

definition of three *classes* of unicast addresses [2]

Class	High-order bits	Length of subnet id	Number of networks	Addresses per network
Class A	*0*	8 bits	128	16,777,216 (2^{24})
Class B	*10*	16 bits	16,384	65,536 (2^{16})
Class C	*110*	24 bits	2,097,152	256 (2^{8})

However, these three classes of addresses were not flexible enough. A class *A* subnet was too large for most organisations and a class *C* subnet was too small. Flexibility was added by the introduction of *variable-length subnets* in **RFC 1519**. With *variable-length* subnets, the subnet identifier can have any size from *1* to *31* bits. *Variable-length* subnets allow the network operators to use a subnet that better matches the number of hosts that are placed inside the subnet. A subnet identifier or IPv4 prefix is usually [3] represented as *A.B.C.D/p* where *A.B.C.D* is the network address obtained by concatenating the subnet identifier with a host identifier containing only *0* and *p* is the length of the subnet identifier in bits. The table below provides examples of IP subnets.

Subnet	Number of addresses	Smallest address	Highest address
10.0.0.0/8	16,777,216	10.0.0.0	10.255.255.255
192.168.0.0/16	65,536	192.168.0.0	192.168.255.255
198.18.0.0/15	131,072	198.18.0.0	198.19.255.255
192.0.2.0/24	256	192.0.2.0	192.0.2.255
10.0.0.0/30	4	10.0.0.0	10.0.0.3
10.0.0.0/31	2	10.0.0.0	10.0.0.1

The figure below provides a simple example of the utilisation of IPv4 subnets in an enterprise network. The length of the subnet identifier assigned to a LAN usually depends on the expected number of hosts attached to the LAN. For point-to-point links, many deployments have used */30* prefixes, but recent routers are now using */31* subnets on point-to-point links **RFC 3021** or even do not use IPv4 addresses on such links [4].

A second issue concerning the addresses of the network layer is the allocation scheme that is used to allocated blocks of addresses to organisations. The first allocation scheme was based on the different classes of addresses. The pool of IPv4 addresses was managed by a secretariat that allocated address blocks on a first-come first served basis. Large organisations such as IBM, BBN, but also Stanford or the MIT were able to obtain a class *A* address block. Most organisations requested a class *B* address block that contains 65536 addresses, which was suitable for most enterprises and universities. The table below provides examples of some IPv4 address blocks in the class *B* space.

Subnet	Organisation
130.100.0.0/16	Ericsson, Sweden
130.101.0.0/16	University of Akron, USA
130.102.0.0/16	The University of Queensland, Australia
130.103.0.0/16	Lotus Development, USA
130.104.0.0/16	Université catholique de Louvain, Belgium
130.104.0.0/16	Open Software Foundation, USA

[2] In addition to the A, B and C classes, **RFC 791** also defined the *D* and *E* classes of IPv4 addresses. Class *D* (resp. *E*) addresses are those whose high order bits are set to *1110* (resp. *1111*). Class *D* addresses are used by IP multicast and will be explained later. Class *E* addresses are currently unused, but there are some discussions on possible future usages [WMH2008] [FLM2008]

[3] Another way of representing IP subnets is to use netmasks. A netmask is a 32 bits field whose *p* high order bits are set to *1* and the low order bits are set to *0*. The number of high order bits set *1* indicates the length of the subnet identifier. Netmasks are usually represented in the same dotted decimal format as IPv4 addresses. For example *10.0.0.0/8* would be represented as *10.0.0.0 255.0.0.0* while *192.168.1.0/24* would be represented as *192.168.1.0 255.255.255.0*. In some cases, the netmask can be represented in hexadecimal.

[4] A point-to-point link to which no IPv4 address has been allocated is called an unnumbered link. See **RFC 1812** section 2.2.7 for a discussion of such unnumbered links.

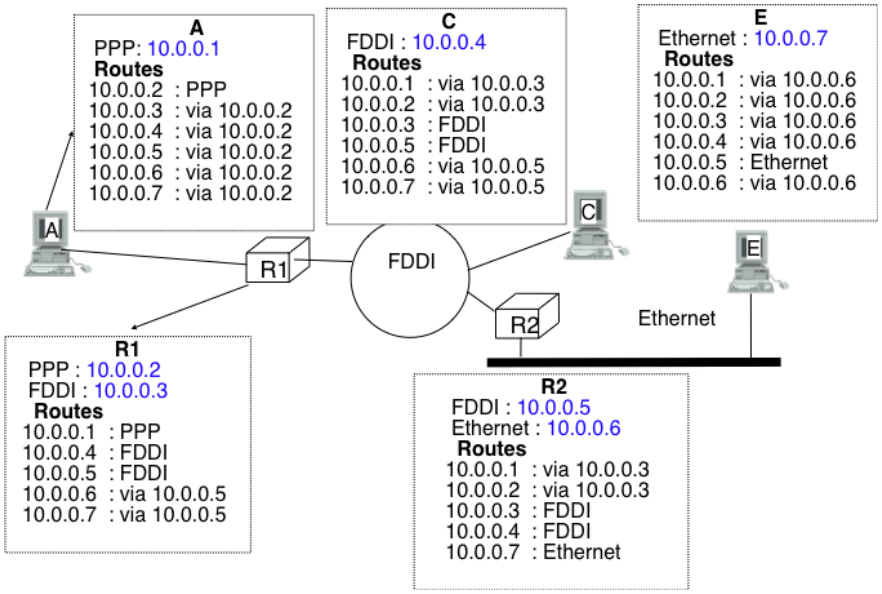


Figure 4.21: IP subnets in a simple enterprise network

However, the Internet was a victim of its own success and in the late 1980s, many organisations were requesting blocks of IPv4 addresses and connected to the Internet. Most of these organisations requested class *B* address blocks because class *A* address were too large and in limited supply while class *C* address blocks were considered to be too small. Unfortunately, there were only 16,384 different class *B* address blocks and this address space was being consumed quickly. As a consequence, the routing tables maintained by the routers were growing quickly and some routers had difficulties in maintaining all these routes in their limited memory [5]. Faced with these two problems,

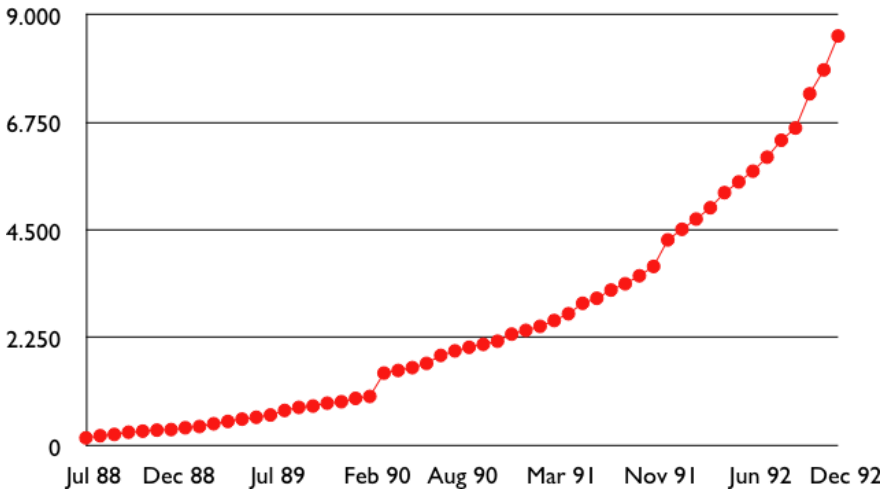


Figure 4.22: Evolution of the size of the routing tables on the Internet (Jul 1988- Dec 1992 - source : **RFC 1518**)

the Internet Engineering Task Force decided to develop the Classless Interdomain Routing (CIDR) architecture **RFC 1518**. This architecture aims at allowing IP routing to scale better than the class-based architecture. CIDR contains three important modifications compared to **RFC 791**.

1. IP address classes are deprecated. All IP equipments must use and support variable-length subnets.

[5] Example routers from this period include the Cisco AGS http://www.knossos.net.nz/don/wn1.html and AGS+ http://www.ciscopress.com/articles/article.asp?p=25296

2. IP address blocks are not allocated anymore on a first-come-first-served basis. Instead, CIDR introduces a hierarchical address allocation scheme.

3. IP routers must use longest-prefix match when they lookup a destination address in their forwarding table

The last two modifications were introduced to improve the scalability of the IP routing system. The main drawback of the first-come-first-served address block allocation scheme was that neighboring address blocks were allocated to very different organisations and conversely, very different address blocks were allocated to similar organisations. With CIDR, address blocks are allocated by Regional IP Registries (RIR) in an aggregatable manner. A RIR is responsible for a large block of addresses and a region. For example, RIPE is the RIR that is responsible for Europe. A RIR allocates smaller address blocks from its large block to Internet Service Providers **RFC 2050**. Internet Service Providers then allocate smaller address blocks to their customers, ... When an organisation requests an address block, it must prove that it already has or expects to have in the near future, a number of hosts or customers that is equivalent to the size of the requested address block.

The main advantage of this hierarchical address block allocation scheme is that it allows the routers to maintain fewer routes. For example, consider the address blocks that were allocated to some of the Belgian universities as shown in the table below.

Address block	Organisation
130.104.0.0/16	Université catholique de Louvain
134.58.0.0/16	Katholiek Universiteit Leuven
138.48.0.0/16	Facultés universitaires Notre-Dame de la Paix
139.165.0.0/16	Université de Liège
164.15.0.0/16	Université Libre de Bruxelles

These universities are all connected to the Internet exclusively via Belnet. As each university has been allocated a different address block, the routers of the Belnet must announce one route for each university and all routers on the Internet must maintain a route towards each university. In contrast, consider all the high schools and the government institutions that are connected to the Internet via Belnet. An address block was assigned to these institutions after the introduction of CIDR in the *193.190.0.0/15* address block owned by Belnet. With CIDR, Belnet can announce a single route towards *193.190.0.0/15* that covers all these high schools. However, there is one difficulty with the aggregatable variable length subnets used by CIDR. Consider for example FEDICT, a governmental institution that uses the *193.191.244.0/23* address block. Assume that in addition to being connected to the Internet via Belnet , FEDICT also wants to be connected to another Internet Service Provider. The FEDICT network is then said to be multihomed. This is shown in the figure below.

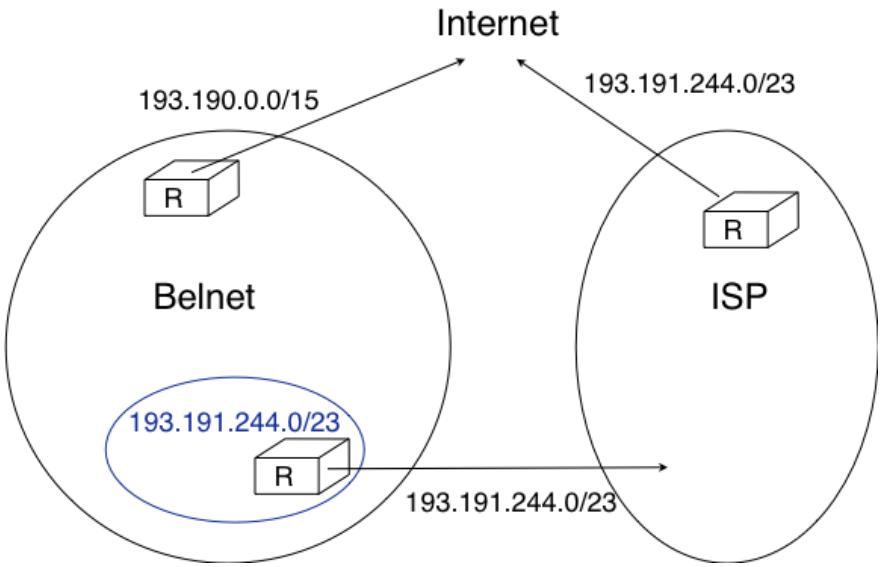


Figure 4.23: Multihoming and CIDR

With such a multihomed network, routers *R1* and *R2* would have two routes towards IPv4 address *193.191.245.88* : one route via Belnet (*193.190.0.0/15*) and one direct route (*193.191.244.0/23*). Both routes match IPv4 address *193.192.145.88*. 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 *193.191.245.88*, this is the route *193.191.244.0/23* that is used to forward the packet. This forwarding rule is called the *longest prefix match* or the *more specific match*. All IPv4 routers implement this forwarding rule.

To understand the *longest prefix match* forwarding, consider the figure below. With this rule, the route *0.0.0.0/0* 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 *192.168.1.1* received by router *R* is destined to the router itself. It is delivered to the appropriate transport protocol.
- a packet with destination *11.2.3.4* matches two routes : *11.0.0.0/8* and *0.0.0.0/0*. The packet is forwarded on the *West* interface.
- a packet with destination *130.4.3.4* matches one route : *0.0.0.0/0*. The packet is forwarded on the *North* interface.
- a packet with destination *4.4.5.6* matches two routes : *4.0.0.0/8* and *0.0.0.0/0*. The packet is forwarded on the *West* interface.
- a packet with destination *4.10.11.254* matches three routes : *4.0.0.0/8*, *4.10.11.0/24 and '0.0.0.0/0*. The packet is forwarded on the *South* interface.

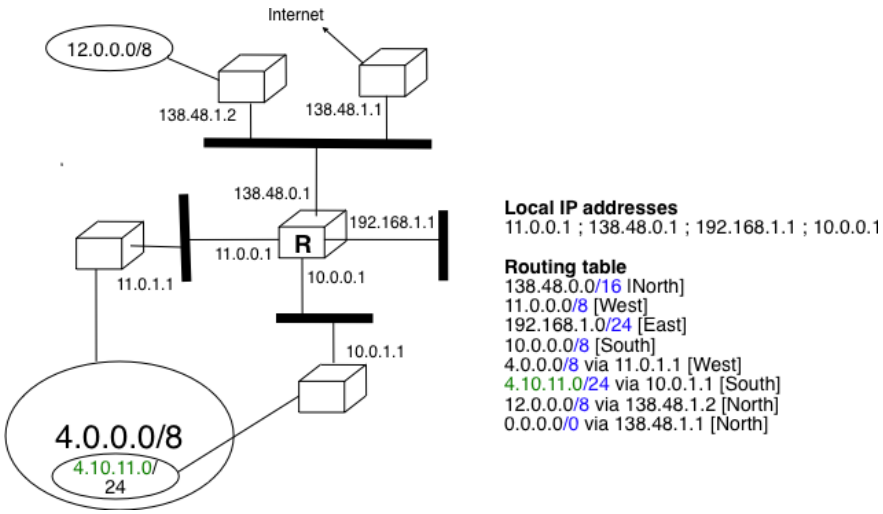


Figure 4.24: Longest prefix match example

The longest prefix match can be implemented by using different data structures. One possibility is to use a trie. The figure below shows a trie that encodes six routes having different outgoing interfaces.

Subnet	Next-hop
138.48.0.0/16	A
139.165.0.0/16	B
139.165.16.0/24	C
138.48.232.0/24	E
138.48.32.200/32	G
0.0.0.0/0	F

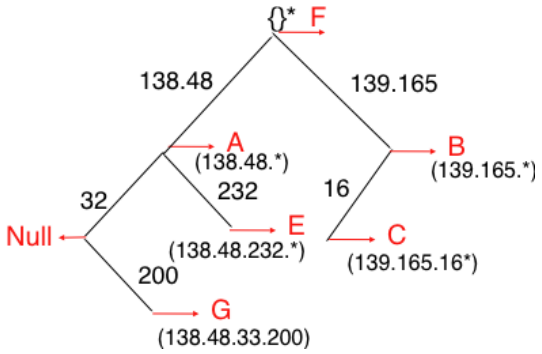


Figure 4.25: A trie representing a routing table

Special IPv4 addresses

Most unicast IPv4 addresses can appear as source and destination addresses in packets on the global Internet. However, it is worth to note that some blocks of IPv4 addresses have a special usage as described in **RFC 3330**. These include :

- *0.0.0.0/8* that is reserved for self-identification. A common address in this block is *0.0.0.0* that is sometimes used when a host boots and does not yet know its IPv4 address.
- *127.0.0.0/8* that is reserved for loopback addresses. Each host implementing IPv4 must have a loopback interface (that is not attached to a datalink layer). By convention, IPv4 address *127.0.0.1* is assigned to this interface. This allows processes running on a host to use TCP/IP to contact other processes running on the same host. This can be very useful for testing purposes.
- *10.0.0.0/8*, *172.16.0.0/12* and *192.168.0.0/16* are reserved for private networks that are not directly attached to the Internet. These addresses are often called private addresses or **RFC 1918** addresses.
- *169.254.0.0/16* is used for link-local addresses **RFC 3927**. Some hosts use an address in this block when they are connected to a network that does not allocate addresses as expected.

IPv4 packets

Now that we have clarified the allocation of IPv4 addresses and the utilisation of the longest prefix match to forward IPv4 packets, we can have a more detailed look at IPv4 by starting with the format of the IPv4 packets. The IPv4 packet format was defined in **RFC 791**. Besides a few clarifications and some backward compatible changes, the IPv4 packet format did not change significantly since the publication of **RFC 791**. All IPv4 packets use the 20 bytes header shown below. Some IPv4 packets contain an optional header extension that is described later.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |    DS Field   |          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The IP version 4 header
```

The main fields of the IPv4 header are :

- a 4 bits *version* that indicates the version of IP used to build the header. Using a version field in the header allows the network layer protocol to evolve.
- a 4 bits *IP Header Length (IHL)* that indicates the length of the IP header in 32 bits words. This field allows IPv4 to use options if required, but as it is encoded as a 4 bits field, the IPv4 header cannot be longer than 64 bytes.
- an 8 bits *DS* field that is used for Quality of Service and whose usage is described later.
- an 8 bits *Protocol* field that indicates the transport layer protocol that must process the packet's payload at the destination. Common values for this field [6] are *6* for TCP and *17* for UDP
- a 16 bits *length* field that indicates the total length of the entire IPv4 packet (header and payload) in bytes. This implies that an IPv4 packet cannot be longer than 65535 bytes.
- a 32 bits *source address* field that contains the IPv4 address of the source host
- a 32 bits *destination address* field that contains the IPv4 address of the destination host
- a 16 bits *checksum* that protects only the IPv4 header against transmission errors

The other fields of the IPv4 header are used for specific purposes. The first is the 8 bits *Time To Live (TTL)* field. This field is used by IPv4 to avoid the risk of having an IPv4 packet caught in an infinite loop due to a transient or permanent error in routing tables [7]. Consider for example the situation depicted in the figure below where destination *D* uses address *11.0.0.56*. If *S* sends a packet towards this destination, the packet is forwarded to router *B* that forwards it to router *C* that forwards it back to router *A*...

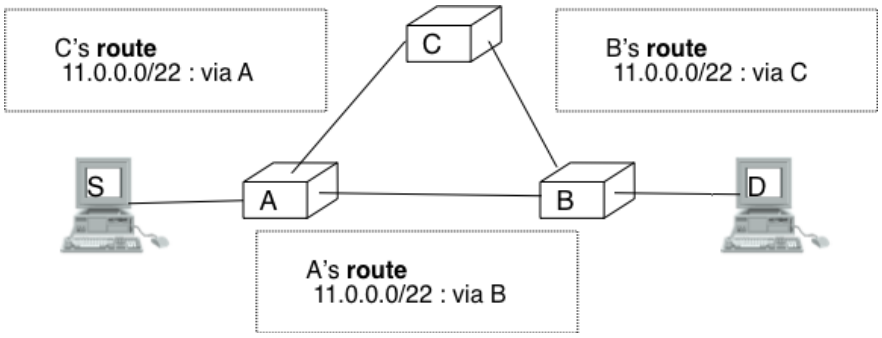


Figure 4.26: Forwarding loops in an IP network

Unfortunately, such loops can occur for two reasons in IP networks. First, if the network uses static routing, the loop can be caused by a simple configuration error. Second, if the network uses dynamic routing, such a loop can occur transiently, for example during the convergence of the routing protocol after a link or router failure. The *TTL* field of the IPv4 header ensures that even if there are forwarding loops in the network, packets will not loop forever. Hosts send their IPv4 packets with a positive *TTL* (usually *64* or more [8]). When a router receives an IPv4 packet, it first decrements the *TTL* by one. If the *TTL* becomes *0*, the packet is discarded and a message is sent back to the packet's source (see section ICMP). Otherwise, the router performs a lookup in its forwarding table to forward the packet. A second problem for IPv4 is the heterogeneity of the datalink layer. IPv4 is used above many very different datalink layers. Each datalink layer has its own characteristics and as indicated earlier, each datalink layer is characterised by a maximum frame size. From IP's viewpoint, a datalink layer interface is characterised by its *Maximum Transmission*

[6] See http://www.iana.org/assignments/protocol-numbers/ for the list of all assigned *Protocol* numbers

[7] The initial IP specification in **RFC 791** suggested that routers would decrement the *TTL* at least once every second. This would ensure that a packet would never remain for more than *TTL* seconds in the network. However, in practice most router implementations simply chose to decrement the *TTL* by one.

[8] The initial TTL value used to send IP packets vary from one implementation to another. Most current IP implementations use an initial TTL of 64 or more. See http://members.cox.net/~ndav1/self_published/TTL_values.html for additional information.

Unit (MTU). The MTU of an interface is the largest IPv4 packet (including header) that it can send. The table below provides some common MTU sizes [9].

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 IPv4 can send 64 KBytes long packets, few datalink layer technologies that are used today are able to send a 64 KBytes IPv4 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. To

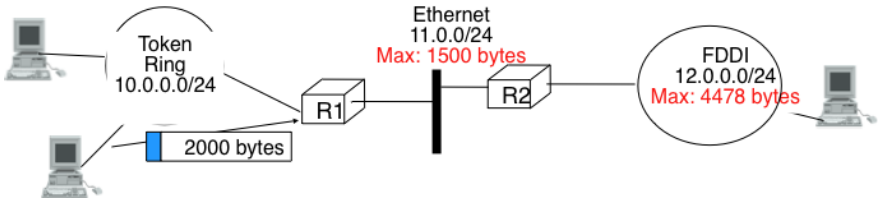


Figure 4.27: The need for fragmentation and reassembly

solve these problems, IPv4 includes a packet fragmentation and reassembly mechanism. Both hosts and intermediate routers may fragment an IPv4 packet if the packet is too long to be sent via the datalink layer. In IPv4, fragmentation is completely performed in the IP layer and a large IPv4 is fragmented into two or more IPv4 packets (called fragments). The IPv4 fragments of a large packet are normal IPv4 packets that are forwarded towards the destination of the large packet by intermediate routers.

The IPv4 fragmentation mechanism relies on four fields of the IPv4 header : *Length*, *Identification*, the *flags* and the *Fragment Offset*. The IPv4 header contains two flags : *More* and *Don't Fragment (DF)*. When the *DF* flag is set, this indicates that the packet cannot be fragmented. The basic operation of the IPv4 fragmentation is as follows. A large packet is fragmented into two or more fragments. The size of all fragments, except the last one, is equal to the Maximum Transmission Unit of the link used to forward the packet. Each IPv4 packet contains a 16 bits *Identification* field. When a packet is fragmented, the *Identification* of the large packet is copied in all fragments to allow the destination to reassemble the received fragments together. In each fragment, the *Fragment Offset* indicates, in units of 8 bytes, the position of the payload of the fragment in the payload of the original packet. The *Length* field in each fragment indicates the length of the payload of the fragment as in a normal IPv4 packet. Finally, the *More* flag is set only in the last fragment of a large packet.

The following pseudo-code details the IPv4 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-20)/8)  # must be n times 8 bytes
if p.flags=='DF' :
   discard(p)
# packet must be fragmented
payload=p[IP].payload
pos=0
while len(payload) > 0 :
```

[9] Supporting IP over the 802.15.4 datalink layer technology requires special mechanisms. See **RFC 4944** for a discussion of the special problems posed by 802.15.4

```
if len(payload) > maxpayload :
   toSend=IP(dest=p.dest,src=p.src,
             ttl=p.ttl, id=p.id,
             frag=p.frag+(pos/8),
             len=mtu, proto=p.proto)/payload[0:maxpayload]
   pos=pos+maxpayload
   payload=payload[maxpayload+1:]
else
   toSend=IP(dest=p.dest,src=p.src,
             ttl=p.ttl, id=p.id,
             frag=p.frag+(pos/8),
             flags=p.flags,
             len=len(payload), proto=p.proto)/payload
forward(toSend)
```

The fragments of an IPv4 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 IPv4 packet is a fragment or not by checking the value of the *More* flag and the *Fragment Offset*. If the *Fragment Offset* is set to *0* and the *More* flag is reset, the received packet has not been fragmented. Otherwise, the packet has been fragmented and must be reassembled. 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 *More* 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. The original IP specification defined in **RFC 791** several types of options that can be added to the IP header. Each option is encoded by using a *type length value* format. They are not widely used today and are thus only briefly described. Additional details may be found in **RFC 791**.

The most interesting options in IPv4 are the three options that are related to routing. The *Record route* option was defined to allow network managers to determine the path followed by a packet. When the *Record route* option was present, routers on the packet's path had to insert their IP address in the option. This option was implemented, but as the optional part of the IPv4 header can only contain 44 bytes, it is impossible to discover an entire path on the global Internet. `traceroute(8)`, despite its limitations, is a better solution to record the path towards a destination.

The other routing options are the *Strict source route* and the *Loose source route* option. The main idea behind these options is that a host may want, for any reason, to specify the path to be followed by the packets that it sends. The *Strict source route* option allows a host to indicate inside each packet the exact path to be followed. The *Strict source route* option contains a list of IPv4 address and a pointer to indicate the next address in the list. When a router receives a packet containing this option, it does not lookup the destination address in its routing table but forwards the packet directly to the next router in the list and advances the pointer. This is illustrated in the figure below where *S* forces its packets to follow the *RA-RB-RD* path.

The maximum length of the optional part of the IPv4 header is a severe limitation for the *Strict source route* option as for the *Record Route* option. The *Loose source route* option does not suffer from this limitation. This option allows the sending host to indicate inside its packet *some* of the routers that must be traversed to reach the destination. This is shown in the figure below. *S* sends a packet containing a list of addresses and a pointer to the next router in the list. Initially, this pointer points to *RB*. When *RA* receives the packet sent by *S*, it looks up in its forwarding table the address pointed in the *Loose source route* option and not the destination address. The packet is then forwarded to

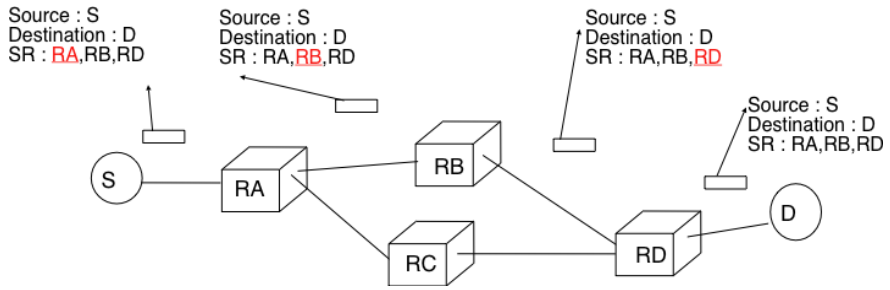


Figure 4.28: Usage of the *Strict source route* option

router *RB* that recognises its address in the option and advances the pointer. As there is no address listed in the *Loose source route* option anymore, *RB* and other downstream routers forward the packet by performing a lookup for the destination address.

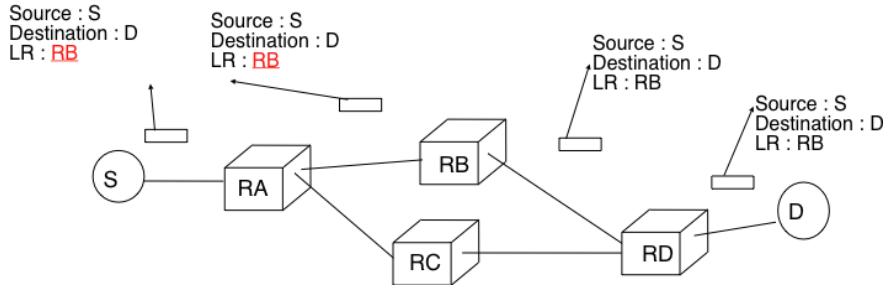


Figure 4.29: Usage of the *Loose source route* option

These two options are usually ignored by routers because they cause security problems.

4.2.2 ICMP version 4

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). ICMP is defined in **RFC 792**. ICMP messages are carried as the payload of IP packets (the protocol value reserved for ICMP is *1*). An ICMP message is composed of an 8 byte header and a variable length payload that usually contains the first bytes of the packet that triggered the transmission of the ICMP message.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |    DS Field   |          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
|                               Data                                    |
~                                                                       ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      IPv4 header of errored packet + 64 bits of payload               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

ICMP version 4 (:rfc:`792`)

In the ICMP header, the *Type* and *Code* fields indicate the type of problem that was detected by the sender of the ICMP message. The *Checksum* protects the entire ICMP message against transmission errors and the *Data* field contains additional information for some ICMP messages.

The main types of ICMP messages are :

- *Destination unreachable* : a *Destination unreachable* ICMP message is sent when a packet cannot be delivered to its destination due to routing problems. Different types of unreachability are distinguished :
  - *Network unreachable* : this ICMP message is sent by a router that does not have a route for the subnet containing the destination address of the packet
  - *Host unreachable* : this ICMP message is sent by a router that is attached to the subnet that contains the destination address of the packet, but this destination address cannot be reached at this time
  - *Protocol unreachable* : this ICMP message is sent by a destination host that has received a packet, but does not support the transport protocol indicated in the packet's *Protocol* field
  - *Port unreachable* : this ICMP message is sent by a destination host that has received a packet destined to a port number, but no server process is bound to this port
- *Fragmentation needed* : this ICMP message is sent by a router that receives a packet with the *Don't Fragment* flag set that is larger than the MTU of the outgoing interface
- *Redirect* : this ICMP message can be sent when there are two routers on the same LAN. Consider a LAN with one host and two routers : *R1* and *R2*. Assume that *R1* is also connected to subnet *130.104.0.0/16* while *R2* is connected to subnet *138.48.0.0/16*. If a host on the LAN sends a packet towards *130.104.1.1* to *R2*, *R2* needs to forward the packet again on the LAN to reach *R1*. This is not optimal as the packet is sent twice on the same LAN. In this case, *R2* could send an ICMP *Redirect* message to the host to inform it that it should have sent the packet directly to *R1*. This allows the host to send the other packets to *130.104.1.1* directly via *R1*.

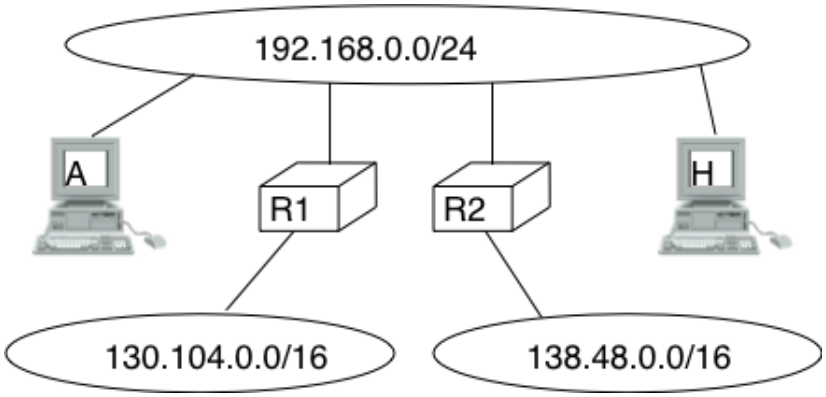


Figure 4.30: ICMP redirect

- *Parameter problem* : this ICMP message is sent when a router or a host receives an IP packet containing an error (e.g. an invalid option)
- *Source quench* : a router was supposed to send this message when it had to discard packets due to congestion. However, sending ICMP messages in case of congestion was not the best way to reduce the congestion and since the inclusion of a congestion control scheme in TCP, this ICMP message has been deprecated.
- *Time Exceeded* : there are two types of *Time Exceeded* ICMP messages
  - *TTL exceeded* : a *TTL exceeded* message is sent by a router when it discards an IPv4 packet because its *TTL* reached *0*.
  - *Reassembly time exceeded* : this ICMP message is sent when a destination has been unable to reassemble all the fragments of a packet before the expiration of its reassembly timer.
- *Echo request* and *Echo reply* : these ICMP messages are used by the `ping(8)` network debugging software.

Redirection attacks

ICMP redirect messages are useful when several routers are attached to the same LAN as hosts. However, they should be used with care as they also create an important security risk. One of the most annoying attack in an IP network is called the *man in the middle attack*. Such an attack occurs if an attacker is able to receive, process, possibly modify and forward all the packets exchanged between a source and a destination. As the attacker receives all the packets it can easily collect passwords or credit card numbers or even inject fake information in an established TCP connection. ICMP redirects unfortunately enable an attacker to easily perform such an attack. In the figure above, consider host *H* that is attached to the same LAN as *A* and *R1*. If *H* sends to *A* an ICMP redirect for prefix *138.48.0.0/16*, *A* forwards to *H* all the packets that it wants to send to this prefix. *H* can then forward them to *R2*. To avoid these attacks, host should ignore the ICMP redirect messages that they receive.

`ping(8)` is often used by network operators to verify that a given IP address is reachable. Each host is supposed [10] to reply with an ICMP *Echo reply* message when its receives an ICMP *Echo request* message. A sample usage of `ping(8)` is shown below

```
ping 130.104.1.1
PING 130.104.1.1 (130.104.1.1): 56 data bytes
64 bytes from 130.104.1.1: icmp_seq=0 ttl=243 time=19.961 ms
64 bytes from 130.104.1.1: icmp_seq=1 ttl=243 time=22.072 ms
64 bytes from 130.104.1.1: icmp_seq=2 ttl=243 time=23.064 ms
64 bytes from 130.104.1.1: icmp_seq=3 ttl=243 time=20.026 ms
64 bytes from 130.104.1.1: icmp_seq=4 ttl=243 time=25.099 ms
--- 130.104.1.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 19.961/22.044/25.099/1.938 ms
```

Another very useful debugging tool is `traceroute(8)`. The traceroute man page describes this tool as *"print the route packets take to network host"*. traceroute uses the *TTL 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 *TTL* is set to *1* it decrements the *TTL* and is forced to return to the sending host a *TTL exceeded* ICMP message containing the header and the first bytes of the discarded IP packet. To discover all routers on a network path, a simple solution is to first send a packet whose *TTL* is set to *1*, then a packet whose *TTL* is set to *2*, ... A sample traceroute output is shown below

[10] 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.

```
traceroute www.ietf.org
traceroute to www.ietf.org (64.170.98.32), 64 hops max, 40 byte packets
 1  CsHalles3.sri.ucl.ac.be (192.168.251.230)  5.376 ms  1.217 ms  1.137 ms
 2  CtHalles.sri.ucl.ac.be (192.168.251.229)  1.444 ms  1.669 ms  1.301 ms
 3  CtPythagore.sri.ucl.ac.be (130.104.254.230)  1.950 ms  4.688 ms  1.319 ms
 4  fe.m20.access.lln.belnet.net (193.191.11.9)  1.578 ms  1.272 ms  1.259 ms
 5  10ge.cr2.brueve.belnet.net (193.191.16.22)  5.461 ms  4.241 ms  4.162 ms
 6  212.3.237.13 (212.3.237.13)  5.347 ms  4.544 ms  4.285 ms
 7  ae-11-11.car1.Brussels1.Level3.net (4.69.136.249)  5.195 ms  4.304 ms  4.329 ms
 8  ae-6-6.ebr1.London1.Level3.net (4.69.136.246)  8.892 ms  8.980 ms  8.830 ms
 9  ae-100-100.ebr2.London1.Level3.net (4.69.141.166)  8.925 ms  8.950 ms  9.006 ms
 10  ae-41-41.ebr1.NewYork1.Level3.net (4.69.137.66)  79.590 ms
     ae-43-43.ebr1.NewYork1.Level3.net (4.69.137.74)  78.140 ms
     ae-42-42.ebr1.NewYork1.Level3.net (4.69.137.70)  77.663 ms
 11  ae-2-2.ebr1.Newark1.Level3.net (4.69.132.98)  78.290 ms  83.765 ms  90.006 ms
 12  ae-14-51.car4.Newark1.Level3.net (4.68.99.8)  78.309 ms  78.257 ms  79.709 ms
 13  ex1-tg2-0.eqnwnj.sbcglobal.net (151.164.89.249)  78.460 ms  78.452 ms  78.292 ms
 14  151.164.95.190 (151.164.95.190)  157.198 ms  160.767 ms  159.898 ms
 15  ded-p10-0.pltn13.sbcglobal.net (151.164.191.243)  161.872 ms  156.996 ms  159.425 ms
 16  AMS-1152322.cust-rtr.swbell.net (75.61.192.10)  158.735 ms  158.485 ms  158.588 ms
 17  mail.ietf.org (64.170.98.32)  158.427 ms  158.502 ms  158.567 ms
```

The above *traceroute(8)* output shows a 17 hops path between a host at UCLouvain and one of the main IETF servers. For each hop, traceroute provides the IPv4 address of the router that sent the ICMP message and the measured round-trip-time between the source and this router. traceroute sends three probes with each *TTL* value. In some cases, such as at the 10th hop above, the ICMP messages may be received from different addresses. This is usually because different packets from the same source have followed different paths [11] in the network. Another important utilisation of ICMP messages is to discover the maximum MTU that can be used to reach a destination without fragmentation. As explained earlier, when an IPv4 router receives a packet that is larger than the MTU of the outgoing link, it must fragment the packet. Unfortunately, fragmentation is a complex operation and routers cannot perform it at line rate [KM1995]. Furthermore, 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].

ICMP, combined with the *Don't fragment (DF)* IPv4 flag, is used by TCP implementations to discover the largest MTU size to be used to reach a destination host without causing network fragmentation. This is the *Path MTU discovery* mechanism defined in **RFC 1191**. A TCP implementation that includes *Path MTU discovery* (most do) requests the IPv4 layer to send all segments inside IPv4 packets having the *DF* flag set. This prohibits intermediate routers from fragmenting these packets. If a router needs to forward an unfragmentable packet over a link with a smaller MTU, it returns a *Fragmentation needed* ICMP message to the source indicating the MTU of its outgoing link. This ICMP message contains in its *Data* field the MTU of the router's outgoing link. Upon reception of this 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.

Interactions between IPv4 and the datalink layer

As mentionned in the first section of this chapter, there are three main types of datalink layers : *point-to-point* links, LANs supporting broadcast and multicast and NBMA networks. There are different issues to be addressed when using IPv4 in these types of networks. The first issue is how an IPv4 device determines its IPv4 address. The second issue is how IPv4 packets are exchanged over the datalink layer service.

[11] A detailed analysis of traceroute output is outside the scope of this document. Additional information may be found in [ACO+2006] and [DT2007]

On a *point-to-point* link, the IPv4 addresses of the communicating devices can be configured manually or through a protocol. IPv4 addresses are often configured manually on *point-to-point* links between routers. When *point-to-point* links are used to attach hosts to the network, automatic configuration is often preferred to avoid problems with incorrect IPv4 addresses. For example, the PPPP (Point-to-Point Protocol), specified in **RFC 1661** includes an IP network control protocol that can be used by the router in the figure below to advertise the IPv4 address that the attached hosts must configure for its interface. The transmission of IPv4 packets on a point-to-point link is usually easy. Depending on the particular datalink layer considered, this may require some fragmentation and reassembly mechanisms in the datalink layer. Furthermore, IPv4 should be informed of the link's MTU. We will discuss these issues in the next chapter.

Figure 4.31: IPv4 on point-to-point links

Using IPv4 in a LAN introduces an additonal problem. On a LAN, each device is identified by its unique datalink layer address. The datalink layer service can be used by any host attached to the LAN to send a frame to any other host attached to the same LAN provided that the sending host knows the datalink layer of the destination host. For example, the figure below shows four hosts attached to the same LAN configured with IPv4 addresses in the *10.0.1.0/24* subnet and datalink layer addresses represented as a single character [12]. In this network, if host *10.0.1.22/24* wants to send an IPv4 packet to the host having address *10.0.1.8*, it must know that the datalink layer address of this host is *C*. In a simple network such as the one shown above, it could be possible to manually configure the mapping between the IPv4 addresses of the hosts and the corresponding datalink layer addresses. However, in a larger LAN this is impossible and it should be possible for an IPv4 host to automatically obtain the datalink layer address that corresponds to any IPv4 address on the same LAN. This is the objective of the *Address Resolution Protocol* (*ARP*) defined in **RFC 826**. ARP is a datalink layer protocol that is used by IPv4. It relies on the ability of the datalink layer service to easily deliver a broadcast frame to all devices attached to the same LAN. The easiest way to unerstand the operation of ARP is to consider the simple network shown above and assume that host *10.0.1.22/24* needs to send an IPv4 packet to host *10.0.1.8*. As this IP address belongs to the same subnet, the packet must be sent directly to its destination via the datalink layer service. To use this service, the sending host must find the datalink layer address that is attached to host

[12] In practice, most local area networks use addresses encoded as a 48 bits field **[802]_**. Some recent local area network technologies use 64 bits addresses.

Figure 4.32: A simple LAN

10.0.1.8. Each IPv4 host maintains an *ARP cache* that contains the list of all the mappings between IPv4 addresses and datalink layer addresses that it knows. When an IPv4 hosts boots, its ARP cache is empty. *10.0.1.22* thus consults first its ARP cache. As the cache does not contain the requested mapping, host *10.0.1.22* sends a broadcast ARP query frame on the LAN. The frame contains the datalink layer address of the sending host (*A*) and the requested IPv4 address (*10.0.1.8*). This broadcast frame is received by all devices on the LAN and only the host that owns the requested IPv4 address replies by returning a unicast ARP reply frame with the requested mapping. Upon reception of this reply, the sending host updates its ARP cache and sends the IPv4 packet by using the datalink layer service. To deal with devices that move or whose addresses are reconfigured, some ARP implementations remove the cache entries that have not been used for a few minutes. Other implementations revalidate ARP cache entries from time to time by sending ARP queries [13].

[13] See chapter 28 of [Benvenuti2005] for a description of the implementation of ARP in the Linux kernel.

Security issues with the Address Resolution Protocol

ARP is an old and widely used protocol that was unfortunately designed when security issues were not a concern. *ARP* is almost insecure by design. Hosts using *ARP* can suffer from several types of attacks. First, a malicious host could create a denial of service attack on a LAN by sending random replies to the received ARP queries. This would pollute the ARP cache of the other hosts on the same LAN. On a fixed network, such attacks are usually fixed by the system administrator who physically removes the malicious hosts from the LAN. On a wireless network, this is much more difficult.
A second type of attack are the *man-in-the-middle* attacks. This name is used for network attacks where the attacker is able to read and possibly modify all the messages sent by the attacked users. Such an attack is possible in a LAN. Assume in the figure above that host *10.0.1.9* is malicious and would like to receive and modify all packets sent by host *10.0.1.22* to host *10.0.1.8*. This can be easily achieved is host *10.0.1.9* manages by sending fake ARP replies to convince host *10.0.1.22* (resp. *10.0.1.8*) that his own datalink layer address must be used to reach *10.0.1.8* (resp. *10.0.1.22*).

ARP is used by all devices that are connected to LANs and implement IPv4. Both routers and endhosts implement ARP. When a host needs to send an IPv4 packet to a destination outside of its local subnet, it must first send the packet to one of the routers that reside on this subnet. Consider for example the network shown in the figure below. Each host is configured with an IPv4 address in the *10.0.1.0/24* subnet and uses *10.0.1.1* as its default router. To send a packet to *1.2.3.4*, host *10.0.1.8* will first need to know the datalink layer of the default router. It will thus send an ARP request for *10.0.1.1*. Upon reception of the ARP reply, host *10.0.1.8* will be able to send its packet in a frame to its default router. The router will then forward the packet towards its final destination. In the early days of the Internet, IP

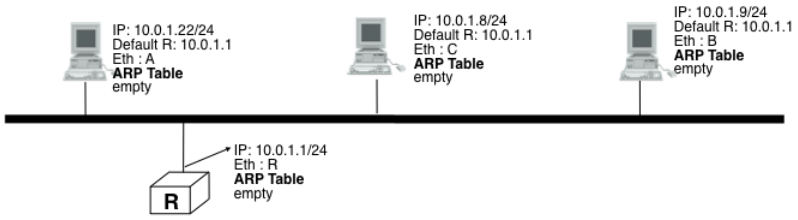


Figure 4.33: A simple LAN with a router

addresses were manually configured on both hosts and routers and never changed. However, this manual configuration can be complex [14] and often causes errors that are sometimes difficult to debug. Recent TCP/IP implementations are able to detect some of these misconfigurations. For example, if two hosts are configured on the same subnet with the same IPv4 address they will be unable to communicate. To detect this problem hosts send an ARP request for their

[14] For example, consider all the options that can be specified for the *ifconfig utility<http://en.wikipedia.org/wiki/Ifconfig>* on Unix hosts.

configured address each time their addressed is changed **RFC 5227**. If they receive an answer to this ARP request, they trigger an alarm or inform the system administrator.

To ease the attachment of hosts to subnets, most networks now support the Dynamic Host Configuration Protocol (DHCP) **RFC 2131**. DHCP allows a host to automatically retrieve its assigned IPv4 address. A DHCP server is associated to each subnet [15]. Each DHCP server manages a pool of IPv4 addresses assigned to the subnet. When a host is first attached to the subnet, it sends a DHCP request message. This message is placed in a UDP segment (the DHCP server listens on port 67). As the host does not know neither its IPv4 address nor the IPv4 address of the DHCP server, this UDP segment is placed in an IPv4 packet whose source and destination addresses are respectively *0.0.0.0* and *255.255.255.255*. The DHCP request may contain various options such as the name of the host, its datalink layer address, ... The server captures the DHCP request and finds an unassigned address in its address pool. It then sends the assigned IPv4 address in a DHCP reply message that contains the datalink layer address of the host and additional information such as the subnet mask of the IPv4 address, the address of the default router or the address of the DNS resolver. This DHCP reply message is sent in an IPv4 packet whose source and destination addresses are respectively the IPv4 address of the DHCP server and the *255.255.255.255* broadcast address. The DHCP reply also contains the lifetime of the address allocation. The host must renew its address allocation once it expires.

In an NBMA network, the interactions between IPv4 and the datalink layer address are more complex as the ARP protocol cannot be used as in a LAN. Such NBMA networks use special servers that store the mappings between IP addresses and the corresponding datalink layer address. Asynchronous Transfer Mode (ATM) networks for example can use either the ATMARP protocol defined in **RFC 2225** or the NextHop Resolution Protocol (NHRP) defined in **RFC 2332**. ATM networks are less frequently used today and we will not describe the detailed operation of these servers.

Operation of IPv4 devices

At this point of the description of IPv4, it is useful to have a detailed look at how an IPv4 implementation sends, receives and forwards IPv4 packets. The simplest case is when a host needs to send a segment in an IPv4 packet. The host performs two operations. First, it must decide on which interface the packet will be sent. Second it must create the corresponding IP packet(s).

To simplify the discussion in this section, we ignore the utilisation of IPv4 options. This is not a severe limitation as today IPv4 packets rarely contain options. Details about the processing of the IPv4 options may be found in the relevant RFCs such as **RFC 791**. Furthermore, we also assume that only point-to-point links are used. We defer the explanation of the operation of IPv4 over Local Area Networks until the next chapter.

An IPv4 host having n datalink layer interfaces manages $n + 1$ IPv4 addresses :

- the *127.0.0.1/32* IPv4 address assigned by convention to its loopback address
- one *A.B.C.D/p* IPv4 address assigned to each of its n datalink layer interfaces

Such a host maintains a routing table that contains one entry for its loopback address and one entry for each subnet identifier assigned to its interfaces. Furthermore, the host usually uses one of its interfaces as the *default* interface when sending packets that are not addressed to a directly connected destination. This is represented by the *default* route : *0.0.0.0/0* that is associated to one interface.

When a transport protocol running on the host requests the transmission of a segment, it usually provides to the IPv4 layer the IPv4 destination address in addition to the segment [16]. The IPv4 implementation first performs a longest prefix match with the destination address in its routing table. The lookup returns the identification of the interface that must be used to send the packet. The host can then create the IPv4 packet that contains the segment. The source IPv4 address of the packet is the IPv4 address of the host on the interface returned by the longest prefix match. The *Protocol* field of the packet is set to the identification of the local transport protocol that created the segment. The *TTL* field of the packet

[15] 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.

[16] A transport protocol implementation can also specify whether the packet must be sent with the *DF* set or set. A TCP implementation using *Path MTU Discovery* would always request the transmission of IPv4 packets with the *DF* flag set.

is set to the default *TTL* used by the host. The host must now choose the packet's *Identification*. This *Identification* is important if the packet becomes fragmented in the network as it ensures that the destination is able to reassemble the received fragments. Ideally, a sending host should never send twice a packet with the same *Identification* to the same destination host to ensure that all fragments are correctly reassembled by the destination. Unfortunately, with a 16 bits *Identification* field and an expected MSL of 2 minutes, this implies that the maximum bandwidth to a given destination is limited to roughly 286 Mbps. With a more realistic 1500 bytes MTU, that bandwidth drops to 6.4 Mbps **RFC 4963** if fragmentation must be possible [17]. This is very low and is another reason why hosts are highly encouraged to avoid fragmentation. If despite of this the MTU of the outgoing interface is smaller than the packet's length, the packet is fragmented. Finally, the packet's checksum is computed before transmission.

When a host receives an IPv4 packet destined to itself, there are several operations that it must perform. First, it must check the packet's checksum. If the checksum is incorrect, the packet is discarded. Then, it must check whether the packet has been fragmented. If yes, the packet is passed to the reassembly algorithm described earlier. Otherwise, the packet must be passed to the upper layer. This is done by looking at the *Protocol* field (*6* for TCP, *17* for UDP). If the host does not implement the transport layer protocol corresponding to the received *Protocol* field, it sends a *Protocol unreachable* ICMP message to the sending host. If the received packet contains an ICMP message (*Protocol* field set to *1*), the processing is more complex. An *Echo-request* ICMP message triggers the transmission of an *ICMP Echo-reply* message. The other types of ICMP messages indicate an error that was caused by a previously transmitted packet. These ICMP messages are usually forwarded to the transport protocol that sent the erroneous packet. This can be done by inspecting the contents of the ICMP message that includes the header and the first 64 bits of the erroneous packet. If the IP packet did not contain options, which is the case for most IPv4 packets, the transport protocol can find in the first 32 bits of the transport header the source and destination ports to determine the affected transport flow. This is important for Path MTU discovery for example.

When a router receives an IPv4 packet, it must first check the packet's checksum. If the checksum is invalid, it is discarded. Otherwise, the router must check whether the destination address is one of the IPv4 addresses assigned to the router. If so, the router must behave as a host and process the packet as described above. Although routers mainly forward IPv4 packets, they sometimes need to be accessed as hosts by network operators or network management software.

If the packet is not addressed to the router, it must be forwarded on an outgoing interface according to the router's routing table. The router first decrements the packet's *TTL*. If the *TTL* reaches *0*, a *TTL Exceeded* ICMP message is sent back to the source. As the packet header has been modified, the checksum must be recomputed. Fortunately, as IPv4 uses an arithmetic checksum, a router can incrementally update the packet's checksum as described in **RFC 1624**. Then, the router performs a longest prefix match for the packet's destination address in its forwarding table. If no match is found, the router must return a *Destination unreachable* ICMP message to the source. Otherwise, the lookup returns the interface over which the packet must be forwarded. Before forwarding the packet over this interface, the router must first compare the length of the packet with the MTU of the outgoing interface. If the packet is smaller than the MTU, it is forwarded. Otherwise, a *Fragmentation needed* ICMP message is sent if the *DF* flag was sent or the packet is fragmented if the *DF* was not set.

Longest prefix match in IP routers

Performing the longest prefix match at line rate on routers requires highly tuned data structures and algorithms. Consider for example an implementation of the longest match based on a Radix tree on a router with a 10 Gbps link. On such a link, a router can receive 31,250,000 40 bytes IPv4 packets every second. To forward the packets at line rate, the router must process one IPv4 packet every 32 nanoseconds. This cannot be achieved by a software implementation. For a hardware implementation, the main difficulty lies in the number of memory accesses that are necessary to perform the longest prefix match. 32 nanoseconds is very small compared to the memory accesses that are required by a naive longest prefix match implement. Additional information about faster longest prefix match algorithms may be found in [Varghese2005].

[17] It should be noted that only the packets that can be fragmented (i.e. whose *DF* flag is reset) must have different *Identification* fields. The *Identification* field is not used in the packets having the *DF* flag set.

4.2.3 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 be quickly allocated. In 1987, a study cited in **RFC 1752** estimated 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 solutions 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** that allowed multiple hosts to share a single public IP address. NAT is explained in section *Middleboxes*.

However, in parallel with these short-term solutions, that 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 proposal outline 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 on the basis 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 be easily parsed by hardware devices
- A host should be able to configure its IPv6 address automatically
- Security must be part of IPv6

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 on the most suitable address length. A first approach, proposed by SIP in **RFC 1710** was to use 64 bits addresses. A 64 bits address space was 4 billion times larger than the IPv4 address space and furthermore from an implementation viewpoint, 64 bits CPU were being considered and 64 bits 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 as a reasonable compromise at that time.

IPv6 addressing architecture

The experience with 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 × 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 that offers only 8 addresses per square kilometer, this is a significant improvement on paper.

IPv6 supports unicast, multicast and anycast addresses. As with IPv4, 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.

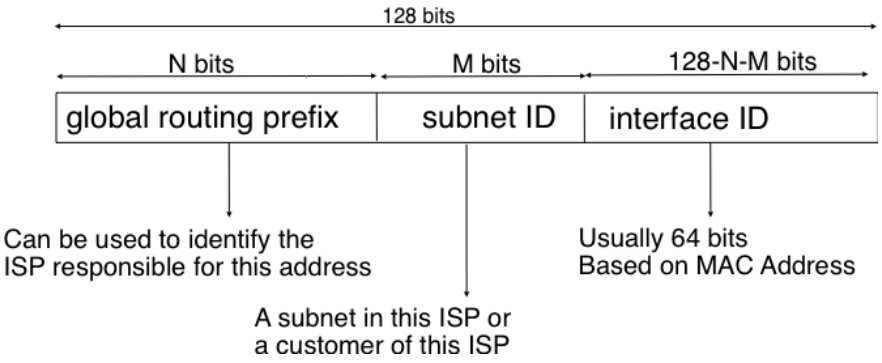


Figure 4.34: Structure of IPv6 unicast addresses

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

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 viewpoint as explained in section ICMPv6

Textual representation of IPv6 addresses

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

- ABCD:EF01:2345:6789: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

There are in practice 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 of 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 of provider, it needs to change all the addresses that it uses. This can be a nightmare from an operational viewpoint 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
- /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

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. This is the equivalent of *127.0.0.1* in IPv4.
- *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.

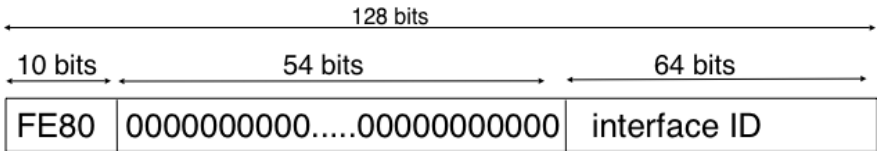


Figure 4.35: IPv6 link local address structure

An important consequence of the IPv6 unicast addressing architecture and the utilisation of link-local addresses is that an IPv6 host has several IPv6 addresses. This implies that an IPv6 stack must be able to handle multiple IPv6 addresses. This was not always the case with IPv4.

RFC 4291 defines a special type of IPv6 anycast address. On a subnetwork having prefix *p/n*, the IPv6 address whose *128-n* low-order bits are set to *0* is the anycast address that corresponds to all routers inside this subnetwork. This anycast address can be used by hosts to quickly send a packet to any of the routers inside their own subnetwork.

Finally, **RFC 4291** defines the structure of the IPv6 multicast addresses [18]. This structure is depicted in the figure below

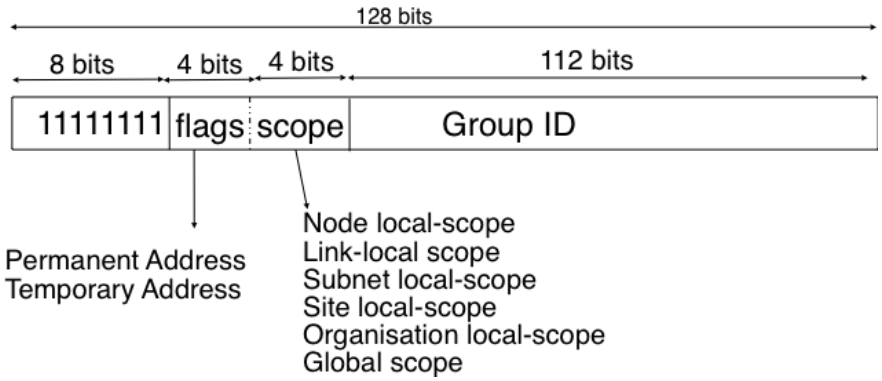


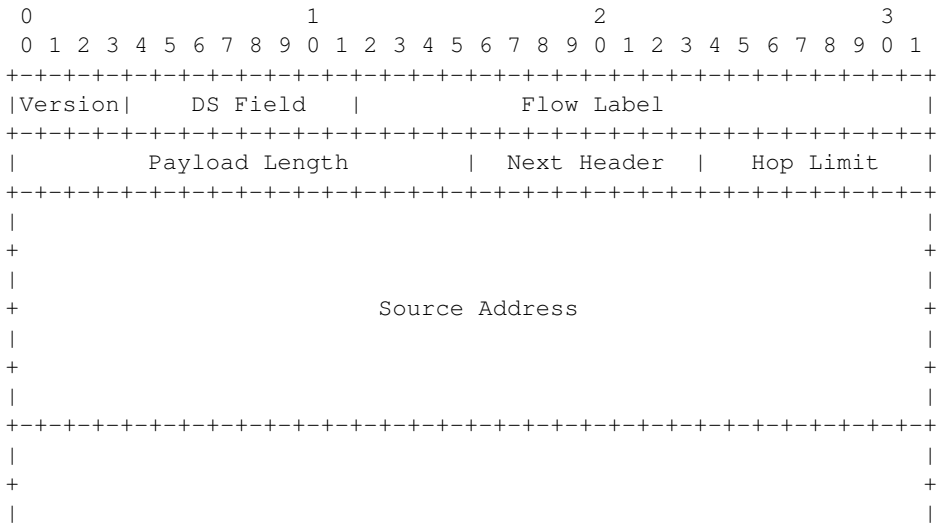
Figure 4.36: IPv6 multicast address structure

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. The 4 bits flag field indicates notably 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. We discuss IPv6 multicast later.

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.



[18] The full list of allocated IPv6 multicast addresses is available at http://www.iana.org/assignments/ipv6-multicast-addresses

```
+                       Destination Address                     +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The IP version 6 header (:rfc:`2460`)

Besides the source and destination addresses, the IPv6 header contains the following fields :

- *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 plays a similar role as the *DS* byte in the IPv4 header
- *Flow label* : this field was initially intended to be used to tag packets belonging to the same *flow*. However, as of this writing, there is no clear guideline on how this field should be used by hosts and routers
- *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.
- *Next Header* : this 8 bits field indicates the type [19] 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. Handling options as a next header allows to simplify the processing of IPv6 packets compared to IPv4.
- *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 has the same purpose as the TTL field of the IPv4 header.

In comparison with IPv4, the IPv6 packets are much simpler and easier to process by routers. A first important difference is 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 a limited benefit in detecting errors. In practice, an IP checksum allows to catch 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.

A second difference with IPv4 is that the IPv6 header does not support fragmentation and reassembly. The experience with IPv4 has shown that fragmenting packets in routers was costly [KM1995] and the developers of IPv6 have decided that routers would not fragment packets anymore. If a router receives a packet that is too long to be forwarded, the packet is dropped and the router returns an ICMPv6 messages 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.

The third difference are the IPv6 options that are simpler and easier to process than the IPv4 options.

[19] The IANA maintains the list of all allocated Next Header types at http://www.iana.org/assignments/protocol-numbers/ The same registry is used for the IPv4 protocol field and for the IPv6 Next Header.

Header compression on low bandwidth links

Given the size of the IPv6 header, it can cause a 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**. The 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 2507** take advantage of the redundancy found in successive packets from the same flow to reduce significantly 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**.

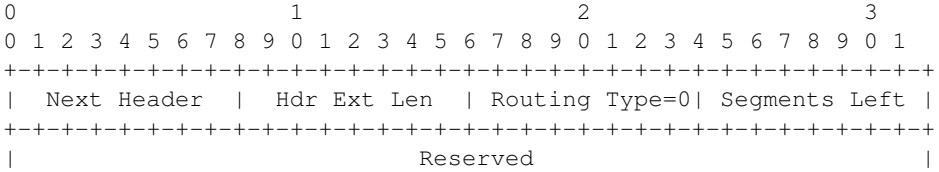
IPv6 options

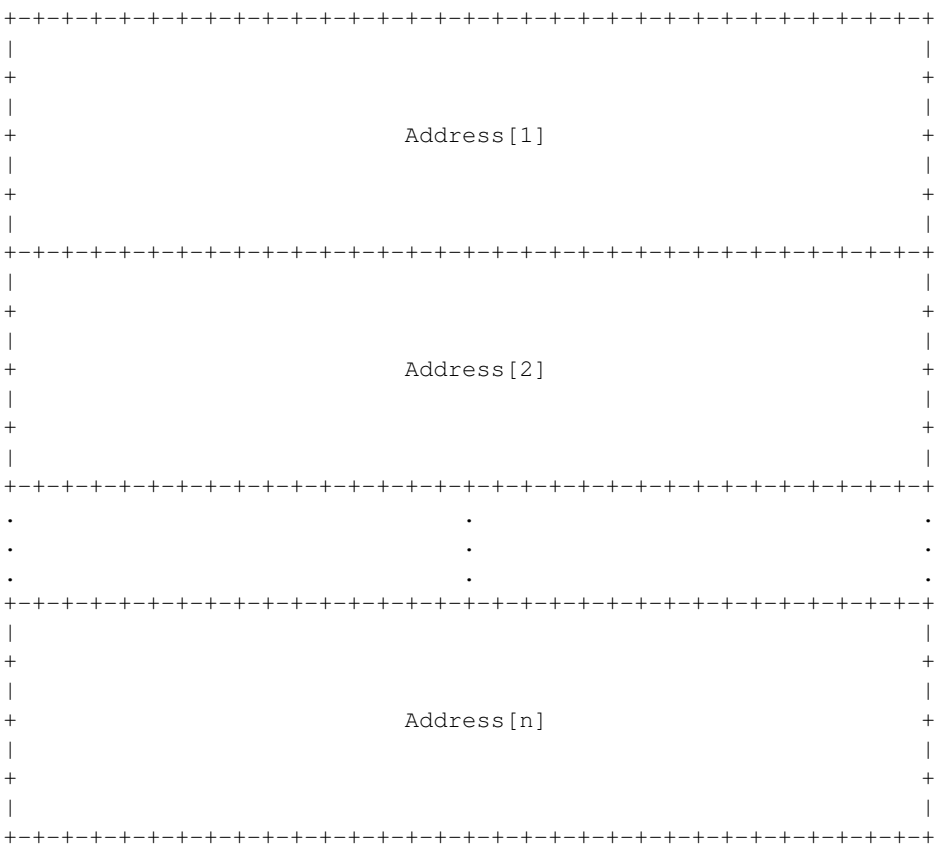
In IPv6, each option is considered as one header containing a multiple of 8 bytes to ensure that IPv6 options in a packet are aligned on 64 bits boundaries. IPv6 defines several types of options :

- the hop-by-hop options are the options that must be processed by the routers on the packet's path
- the type 0 routing header that is similar to the IPv4 loose source routing option
- the fragmentation option that is used when fragmenting an IPv6 packet
- the destination options
- the security options that allow IPv6 hosts to exchange packets with cryptographic authentication (AH header) or encryption and authentication (ESP header)

RFC 2460 provides lots of details on the encodings of the different types of options. In this section, we only discus some of them. The reader may consult **RFC 2460** for more information about the other options. The first point to note is that each option contains a *Next Header* field that indicates the type of the next header that follows the option. A second point to note is that to allow routers to efficiently parse IPv6 packets, the options that must be processed by routers (hop-by-hop options and type 0 routing header) must appear first in the packet. This allows the router to process a packet without being forced to analyse all the packet's options. A third point to note is that hop-by-hop and destination options are encoded by using a *type length value* format. Furthermore, the *type* field contains bits that indicate whether a router that does not understand this option should ignore the option or discard the packet. This allows to introduce new options in the network without forcing all devices to be upgraded to support it at the same time. 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. Its usage is explained later.

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 header (:rfc:`2460`)

The type 0 routing option was intended to allow a host to indicate a loose source router 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 some 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**. 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 that 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 that is 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.

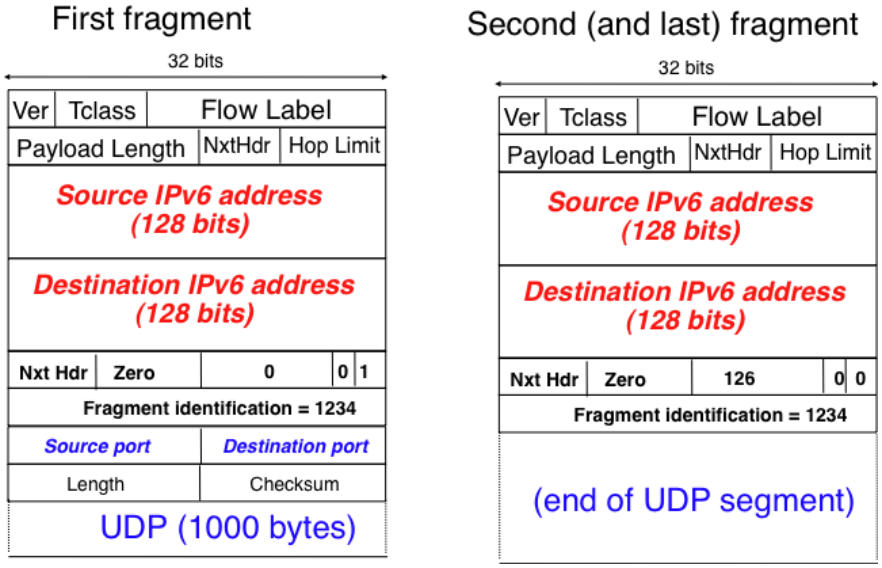


Figure 4.37: IPv6 fragmentation example

Finally, the last type of IPv6 options are the Encaspulating Security Payload (ESP) defined in **RFC 4303** and the Authentication Header (AH) defined in **RFC 4302**. These two headers are used by IPSec **RFC 4301**. They are discussed in another chapter.

4.2.4 ICMP version 6

ICMPv6 defined in **RFC 4443** is the companion protocol for IPv6 as ICMPv4 is the companion protocol for IPv4. ICMPv6 is used by routers and hosts to report problems when processing IPv6 packets. However, as we will see in chapter `../lan/lan.rst`, ICMPv6 is also used when auto-configuring addresses.

The traditional utilisation of ICMPv6 is similar to ICMPv4. 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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |           Checksum              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Message Body                          +
|                                                               |
```

ICMP version 6 packet format

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** :

- **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 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 messages 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
- 2 : Packet Too Big. The router that sends 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 messages 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.

Two types of informational ICMPv6 messages are defined in **RFC 4443** : *echo request* and *echo reply* that are used to test the reachability of a destination by using `ping6(8)`. ICMPv6 also allows to discover the path between a source and a destination by using `traceroute6(8)`. The output below shows a traceroute between a host at UCLouvain and one of the main IETF servers. Note that this IPv6 path is different than the IPv4 path that was described earlier although the two traceroutes were performed at the same time

```
traceroute6 www.ietf.org
traceroute6 to www.ietf.org (2001:1890:1112:1::20) from 2001:6a8:3080:2:217:f2ff:fed6:65c0, 30 hops r
 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.belnet
 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
 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
```

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 [a].

[a] 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

Interactions between IPv6 and the datalink layer

There are several differences between IPv6 and IPv4 when considering their interactions with the datalink layer. ICMPv6 plays a key role in these interactions. ICMPv6 is first used to resolve the datalink layer address that corresponds to a given IPv6 address. This part of ICMPv6 is the Neighbour Discovery Protocol (NDP) defined in **RFC 4861**. NDP is similar to ARP, but there are two important differences. First, NDP messages are exchanged in ICMPv6 messages while ARP messages are sent as datalink layer frames. Second, an ARP request is sent as a broadcast frame while an NDP sollication message is sent as a multicast ICMPv6 packet that is transported inside a multicast frame. The operation of the NDP protocol is as simple as ARP. To obtain an address mapping, a host sends a Neighbour Sollicitation message. This message is sent inside an ICMPv6 message that is placed in an IPv6 packet whose source address is the IPv6 address of the requesting host and the destination address is the all-hosts IPv6 multicast address (*FF02::1*) to which all IPv6 hosts listen. The HopLimit of the IPv6 packet is set to *255* **RFC 58032**. The Neighbour Sollication contains the requested IPv6 address. The owner of the requested address replies by sending a unicast Neighbour Advertisement message to the requesting host. NDP suffers from similar security issues as the ARP protocol. A Secure Neighbour Discovery Protocol has been defined in **RFC 3971**, but a detailed description of this protocol is outside the scope of this chapter. IPv6 networks also support the Dynamic Host Configuration Protocol. The IPv6 extensions to DHCP are defined in **RFC 3315**. The operation of DHCPv6 is similar to DHCP that was described earlier. In addition to DHCPv6, IPv6 networks support another mechanism to assign IPv6 addresses to hosts. This is the Stateless Address Configuration (SLAC) defined in **RFC 4862**. When a host boots, it derives its identifier from its datalink layer address [20] and concatenates this 64 bits identifier to the *FE80::/64* prefix to obtain its link-local IPv6 address. It then sends a Neighbour Sollicitation 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 that indicates that the link-local address is used by another host, it generates another 64 bits identifier and sends again a Neighbour Sollicitation. If there is no answer, the host consider 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 IPv6 prefix that is used on the local subnet. IPv6 routers send regularly ICMPv6 Router Advertisement messages that indicate the IPv6 prefix assigned to each subnet. 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 SLAC by sending a Neighbour Sollicitation message targeted at its global IPv6 address to ensure that another host is not using the same IPv6 address.

[20] 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 everytime it attaches to a subnet. It then becomes impossible for an attacker to use the 64-bits identifier to track a host.

4.2.5 Middleboxes

When the TCP/IP architecture and the IP protocol were defined, two types of devices were considered in the network layer : endhosts and routers. Endhosts are the sources and destinations of IP packets while routers forward packets. When a router forwards an IP packet, it consults its forwarding table, updates the packets' TTL, recomputes its checksum and forward it to the nexthop. A router does not need to read nor change the contents of the packet's payload.

However, in today's Internet, there exist devices that are not strictly routers but process, sometimes modify, and forward IP packets. These devices are often called *middleboxes* **RFC 3234**. Some middleboxes operate only in the network layer, but most middleboxes are able to analyse the payload of the received packets and extract the transport header and in some cases the application layer protocols.

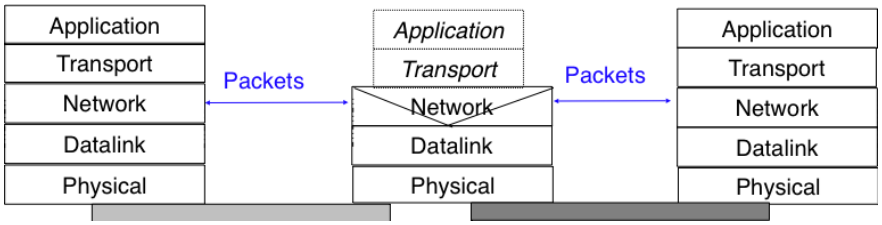


Figure 4.38: IP middleboxes and the reference model

In this section, we briefly describe two types of middleboxes : firewalls and network address translation (NAT) devices. A discussion of the different types of middleboxes with references may be found in **RFC 3234**.

Firewalls

When the Internet was only a research network interconnecting research labs, security was not a concern and most hosts agreed to exchange packets over TCP connections with most other hosts. However, as companies and more and more users became connected to Internet, allowing unlimited access to the hosts managed by companies was becoming a concern. Furthermore, at the end of the 1980s, several security issues affected the Internet such as the first Internet worm [RE1989] and some widely publicised security breaches [Stoll1988] [CB2003] [Cheswick1990]

These security problems convinced the industry that IP networks are a key part of the infrastructure of a company that should be protected by special devices like security guards and fences are used to protect buildings. These special devices were quickly called *firewalls*. A typical firewall has two interfaces :

- an external interface connected to the global Internet
- an internal interface connected to a trusted network

The first firewalls included configurable packet filters. A packet filter is a set of rules that define the security policy of a network. In practice, these rules are based on the values of fields of the IP or transport layer headers. Any field of the IP or transport header can be used in a firewall rule, but the most common ones :

- filter on the source address. For example, a company may decide to discard all packets received from one of its competitors. In this case, all packets whose source address belong to the competitor's address block would be rejected
- filter on destination address. For example, the hosts of the research lab of a company may receive packets from the global Internet, but not the hosts of the financial department
- filter on the *Protocol* number found in the IP header. For example, a company may only allow its hosts to use TCP or UDP, but not other more experimental transport protocols

- filter on the TCP or UDP port numbers. For example, only the DNS server of a company should received UDP segments whose destination port is set to *53* or only the official SMTP servers of the company can send TCP segments whose source ports are set to *25*
- filter on the TCP flags. For example, a simple solution to prohibit external hosts from opening TCP connections with hosts inside the company is to discard all TCP segments received from the external interface with only the *SYN* flag set.

Such firewalls are often called *stateless* firewalls because they do not maintain any state about the TCP connections that pass through them.

A second type of firewalls are the *stateful* firewalls. A stateful firewall tracks the state of each TCP connection passing through it. It maintains a TCB for each TCP connection. This TCB allows it to reassemble the received segments to extract their payload and perform verifications in the application layer. Some firewalls are able to inspect the URLs accessed by using HTTP and log all URLs visited or block TCP connections where a dangerous URL is exchanged. Some firewalls can verify that SMTP commands are used when a TCP connection is established on port *25* or that a TCP connection on port *80* carries HTTP commands and responses, ...

Beyond firewalls

Besides the firewalls, different types of "security" devices have been installed at the periphery of corporate networks. Intrusion Detection Systems (IDS) such as the popular snort are stateful devices that are capable of matching reassembled segments against regular expressions that correspond to signatures of viruses, worms or other types of attacks. Deep Packet Inspection (DPI) is another type of middlebox that analyse the packet's payload and possibly reassemble TCP segments to detect inappropriate usages. While IDS are mainly used in corporate networks, DPI is mainly used in Internet Service Providers. Some ISPs use DPI to detect and limit the bandwidth consumed by peer-to-peer applications. Some countries such as China or Iran use DPI to detect inappropriate Internet usage.

NAT

Network Address Translation (NAT) was proposed in [TE1993] and **RFC 3022** as a short term solution to deal with the expected shortage of IPv4 addresses in the late 1980s - early 1990s. Combined with CIDR, NAT allowed to significantly slow the consumption of IPv4 addresses. A NAT is a middlebox that interconnects two networks that are using IPv4 addresses from different addressing spaces. Usually, one of these addressing spaces is the public Internet while the other is using the private IPv4 addresses defined in **RFC 1918**.

A very common deployment of NAT is in broadband access routers as shown in the figure below. The broadband access router interconnects a home network, either WiFi or Ethernet based and the global Internet via one ISP over ADSL or CATV. A single IPv4 address is allocated to the broadband access router and network address translation allows all the hosts attached to the home network to share a single public IPv4 address.

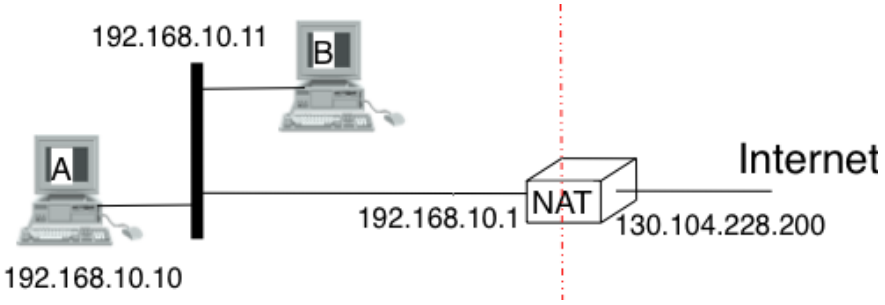


Figure 4.39: A simple NAT with one public IPv4 address

A second type of deployment is in enterprise networks as shown in the figure below. In this case, the NAT functionality is installed on a border router of the enterprise. A private IPv4 address is assigned to each enterprise host while the border router manages a pool containing several public IPv4 addresses.

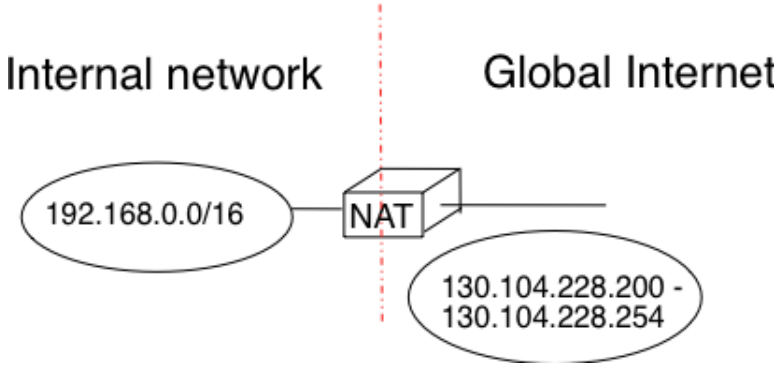


Figure 4.40: An enterprise NAT with several public IPv4 addresses

As the name implies, a NAT is a device that "translates" IP addresses. A NAT maintains a mapping table between the private IP addresses used in the internal network and the public IPv4 addresses. NAT allows a large number of hosts to share a pool of IP addresses because these hosts do not all access the global Internet at the same time.

The simplest NAT is a middlebox that uses a one-to-one mapping between a private IP address and a public IP address. To understand its operation, let us assume that a NAT such as the one shown above has booted. When the NAT receives a first packet from source *S* in the internal network destined to the public Internet, it creates a mapping between internal address *S* and the first address of its pool of public addresses (*P1*). Then it translates the received packet so that it can be sent to the public Internet. This translation is performed as followed :

- the source address of the packet (*S*) is replaced by the mapped public address (*P1*)
- the checksum of the IP header is incrementally updated as its content has changed
- if the packet carried a TCP or UDP segment, the transport layer checksum found of the included segment must also be updated as it is computed over the segment and a pseudo-header that includes the source and destination addresses

When a packet destined to *P1* is received from the public Internet, the NAT consults its mapping table to find *S*. The received packet is translated and forwarded in the internal network.

This works as long as the pool of public IP addresses of the NAT does not become empty. In this case, a mapping must be removed from the mapping table to allow a packet from a new host to be translated. This garbage collection can be implemented by adding to each entry in the mapping table a timestamp that contains the last utilisation time of a mapping entry. This timestamp is updated each time a the corresponding entry is used. Then, the garbage collection algorithm can remove the oldest mapping entry in the table.

A drawback of such as simple enterprise NAT is the size of the pool of public IPv4 addresses that is often too small to allow a large number of hosts to share such a NAT. In this case, a better solution is to allow the NAT to translate both IP addresses and port numbers.

Such a NAT maintains a mapping table that maps an internal IP address and TCP port number with an external IP address and TCP port number. When such a NAT receives a packet from the internal network, it performs a lookup in the mapping table with the packet's source IP address and source TCP port number. If a mapping is found, the source IP address and the source TCP port number of the packet are translated with the values found in the mapping table, the checksums are updated and the packet is sent to the global Internet. If no mapping is found, a new mapping is created with the first available couple *(IP address, TCP port number)* and the packet is translated. The entries of the mapping table are either removed at the end of the corresponding TCP connection is the NAT tracks TCP connection state like a stateful firewall or after some idle time.

When such a NAT receives a packet from the global Internet, it looks up its mapping table with the packet's destination IP address and destination TCP port number. If a mapping is found, the packet is translated and forwarded in the internal network. Otherwise, the packet is discarded as the NAT cannot determine to which particular internal host the packet should be forwarded. For this reason,

With 2^{16} different port numbers, a NAT may support a large number of hosts with a single public IPv4 address. However, it should be noted that some applications open a large number of TCP connections [Miyakawa2008]. Each of these TCP connections consumes one mapping entry in the NAT's mapping table. NAT allows many hosts to share one or a few public IPv4 addresses. However, using NAT has two important drawbacks. First, it is difficult for external hosts to open TCP connections with hosts that are behind a NAT. Some consider this to be a benefit from a security viewpoint. However, a NAT should not be confused with a firewall as there are some techniques to traverse NATs. Second, NAT breaks the end-to-end transparency of the network and transport layers. The main problem is when an application layer protocol uses IP addresses in some of the ADUs that it sends. A popular example is ftp defined in **RFC 959**. In this case, there is a mismatch between the packet header translated by the NAT and the packet payload. The only solution to solve this problem is to place on the NAT an Application Level Gateway (ALG) that understands the application layer protocol and can thus translate the IP addresses and port numbers found in the ADUs. However, defining an ALG for each application is costly and application developers should avoid using IP addresses in the messages exchanged in the application layer **RFC 3235**.

IPv6 and NAT

NAT has been very successful with IPv4. Given the size of the IPv6 addressing space, the IPv6 designers expected that NAT would never be useful with IPv6. The end-to-end transparency of IPv6 has been one of its key selling points compared to IPv4. However, recently the expected shortage of IPv4 addresses lead enterprise network administrators to consider IPv6 more seriously. One of the results of this analysis is that the IETF is considering the definition of NAT devices [WB2008] that are IPv6 specific. Another usage of NAT with IPv6 is to allow IPv6 hosts to access IPv4 destinations and conversely. The early IPv6 specifications included the Network Address Translation - Protocol Translation (NAT-PT) mechanism defined in **RFC 2766**. This mechanism was later deprecated in **RFC 4966** but has been recently restarted under the name NAT64 [BMvB2009]. A NAT64 is a middlebox that performs the IPv6<->IPv4 packet translation to allow IPv6 hosts to contact IPv4 servers.

4.3 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 30,000 different domains and this number is still growing [21]. A domain can be a small enterprise that manages a few routers in a single building, a larger enterprise with hundred routers at multiple locations or a large Internet Service Provider that manages thousands of routers. Two classes of routing protocols are used to allow these domains to efficiently exchange routing information.

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 the 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.

[21] 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.

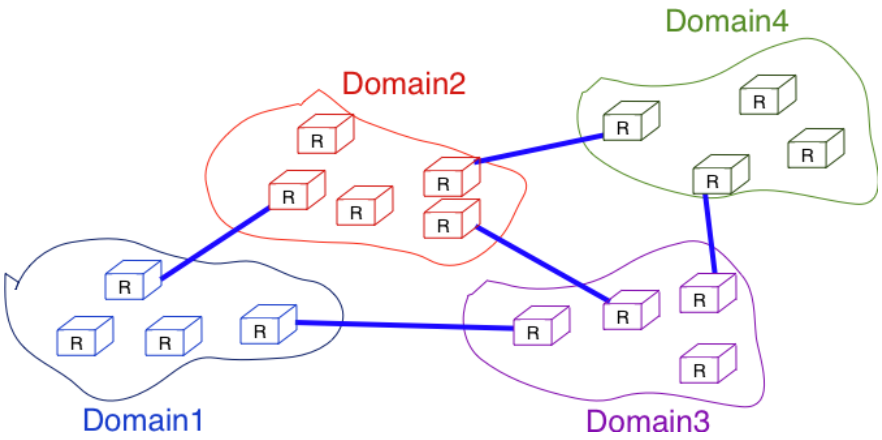


Figure 4.41: Organisation of a small Internet

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 blackbox.

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 hopsor the route with the highest bandwidth, ...

When we consider the interconnection of domains that are managed by different organisations, this is not true anymore. Each domain implements its own routing policy. A routing policy is composed of three elements : an *import 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*.

4.3.1 Intradomain routing

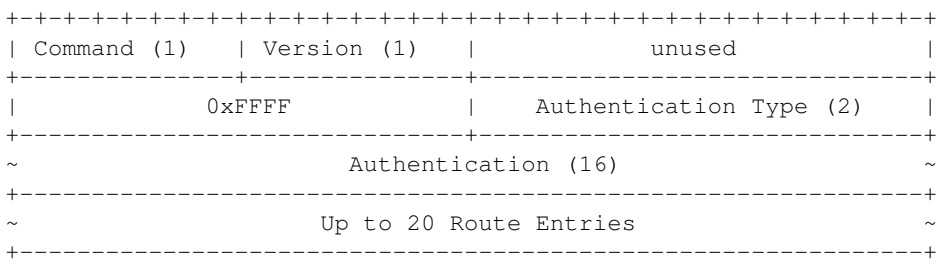
In this section, we briefly describe the key features of the two main intradomain unicast routing protocols : RIP and OSPF.

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 show 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. Routers send one of more RIP response messages every 30 seconds. These messages contain the distance vectors that summarize the router's routing table. The RIP requests messages can be used by routers or hosts to query other routers about the content of their routing table. A typical usage is when a router boots and wants to receive quickly the RIP responses from its neighbours to compute its own routing table. The current version of RIP is version 2 defined in **RFC 2453** for IPv4 and **RFC 2080** for IPv6.

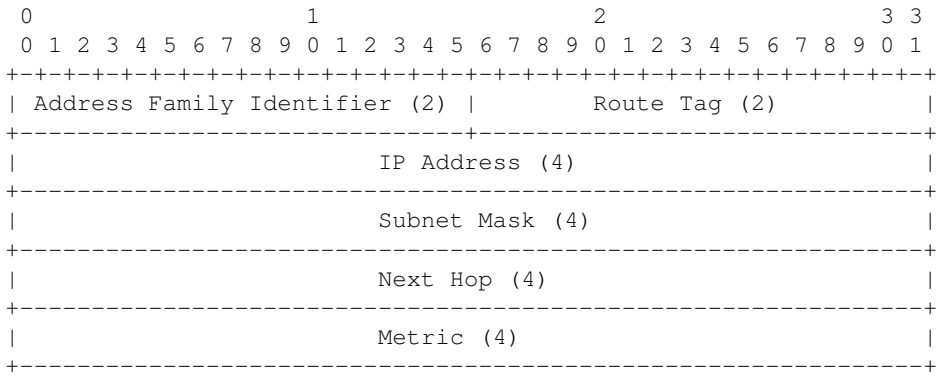
```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```



RIP message format

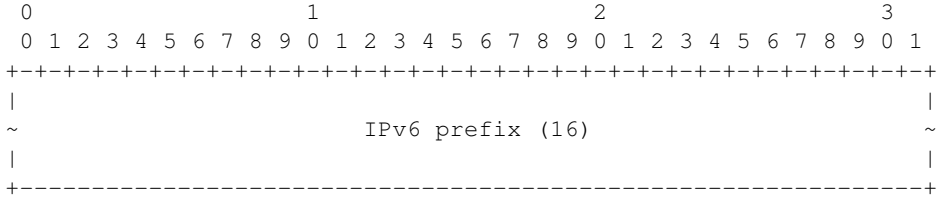
The RIP header contains an authentication field. This authentication can be used by network administrators to ensure that only the RIP messages sent by the routers that they manage are used to build the routing tables. **RFC 2453** only supports a basic authentication scheme where all routers are configured with the same password and include this password in all RIP messages. This is not very secure since an attacker can know the password by capturing a single RIP message. However, this password can protect against configuration errors. Stronger authentication schemes are described in **RFC 2082** and **RFC 4822**, but the details of these mechanisms are outside the scope of this section.

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 designed initially to be suitable for different network layer protocols. Some implementations of RIP were used in XNS or IPX networks. The first field of the RIP route entry is the *Address Family Identifier* (*AFI*). This identifier indicates the type of address found in the route entry [22]. IPv4 uses *AFI=1*. The other important fields of the route entry are the IPv4 prefix, the netmask that indicates the length of the subnet identifier and is encoded as a 32 bits netmask and the metric. Although the metric is encoded as a 32 bits field, the maximum RIP metric is *15* (for RIP, $16 = \infty$)



Format of the RIP IPv4 route entries (:rfc:`2453`)

With a 20 bytes route entry, it was difficult to use the same format as above to support IPv6. Instead of defining a variable length route entry format, the designers of **RFC 2080** defined a new format that does not include an *AFI* field. 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 is still *15*.



[22] The Address Family Identifiers are maintained by IANA at http://www.iana.org/assignments/address-family-numbers/

```
|         route tag (2)         | prefix len (1)|  metric (1)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Format of the RIP IPv6 route entries
```

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 synchronisations that occur with a fixed delay and is today a recommended practice for protocol designers.

OSPF

Two 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 that supports IPv6 is defined in **RFC 5340**. OSPF is frequently used in entreprise networks and in some ISP networks. However, ISP networks often use the IS-IS link-state routing protocol [ISO10589] that was developped 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 *linkstate*, there are some specificities of OSPF that are worth to be discussed. 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 in 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 in 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 [23] . 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 in 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 bits integer, that 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.

[23] OSPF can support *virtual links* to connect together routers that belong to the same area but are not directly connected. However, this goes beyond this introduction to OSPF.

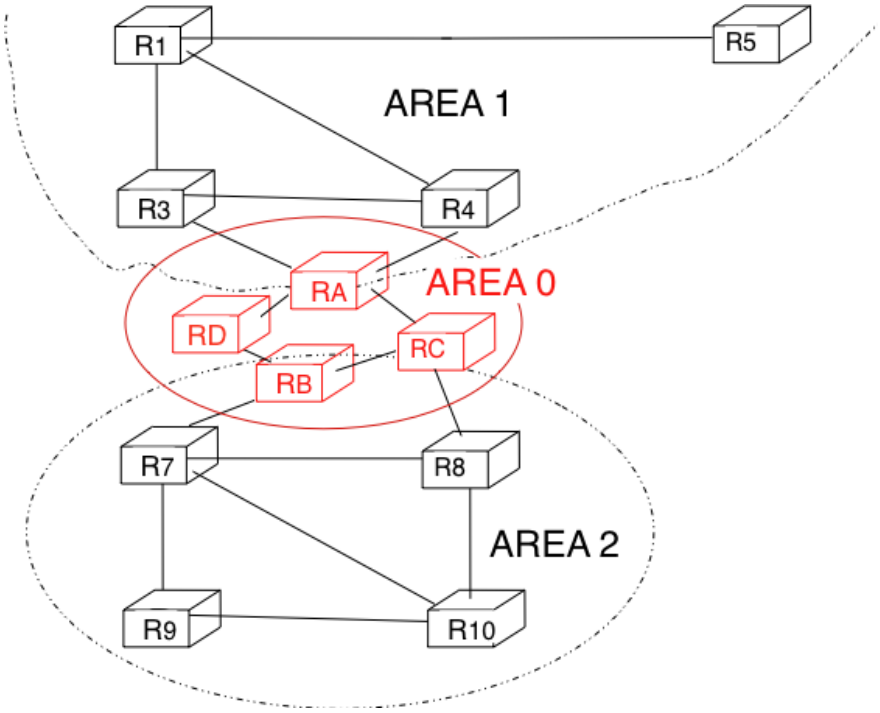


Figure 4.42: OSPF areas

Inside each non-backbone area, routers exchange link state packets to distribute the topology of the area to all routers of 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 exchange link-state packets for all the 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 of the area learn a route towards *192.168.1.0/24* and *192.168.10.0/24*. The two area border routers, *RB* and *RC*, create network summary advertisements. Assuming that all links have a unit link metric :

- *RB* advertises *192.168.1.0/24* at a distance of *2* and *192.168.10.0/24* at a distance of *3*
- *RC* advertises *192.168.1.0/24* at a distance of *3* and *192.168.10.0/24* at a distance of *2*

These summary advertisements are flooded in the backbone area attached to routers *RB* and *RC*. In its routing table, router *RA* selects the summary advertised by *RB* to reach *192.168.1.0/24* and the summary advertised by *RC* to reach *192.168.10.0/24*. Router *RA* advertises inside *area 1* a summary indicating that *192.168.1.0/24* and *192.168.10.0/24* are both at a distance of *3* from itself.

On the other hand, consider the prefixes *10.0.0.0/24* and *10.0.1.0/24* 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 :

- *10.0.0.0/24* at a distance of *1* and *10.0.1.0/24* at a distance of *2* from *RA*
- *10.0.0.0/23* 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 *10.0.0.0/24* and a route towards *10.0.1.0/24* that are both via router *RA*. The second advertisement allows to 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.

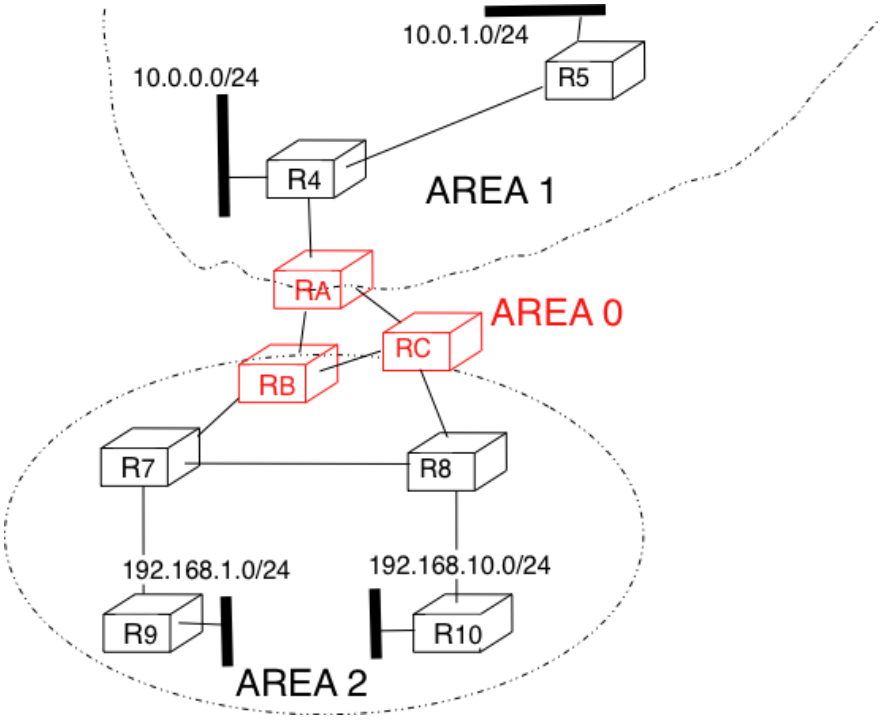


Figure 4.43: Hierarchical routing with OSPF

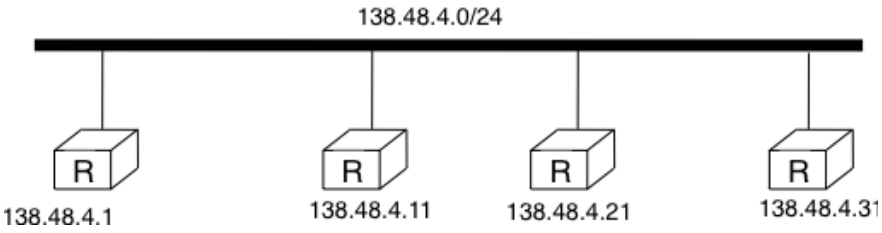


Figure 4.44: An OSPF LAN containing several routers

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 each router that 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 [24] one of them as the *Designated Router (DR)* **RFC 2328**. The *DR* router *represents* the local area network. It advertises the LAN's subnet (*138.48.4.0/24* in the example above). 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* as shown in the figure below.

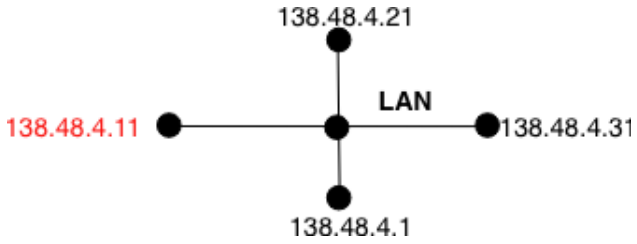


Figure 4.45: OSPF representation of a LAN

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] :

- 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 Bidirectionnal Forwarding Detection (BFD) protocol defined in [KW2009] that can be implemented directly on the router interfaces. Another solution 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

4.3.2 Interdomain routing

As explained earlier, the Internet is composed of more than 30,000 different networks [25] called *domains*. Each domain is composed of a group of routers and hosts that are managed by the same organisation. Example domains include

[24] 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.

[25] 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/

belnet, sprint, level3, geant, abilene cisco, google ... Each domain contains a set of routers. From a routing viewpoint, these domains can be divided in two classes : the *transit* and the *stub* domains. A *stub* domain sends and receive packets whose source or destination are one of its 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 destinations do not belong to the transit domain. As of this writing, there are about 85% of stub domains in the Internet [22]. 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.

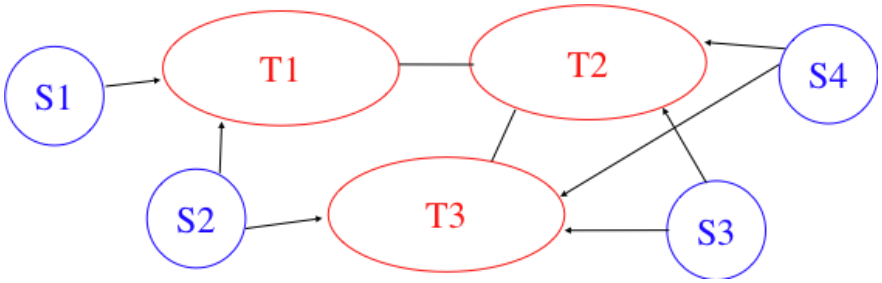


Figure 4.46: Transit and stub domains

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 During the last 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 viewpoint, 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.

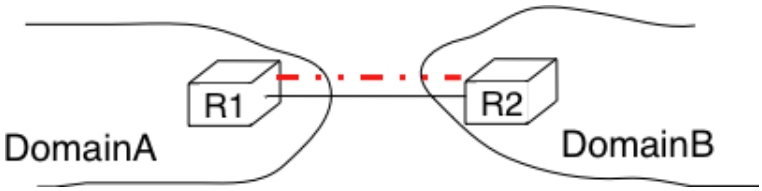


Figure 4.47: Interconnection of two domains via a private peering link

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 [26]. For some 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 domains 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 [27] to exchange packets, they simply use the Local Area Network. IXPs are very popular in Europe and many Internet

[26] See http://as-rank.caida.org/ for an analysis of the interconnections between domains based on measurements collected in the global Internet

[27] 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.

Service Providers and Content providers are present on these IXPs.

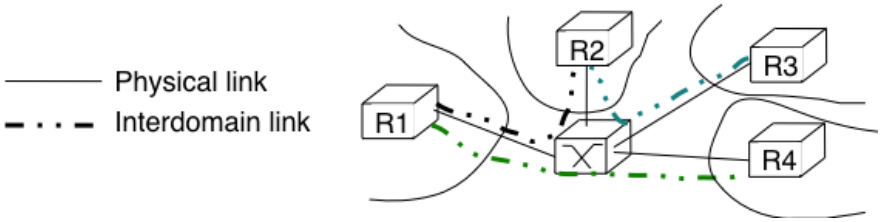


Figure 4.48: Interconnection of two domains at an Internet eXchange Point

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 not true anymore 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, ...) 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 on peering relationships between domains that are connected via a 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.

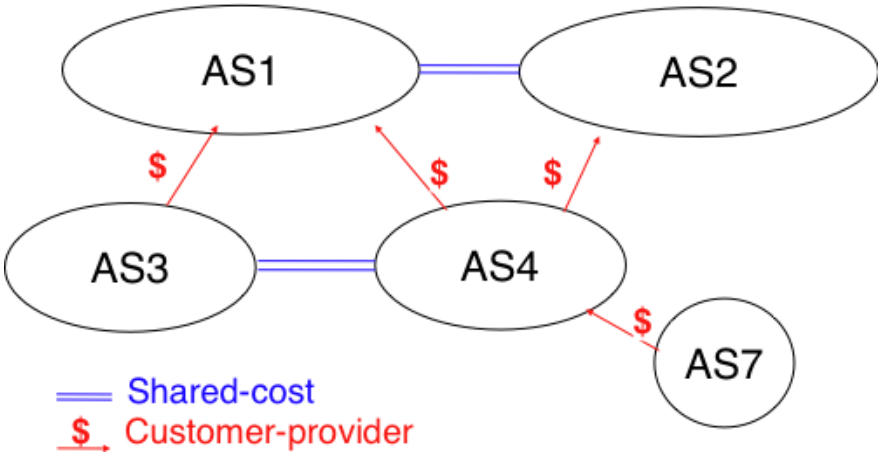


Figure 4.49: A simple Internet with peering relationships

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 know a route via *AS4* that allows them to reach hosts inside *AS7*. From a routing viewpoint, 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* and *AS3* would have to pay to send these packets to *AS1*.

From a routing viewpoint, 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 advertising over a *shared-cost* peering relationship that does not bring revenue the routes that it learned from a provider it would have allowed its *shared-cost* peer to use the link with its provider without any payment. If a domain were advertising over a *shared-cost* peering relationship the routes learned 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 neighboring 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 neighboring 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 [28] their import and export policies. Several tools allow 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 shows clearly 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 to *AS7* all the routes that it uses (including the routes learned from *AS3*).

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 viewpoint, each domain is identified by a

[28] See ftp://ftp.ripe.net/ripe/dbase for the RIPE database that contains the import and export policies of many European ISPs

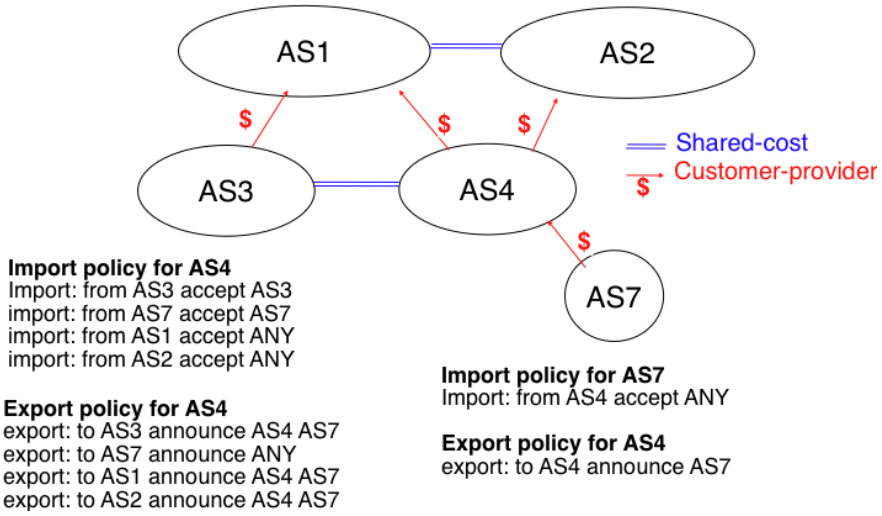


Figure 4.50: Import and export policies

unique *Autonomous System* (AS) number [29] 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 to its neighbours the routes that have changed.

The figure below shows a simple example of the BGP routes that are exchanged between domains. In this example, prefix *1.0.0.0/8* is announced by *AS1*. *AS1* advertises to *AS2* a BGP route towards this prefix. 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 in the AS-Path. *AS2* has learned two routes towards prefix *1.0.0.0/8*. 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 *1.0.0.0/8* with its AS-Path set to *AS2:AS4:AS1*. Thanks to the AS-Path, *AS5* knows that if it sends a packet towards *1.0.0.0/8* 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

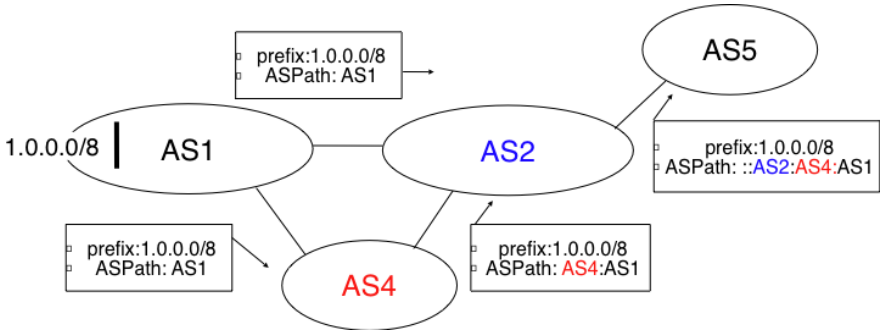


Figure 4.51: Simple exchange of BGP routes

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

[29] 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.

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, ... Furthermore, the two routers consider the peering link to be up as long as the BGP session and the underlying TCP connection remains up [30]. The two endpoints of a BGP session are called *BGP peers*.

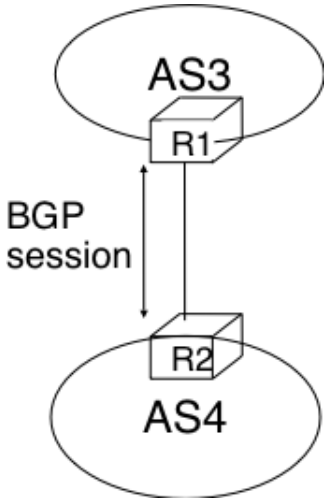


Figure 4.52: A BGP peering session between two directly connected routers

In practice, to establish a BGP session between routers *R1* on *R2* on 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 tries to regularly establish the BGP session with *R2*. *R2* only agrees to establish the BGP session with *R1* once it 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 to negotiate 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 [31], 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* message 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

[30] 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.

[31] 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.

allows BGP routers to efficiently exchange such information while minimising the number of bytes exchanged. Each *UPDATE* message contains :

- 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 by 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 viewpoint, a BGP router connected to *N* BGP peers, can be described as being composed of four parts as shown in the figure below.

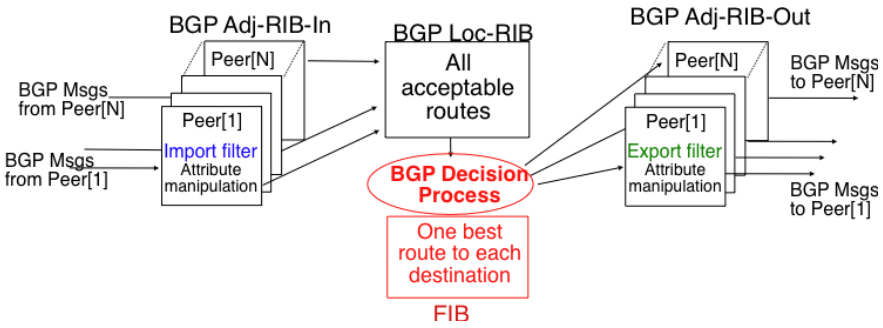


Figure 4.53: Organisation of a BGP router

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

```
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*/
foreach (destination=d inside BGP Loc-RIB)
{
 B=build_BGP_UPDATE(d); // best path
 S=apply_export_filter(RemoteAS,B);
 if (S<>NULL)
       { /* send UPDATE message */
        send_UPDATE(S,RemoteAS, RemoteIP)
       }
 }
/* entire RIB has been sent */
/* new UPDATE will be sent only to reflect local or distant
  changes in routes */
...
}
```

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

```
BGPMsg Apply_export_filter(RemoteAS, BGPMsg)
{ /* check if Remote AS already received route */
if (RemoteAS isin BGPMsg.ASPath)
  BGPMsg==NULL;
 /* Many additional export policies can be configured : */
 /* Accept or refuse the BGPMsg */
 /* Modify selected attributes inside 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.

```
Recvd_BGPMsg(Msg, RemoteAS)
{
 B=apply_import_filer(Msg,RemoteAS);
 if (B==NULL) /* Msg not acceptable */
       exit();
 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<>NULL) /* announce best route */
       send_UPDATE(S,RemoteAS);
   else if (Old_Route<>NULL)
    send_WITHDRAW(Msg.prefix);
  }
 if IsWITHDRAW(Msg)
 {
  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(d);
    S=apply_export_filter(RemoteAS,B);
    if (S<>NULL) /* still one best route */
      send_UPDATE(S,RemoteAS, RemoteIP);
    else if(Old_Route<>NULL)/* 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]

```
BGPMsg apply_import_filter(RemoteAS, BGPMsg)
{ /* check that we are not already inside  ASPath */
 if (MyAS isin BGPMsg.ASPath)
  BGPMsg==NULL;
 /* Many additional import policies can be configured : */
 /* Accept or refuse the BGPMsg */
 /* Modify selected attributes inside BGPMsg */
}
```

The bogon filters

Another example of a 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 private IPv4 prefixes defined in **RFC 1918**, the loopback prefixes (*127.0.0.1/8* and *::1/128*) or the IP 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 at http://www.team-cymru.org/Services/Bogons/

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 details the operation of BGP in an IPv4 network. For this, let us consider the simple network composed of three routers located in three different ASes and shown in the figure below.

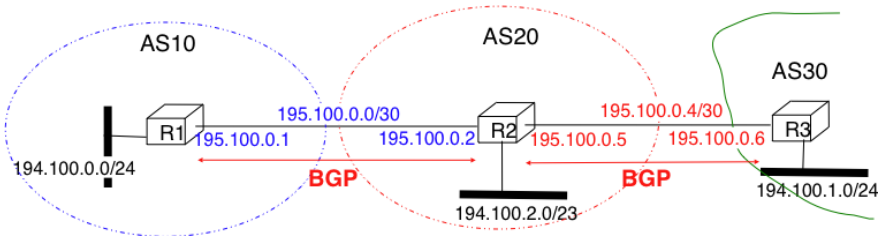


Figure 4.54: Utilisation of the BGP nexthop attribute

This network contains three routers : *R1*, *R2* and *R3*. Each router is attached to a local IPv4 subnet that it advertises by using BGP. There are two BGP sessions, one between *R1* and *R2* and the second between *R2* and *R3*. A */30* subnet is used on each interdomain link (*195.100.0.0/30* on *R1-R2* and *195.100.0.4/30* on *R2-R3*). The BGP sessions run above TCP connections established between the neighbouring routers (e.g. *195.100.0.1 - 195.100.0.2* 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(194.100.0.0/24,195.100.0.1,AS10)* to *R2* and *R2* sends *U(194.100.2.0/23,195.100.0.2,AS20)*. At this point, *R1* can reach *194.100.2.0/23* via *195.100.0.2* and *R2* can reach *194.100.0.0/24* via *195.100.0.1*.

Once the *R2-R3* has been established, *R3* sends *U(194.100.1.0/24,195.100.0.6,AS30)*. *R2* announces on the *R2-R3* session all the routes inside its RIB. It thus sends to *R3* : *U(194.100.0.0/24,195.100.0.5,AS20:AS10)* and *U(194.100.2.0/23,195.100.0.5,AS20)*. Note that when *R2* advertises the route that it learned from *R1*, it updates the BGP nexthop and adds it AS number in the AS-Path. *R2* also sends *U(194.100.1.0/24,195.100.0.2,AS20:AS30)* to *R1* on the *R1-R3* session. At this point, all BGP routes have been exchanged and all routers can reach *194.100.0.0/24*, *194.100.2.0/23* and *194.100.1.0/24*.

If the link between *R2* and *R3* fails, *R3* detects the failure because 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(194.100.1.0/24)* to *R1* over the *R1-R3* BGP session since it does not have a route anymore towards this prefix.

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 *130.104.0.0/16* IPv4 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 as if the route flaps [a], this w cause a large number of BGP messages being exchanged in the global Internet.

[a] A link is said to be flapping if it switches several between an operationnal 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.

Most networks that use BGP contain more than one router. For example, consider the network shown in the figure below where *AS20* contains two routers attached to interdomain links : *R2* and *R4*. In this network, two routing protocols are used by *R2* and *R4*. They use an intradomain routing protocol such as OSPF to distribute the routes towards the internal prefixes : *195.100.0.8/30*, *195.100.0.0/30*, ... *R2* and *R4* also use BGP. *R2* receives the routes advertised by *AS10* while *R4* receives the routes advertised by *AS30*. These two routers need to exchange the routes that they have respectively received over their BGP sessions.

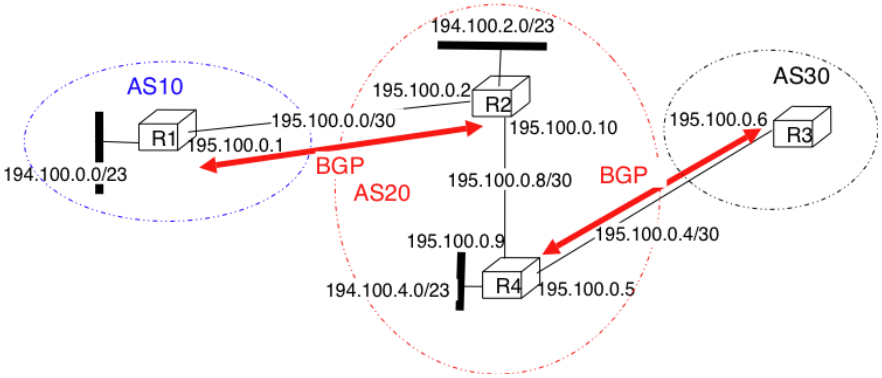


Figure 4.55: A larger network using BGP

A first solution to allow *R2* and *R3* to exchange the interdomain routes that they have learned over their respective BGP sessions would be to configure the intradomain routing protocol to distribute inside *AS20* the routes learned over the BGP sessions. Although current routers support this feature, this is a bad solution for two reasons :

1. Intradomain routing protocols cannot distribute the attributes that are attached to a BGP route. If *R4* received via the intradomain routing protocol a route towards *194.100.0.0/23* that *R2* learned via BGP, it would not know that the route was originated by *AS10* and the only advertisement that it could send to *R3* would contain an incorrect AS-Path
2. Intradomain routing protocols have not been designed to support the hundreds of thousands routes that a BGP router can receive on today's global Internet.

The best solution to allow BGP routers to distribute, inside an AS, all the routes learned over BGP sessions is to establish BGP sessions among all the BGP routers inside the AS. In practice, there are two types of BGP sessions :

- *eBGP session* or *external BGP session*. Such a BGP session is established between two routers that are directly connected and belong to two different domains.

- *iBGP session* or *internal BGP session*. Such a BGP session is established between two routers belonging to the same domain. These two routers do not need to be directly connected.

In practice, each BGP router inside a domain maintains an *iBGP session* with each other BGP router in the domain [32]. This creates a full-mesh of *iBGP sessions* among all BGP routers of the domain. *iBGP sessions*, like *eBGP sessions* run over TCP connections. Note that in contrast with *eBGP sessions* that are established between directly connected routers, *iBGP sessions* are often established between routers that are not directly connected.

An important point to note about *iBGP sessions* is that a BGP router only advertises a route over an *iBGP session* provided that :

- the router uses this route to forward packets, and
- the route was learned over one of the router's *eBGP sessions*

A BGP router does not advertise over an *iBGP session* a route that it has learned over another *iBGP session*. Note that a router can, of course, advertise over an *eBGP session* a route that it has learned over an *iBGP session*. This difference between the behaviour of a BGP router over *iBGP* and *eBGP* session is due to the utilisation of a full-mesh of *iBGP sessions*. Consider a network containing three BGP routers : *A*, *B* and *C* interconnected via a full-mesh of iBGP sessions. If router *A* learns a route towards prefix *p* from router *B*, router *A* does not need to advertise the received route to router *C* since router *C* also learns the same route over the *C-B iBGP session*.

To understand the utilisation of an *iBGP session*, let us consider in the network shown below what happens when router *R1* sends *U(194.100.0.0/23,195.100.0.1,AS10)*. This BGP message is processed by *R2* that advertises it over its *iBGP session* with *R4*. The *BGP Update* sent by *R2* contains the same nexthop and the same AS-Path as in the *BGP Update* received by *R2*. *R4* then sends *U(194.100.0.0/23,195.100.0.5,AS20:AS10)* to *R3*. Note that the BGP nexthop and the AS-Path are only updated [33] when a BGP route is advertised over an *eBGP session*.

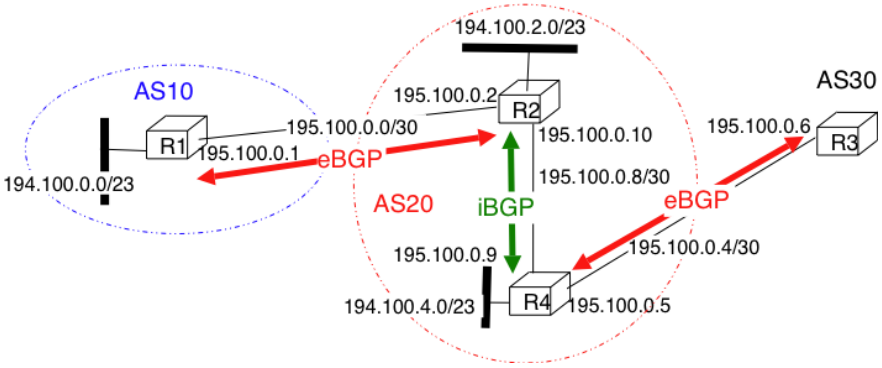


Figure 4.56: iBGP and eBGP sessions

[32] Using a full-mesh of iBGP sessions is suitable in small networks. However, this solution does not scale in large networks containing hundreds or more routers since $\frac{n \times (n-1)}{2}$ iBGP sessions must be established in a domain containing n BGP routers. Large domains use either Route Reflection **RFC 4456** or confederations **RFC 5065** to scale their iBGP, but this goes beyond this introduction.

[33] Some routers, when they receive a *BGP Update* over an *eBGP session*, set the nexthop of the received route to one of their own addresses. This is called *nexthop-self*. See e.g. [WMS2004] for additional details.

Loopback interfaces and iBGP sessions

In addition to their physical interfaces, routers can also be configured with a special loopback interface. A loopback interface is a software interface that is always up. When a loopback interface is configured on a router, the address associated to this interface is advertised by the intradomain routing protocol inside the domain. Thus, the IP address associated to a loopback interface is always reachable while an IP address associated to a physical interface becomes unreachable as soon as the corresponding physical interface fails. *iBGP sessions* are usually established by using the router's loopback addresses as endpoints. This allows the *iBGP session* and its underlying TCP connection to remain up even if physical interfaces fail on the routers.

Now that routers can learn interdomain routes over iBGP and eBGP sessions, let us examine what happens when router *R3* sends a packet destined to *194.100.1.234*. *R3* forwards this packet to *R4*. *R4* uses an intradomain routing protocol and BGP. Its BGP routing table contains the following longest prefix match :

- *194.100.0.0/23* via *195.100.0.1*

This routes indicates that to forward a packet towards *194.100.0.0/23*, *R4* needs to forward the packet along the route towards *195.100.0.1*. However, *R4* is not directly connected to *195.100.0.1*. *R4* learned a route that matches this address thanks to its intradomain routing protocol that distributed the following routes :

- *195.100.0.0/30* via *195.100.0.10*
- *195.100.0.4/30* East
- *195.100.0.8/30* North
- *194.100.2.0/23* via *195.100.0.10*
- *194.100.0.4/23* West

To build its forwarding table, *R4* must combine the routes learned from the intradomain routing protocol with the routes learned from BGP. Thanks to its intradomain routing table, *R4* replaces for each interdomain route the BGP nexthop with its shortest path computed by the intradomain routing protocol. In the figure above, *R4* forwards packets to *194.100.0.0/23* via *195.100.0.10* to which it is directly connected via its North interface. *R4* resulting forwarding table, that associates an outgoing interface for a directly connected prefix or a directly connected nexthop and an outgoing interface for prefixes learned via BGP, is shown below :

- *194.100.0.0/23* via *195.100.0.10* (North)
- *195.100.0.0/30* via *195.100.0.10* (North)
- *195.100.0.4/30* East
- *195.100.0.8/30* North
- *194.100.2.0/23* via *195.100.0.10* (North)
- *194.100.4.0/23* West

There is thus a coupling between the interdomain and the intradomain routing tables. If the intradomain routes change, e.g. due to link failures or changes in link metrics, then the forwarding table must be updated on each router as the shortest path towards a BGP nexthop may have changed.

The last point to be discussed before looking at the BGP decision process is that a network may contain routers that do not maintain any eBGP session. These routers can be stub routers attached to a single router in the network or core routers that reside on the path between two border routers that are using BGP as illustrated in the figure below.

In the scenario above, router *R2* needs to be able to forward a packet towards any destination in the *12.0.0.0/8* prefix inside *AS30*. Such a packet would need to be forwarded by router *R5* since this router resides on the path between *R2* and its BGP nexthop attached to *R4*. Two solutions can be used to ensure that *R2* is able to forward such interdomain packets :

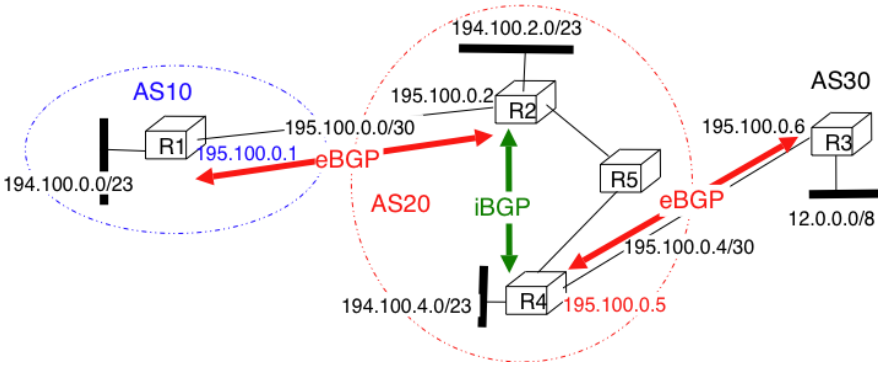


Figure 4.57: How to deal with non-BGP routers ?

- enable BGP on router *R5* and include this router in the *iBGP* full-mesh. Two iBGP sessions would be added in the figure above : *R2-R5* and *R4-R5*. This solution works and is used by many ASes. However, it forces all routers to have enough resources (CPU and memory) to run BGP and maintain a large forwarding table
- encapsulate the interdomain packets sent through the AS so that router *R5* never needs to forward a packet whose destination is outside the local AS. Different encapsulation mechanisms exist. MultiProtocol Label Switching (MPLS) **RFC 3031** and the Layer 2 Tunneling Protocol (L2TP) **RFC 3931** are frequently used in large domains, but a detailed explanation of these techniques is outside the scope of this section. The simplest encapsulation scheme to understand is in IP in IP defined in **RFC 2003**. This encapsulation scheme places an IP packet (called the inner packet), including its payload, as the payload of a larger IP packet (called the outer packet). It can be used by border routers to forward packets via routers that do not maintain a BGP routing table. For example, in the figure above, if router *R2* needs to forward a packet towards destination *12.0.0.1*, it can add at the front of this packet an IPv4 header whose source address is set to one of its IPv4 addresses and whose destination address is one of the IPv4 addresses of *R4*. The *Protocol* field of the IP header is set to *4* to indicate that it contains an IPv4 packet. The packet is forwarded by *R5* to *R4* based on the forwarding table that it built thanks to its intradomain routing table. Upon reception of the packet, *R4* removes the outer header and consult its (BGP) forwarding table to forward the packet towards *R3*.

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. This attribute is always present inside *BGP Updates* exchanged over *iBGP sessions*, but never present in the messages exchanged over *eBGP sessions*.

A common utilisation of *local-pref* is to support backup links. Consider the situation depicted in the figure below. *AS1* would like to always use the high bandwidth link to send an receive packets via *AS2* and only use the backup link upon failure of the primary one.

As BGP routers always prefer the routes having the highest *local-pref* attribute, this policy can be implemented by using the following import filter on *R1*:

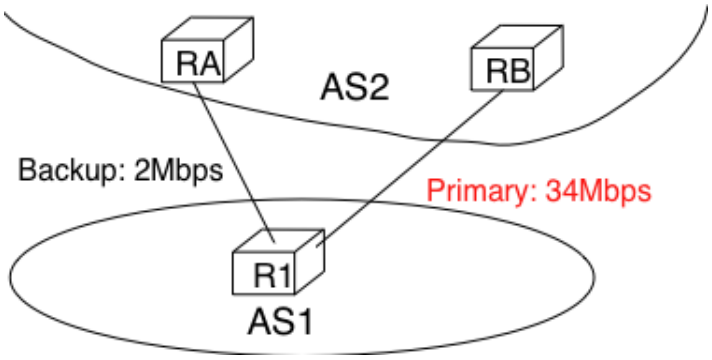


Figure 4.58: How to create a backup link with BGP ?

```
import: from  AS2 RA at R1 set localpref=100;
        from  AS2 RB at R1 set localpref=200;
        accept ANY
```

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*.

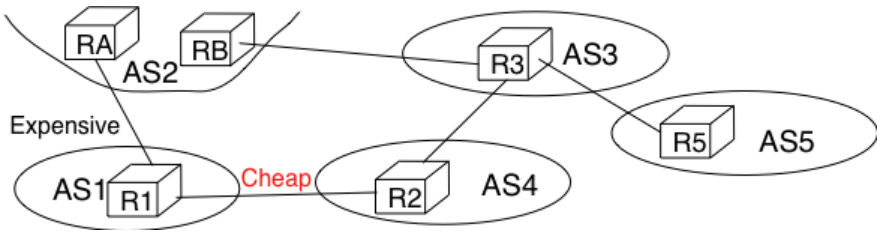


Figure 4.59: How to prefer a cheap link over an more expensive one ?

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 viewpoint, 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 to 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 viewpoint. 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 this setting of the *local-pref* attribute is that Internet paths are often assymetrical. Consider for example the internetwork shown in the figure below.

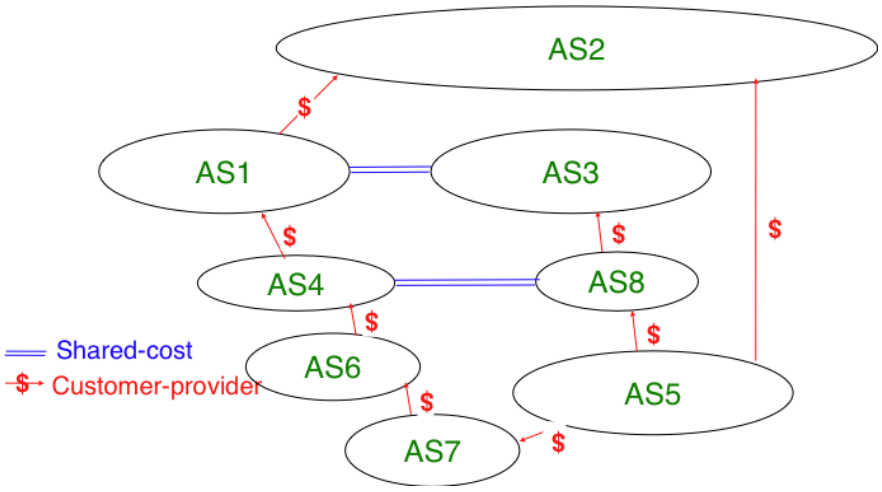


Figure 4.60: Assymetry of Internet paths

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 from learned from a customer. *AS5* on the other hand receives three paths towards *AS1* via its providers. I may select any of these paths to reach *AS1* depending on how it prefers one provider over the others.

Coming back to the organisation of a BGP router shown in the figure *Organisation of a BGP router*, the last part to be discussed is the BGP decision process. The *BGP Decision Process* is the algorithm used by routers to select the route to be installed in the FIB when there are multiple routes towards the same prefix. The BGP decision process receives a set of candidate routes towards the same prefix and uses seven steps. At each step, some routes are removed from the candidate set and the process stops when the set contains only one route [34] :

1. Ignore routes having an unreachable BGP nexthop
2. Prefer routes having the highest local-pref
3. Prefer routes having the shortest AS-Path

[34] Some BGP implementations can be configured to install several routes towards a single prefix in their FIB for load-balancing purposes. However, this goes beyond this introduction to BGP.

4. Prefer routes having the smallest MED
5. Prefer routes learned via eBGP sessions over routes learned via iBGP sessions
6. Prefer routes having the closest next-hop
7. Tie breaking rules : prefer routes learned from the router with lowest router id

The first step of the BGP decision process ensures that a BGP router does not install in its FIB a route whose nexthop is considered to be unreachable by the intradomain routing protocol. This could happen for example when a router has crashed. The intradomain routing protocol usually advertises the failure of this router before the failure of the BGP sessions that it terminates. This rule implies that the BGP decision process must be re-run each time the intradomain routing protocol reports a change in the reachability of a prefix containing one of more BGP nexthops.

The second rule allows each domain to define its routing preferences. The *local-pref* attribute is set by the import filter of the router that learned a route over an eBGP session.

In contrast with intradomain routing protocols, BGP does not contain an explicit metric. This is because in the global Internet it is impossible for all domains to agree on a common metric that meets the requirements of all domains. Despite of this, BGP routers prefer routes having a short AS-Path attribute over routes with a long AS-Path. This step of the BGP decision process is motivated by the fact that operators expect that a route with a long AS-Path has a lower quality than a route with a shorter AS-Path. However studies have shown that there was not always a strong correlation between the quality of a route and the length of its AS-Path [HFPMC2002]. Before explaining the fourth step of the BGP decision process, let us first describe the fifth and the sixth steps of the BGP decision process. These two steps are used to implement *hot potato* routing. Intuitively, when a domain implements *hot potato routing*, it tries to forward as quickly as possible to other domains packets that are destined to addresses outside of its domain.

To understand *hot potato routing*, let us consider the two domains shown in the figure below. *AS2* advertises prefix *1.0.0.0/8* over the *R2-R6* and *R3-R7* peering links. The routers inside *AS1* learn two routes towards *1.0.0.0/8*: one via *R6-R2* and the second via *R7-R3*.

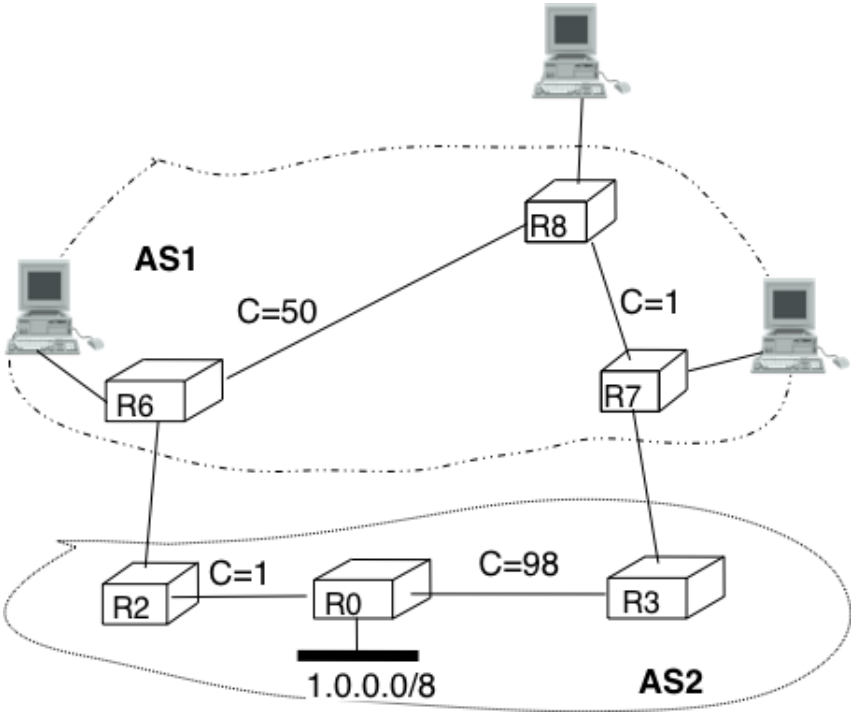


Figure 4.61: Hot and cold potato routing

With the fifth step of the BGP decision process, a router always prefers to use a route learned over an *eBGP session*

compared to a route learned over an *iBGP session*. Thus, router *R6* (resp. *R7*) prefers to use the route via router *R2* (resp. *R3*) to reach prefix *1.0.0.0/8*.

The sixth step of the BGP decision process takes into account the distance, measured as the length of the shortest intradomain path, between a BGP router and the BGP nexthop for routes learned over *iBGP sessions*. This rule is used on router *R8* in the example above. This router has received two routes towards *1.0.0.0/8*:

- *1.0.0.0/8* via *R7* that is at a distance of *1* from *R8*
- *1.0.0.0/8* via *R6* that is at a distance of *50* from *R8*

The first route, via *R7* is the one that router *R8* prefers as this is the route that minimises the cost of forwarding packets inside *AS1* before sending them to *AS2*.

Hot potato routing allows *AS1* to minimise the cost of forwarding packets towards *AS2*. However, there are situations where this is not desirable. For example, assume that *AS1* and *AS2* are domains with routers on both the East and the West coast of the US. In these two domains, the high metric associated to links *R6-R8* and *R0-R2* correspond to the cost of forwarding a packet across the USA. If *AS2* is a customer that pays *AS1*, it would prefer to received the packets destined to *1.0.0.0/8* via the *R2-R6* link instead of the *R7-R3* link. This is the objective of *cold potato routing*. *Cold potato routing* is implemented by using the *Multi-Exit Discriminator (MED)* attribute. This attribute is an optional BGP attribute that may be set [35] by border routers when advertising a BGP route over an *eBGP session*. The MED attribute is usually used to indicate over an *eBGP session* the cost to reach the BGP nexthop for the advertised route. The *MED* attribute is set by the router that advertises a route over an *eBGP session*. In the example above, router *R2* sends *U(1.0.0.0/8,R2,AS2,MED=1)* while *R3* sends *U(1.0.0.0/8,R3,AS2,MED=98)*.

Assume that the BGP session *R7-3* is the first to be established. *R7* sends *U(1.0.0.0/8,R3,AS2,MED=98)* to both *R8* and *R6*. At this point, all routers inside *AS1* send the packets towards *1.0.0.0/8* via *R7-R3*. Then, the *R6-R2* BGP session is established and router *R6* receives *U(1.0.0.0/8,R2,AS2,MED=1)*. Router *R6* runs its decision process for destination *1.0.0.0/8* and selects the route via *R2* as its chosen route to reach this prefix since this is the only route that it knows. *R6* sends *U(1.0.0.0/8,R2,AS2,MED=1)* to routers *R8* and *R7*. They both run their decision prefer and prefer the route advertised by *R6* as it contains the smallest *MED*. Now, all routers inside *AS1* forward the packets to *1.0.0.0/8* via link *R6-R2* as expected by *AS2*. As router *R7* does not anymore use the BGP route learned via *R3* it must stop advertising it over *iBGP sessions* and sends *W(1.0.0.0/8)* over its *iBGP sessions* with *R6* and *R8*. However, router *R7* still keeps the route learned from *R3* inside its Adj-RIB-In. If the *R6-R2* link fails, *R6* sends *W(1.0.0.0/8)* over its iBGP sessions and router *R7* responds by sending *U(1.0.0.0/8,R3,AS2,MED=98)* over its iBGP sessions.

In practice, the fifth step of the BGP decision process is slightly more complex because the routes towards a given prefix can be learned from different ASes. For example, assume that in figure *Hot and cold potato routing*, *1.0.0.0/8* is also advertised by *AS3* (not shown in the figure) that has peering links with routers *R6* and *R8*. If *AS3* advertises a route whose MED attribute is set to *2* and another with a MED set to *3*, how should *AS1*'s router compare the four BGP routes towards *1.0.0.0/8* ? Is a MED value of *1* from *AS2* better than a MED value of *2* from *AS3* ? The fifth step of the BGP decision process solves this problem by only comparing the MED attribute of the routes learned from the same neighbour AS. Additional details about the MED attribute may be found in **RFC 4451**. It should be noted that using the MED attribute may cause some problems in BGP networks as explained in [GW2002]. In practice, the *MED* attribute is not used on *eBGP sessions* unless the two domains agree to enable it.

The last step of the BGP decision allows to select a single route when a BGP router has received several routes that are considered as equal by the first six steps of the decision process. This can happen for example in a dual-homed stub attached to two different providers. As shown in the figure below, router *R1* receives two equally good BGP routes towards *1.0.0.0/8*. To break the ties, each router is identified by a unique *router-id* which in practice is one of the IP addresses assigned to the router. On some routers, the lowest router id step in the BGP decision process is replaced by the selection of the oldest route **RFC 5004**. Preferring the oldest route when breaking ties is used to prefer stable paths over unstable paths. However, a drawback of this approach is that the selection of the BGP routes depends on the arrival times of the corresponding messages. This makes the BGP selection process non-deterministic and can lead to problems that are difficult to debug.

[35] The MED attribute can be used on *customer->provider* peering relationships upon request of the customer. On *shared-cost* peering relationship, the MED attribute is only enabled when there is a explicit agreement between the two peers.

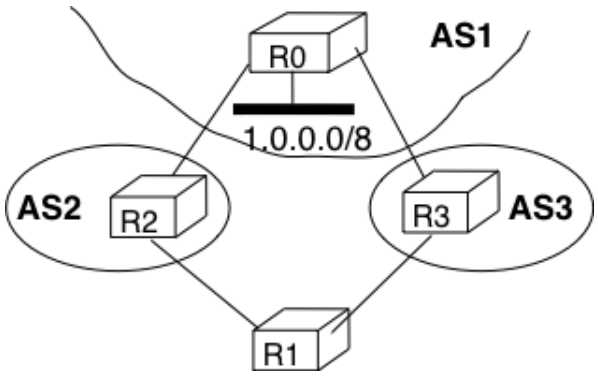


Figure 4.62: A stub connected to two providers

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.

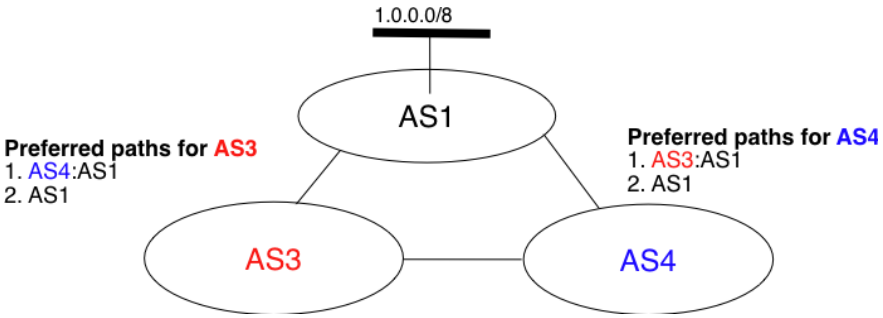


Figure 4.63: The disagree internetwork

In this internetwork, we focus on the route towards *1.0.0.0/8* that is advertised by *AS1*. Let us also assume that *AS3* (resp. *AS4*) prefers, e.g. for economical reasons, a route learned from *AS4* (*AS3*) over a route learned from *AS1*. When *AS1* sends *U(1.0.0.0/8,AS1)* to *AS3* and *AS4*, three sequences of exchanges of BGP messages are possible :

1. *AS3* sends first *U(1.0.0.0/8,AS3:AS1)* to *AS4*. *AS4* has learned two routes towards *1.0.0.0/8*. It runs its BGP decision process and selects the route via *AS3* and does not advertise a route to *AS3*
2. *AS4* sends first *U(1.0.0.0/8,AS3:AS1)* to *AS3*. *AS3* has learned two routes towards *1.0.0.0/8*. It runs its BGP decision process and selects the route via *AS4* and does not advertise a route to *AS4*
3. *AS3* sends *U(1.0.0.0/8,AS3:AS1)* to *AS4* and, at the same time, *AS4* sends *U(1.0.0.0/8,AS4:AS1)*. *AS3* prefers the route via *AS4* and thus sends *W(1.0.0.0/8)* to *AS4*. In the mean time, *AS4* prefers the route via *AS3* and thus sends *W(1.0.0.0/8)* to *AS3*. Upon reception of the *BGP Withdraws*, *AS3* and *AS4* only know the direct route towards *1.0.0.0/8*. *AS3* (resp. *AS4*) sends *U(1.0.0.0/8,AS3:AS1)* (resp. *U(1.0.0.0/8,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 operationnal viewpoint, 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 that is often named *Bad Gadget* [GW1999]

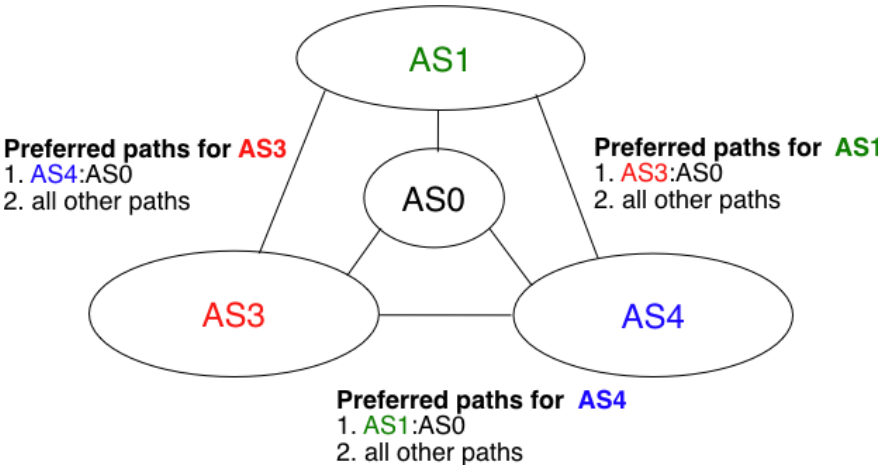


Figure 4.64: The bad gadget internetwork

In this internetwork, there are four ASes. *AS0* advertises one route toward 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 know 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 operationnal 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 as relationship would not make sense from an economical viewpoint as it would imply circular payments. Furthermore, providers are usually larger than customers.

The second guideline also corresponds to economical 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 viewpoint, 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 [36] in most domains. Thanks to the large amount of BGP data that has been collected by operators and researchers [37], 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. 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 entreprise 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.

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.

[36] Some 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 publically available model that goes beyond these classical peering relationships.

[37] 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/

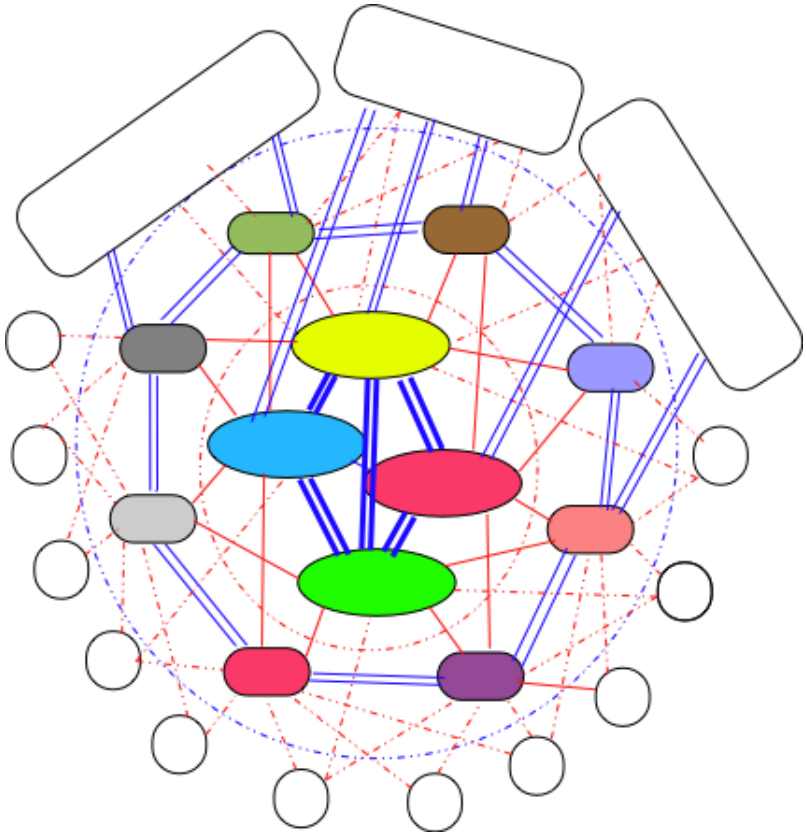


Figure 4.65: The layered structure of the global Internet

CHAPTER

FIVE

THE DATALINK LAYER AND THE LOCAL AREA NETWORKS

The datalink layer is the lowest layer of the reference model that we discuss in details. As mentioned previously, there are two type of datalink layers. The first datalink layers that appeared are the ones that are used on point-to-point links between endsystems that are directly connected by a physical link. We will briefly discuss one of these datalink layers in this chapter. The second type of datalink layers are the ones used in Local Area Networks. The main difference between the point-to-point and the LAN datalink layers is that the latter need to regulate the access to the Local Area Network which is usually a shared medium. This chapter is organised as follows. We first discuss the principles and of datalink layer and the service that it uses from the physical layer. Then we describe in more details several Medium Access Control algorithms that are used by Local Area Networks to regulate the access to the shared medium. Finally we discuss in details several important datalink layer technologies with an emphasis on Ethernet.

5.1 Principles

The datalink resides above and uses the service provided by the physical layer. Although there are many different implementations of the physical layer from a technological viewpoint, they all provide a service that enables the datalink layer to send and receive bits to/from another directly connected endsystem. The datlink layer receives packets from the network layer. Two datalink layer entities exchange *frames*. As explained in the previous chapter, most datalink layer technologies impose limitations on the size of the frames. Some technologies impose only a maximum frame size, others enforce both minimum and maximum frames sizes and finally some technologies only support a single frame size. In the latter case, the datalink layer will usually include an adaptation sublayer to allow the network layer to send and receive variable-length packets. This adaptation layer may include fragmentation and reassembly mechanisms.

The physical layer service allows to send and receive bits. Compared to the requirements of the applications, it is usually far from imperfect as explained in the introduction :

- 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

The datalink layer must be able to allow endsystems to exchange frames containing packets despite of these limitations. On point-to-point links and Local Area Networks, 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 was perfect, the problem would be very simple. The datalinke layer would simply need to define how to encode each frame as a sequence of consecutive bits. The receiver would then be able to easily 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 datalink layer technologies.

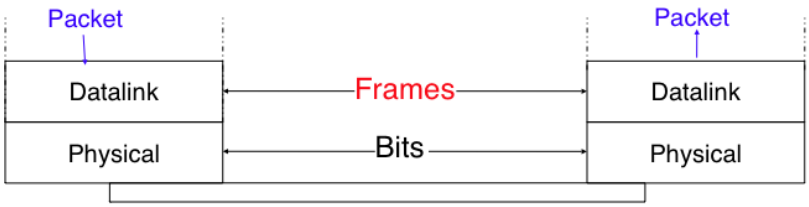


Figure 5.1: The datalink layer and the reference model

5.1.1 Framing

The *framing* problem can be phrased as : "*How does a sender encodes 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 solve the framing 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 server as a marker to delineate frame boundaries. Unfortunately, this solution is not sufficient for two reasons. First, some datalink layers cannot remain idle and need to always transmit bits. Second, inserting an idle period between frames decrease the maximum bandwidth that can be achieved by the datalink layer. Some physical layers provide an alternative to this idle period. All physical alyers are able to send and receive physical symbols that represent values *0* and *1*. However, tor various reasons that are outside the scope of this chapter, several physical layers are able to exchange other physical symbols as well. For example, the Manchester encoding used in several physical layers allows to send four different symbols. The Manchester encoding is a differential encoding scheme in which time is divided in 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. Besides, the encodings for *0* and *1*, the Manchester encoding also supports two additional symbols : *InvH* and *InvB* where the same voltage level is used during the two half periods. By definition, these two symbols cannot appear in the content of a frame which is only composed of *0* and *1*. Some technologies use these special symbols as markers at the beginning or end of frames.

Multi-symbol encodings cannot be used

idle (not perfect from a performance viewpoint)

Character stuffing

Bit stuffing

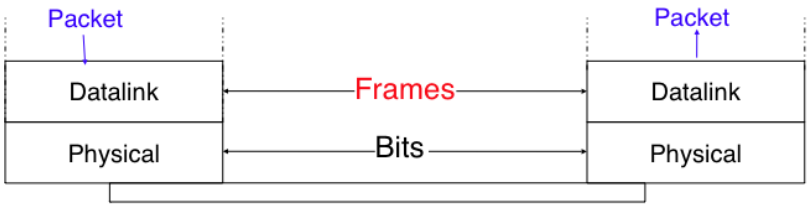


Figure 5.2: Manchester encoding

::

send(bit)

if bit==1 : count=count+1 if count==5 :

send(0) count=0

else count=0

send(bit)

recv(bit) if bit==0 and count==6 :

if bit==1 : count=count+1 if count==

example 0110111111111111111110010

Frame 01111110011011111011111011111011001001111110

Trailer versus header

When a segment format is designed for a transport protocol, it can be composed of three parts : a header, a payload and a trailer. The header is typically used to place most of the control information. However, the checksum/CRC may be placed either inside the header or inside the trailer.

- when the checksum/CRC is placed in the trailer, the sender can use hardware assistance on the interface card to compute the checksum/CRC while the segment is being sent. This is an optimisation that is now found on some high speed interfaces
- when the checksum/CRC is placed in the header, this implies, as segments are sent on the wire one byte after the other starting from the trailer, that the checksum/CRC must be computed before transmitting the segment. It is still possible to use hardware assistance to compute the CRC/checksum, but this is slightly more complex than when the checksum/CRC is placed inside a trailer.

5.1.2 Medium Access Control

ALOHA

Carrier Sense Multiple Access

Carrier Sense Multiple Access with Collision Detection

Carrier Sense Multiple Access with Collision Avoidance

Token Ring

5.2 Local Area Networks technologies

5.2.1 Ethernet

5.2.2 802.11

5.2.3 802.15.4

5.2.4 Token Ring

5.2.5 FDDI

5.2.6 Asynchronous Transfer Mode

CHAPTER

SIX

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 multiplicatevely 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

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 developped 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 stadard 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 langage 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 InterDomain Routing is the current address allocation architecture for IPv4. It was defined in **RFC 1518** and **RFC 4632**.

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

EGP Exterior Gateway Protocol. Synonym of interdomain routing protocol

EIGRP The Enhanced Interior Gateway Routing Protocol (EIGRP) is prorietary intradomain routing protocol that is often used in entreprise networks. EIGRP uses the DUAL algorithm described in [Garcia1993].

frame a frame is the unit of information transfert in the datalink layer

Frame-Relay A wide area networking technology using virtual circuits that is deploed by telecom operators.

ftp The File Transfert Protocol defined in **RFC 959** has been the de facto protocol to exchange files over the Internet before the widespread adoption of **:rfc:'http'**

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.

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 beend used in the path.

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

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

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.

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.

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 entreprise and ISP networks. OSPF is defined in and **RFC 2328** and **RFC 5340**

packet a packet is the unit of information transfert 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 entreprise 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 transfert 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**

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

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.

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

W3C The world wide web consortium was created to standardise the protocols and mechanisms used in the global www. It is thus focussed 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

CHAPTER

SEVEN

INDICES AND TABLES

- *Index*
- *Search Page*

BIBLIOGRAPHY

[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. DOI= http://doi.acm.org/10.1145/1052812.1052823

[B1989] Berners-Lee, T., Information Management: A Proposal, March 1989 http://www.w3.org/History/1989/proposal.html

[BE2007] Biondi, P. and A. Ebalard, "IPv6 Routing Header Security", CanSecWest Security Conference 2007, April 2007. http://www.secdev.org/conf/IPv6_RH_security-csw07.pdf

[CB2003] Cheswick, William R., Bellovin, Steven M., Rubin, Aviel D. Firewalls and internet security - Second edition - Repelling the Wily Hacker, Addison-Wesley 2003

[Clark88] Clark D., "The Design Philosophy of the DARPA Internet Protocols", Computer Communications Review 18:4, August 1988, pp. 106-114, http://groups.csail.mit.edu/ana/Publications/PubPDFs/The%20design%20philosophy%20of%20the%20DARPA%20internet%20pro

[CK74] Vinton Cerf, Robert Kahn, A Protocol for Packet Network Intercommunication, IEEE Transactions on Communications, May 1974

[CNPI09] CNPI, Security Assessment of the Transmission Control Protocol (TCP), http://www.cpni.gov.uk/Docs/tn-03-09-security-assessment-TCP.pdf

[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, http://www.ripe.net/ripe/docs/ipv6policy.html

[Feldmeier95] Feldmeier, D. C. 1995. Fast software implementation of error detection codes. IEEE/ACM Trans. Netw. 3, 6 (Dec. 1995), 640-651. DOI= http://dx.doi.org/10.1109/90.477710

[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, http://www.isi.edu/touch/pubs/infocomm99/

[Gill2004] Gill, V. , Lack of Priority Queuing Considered Harmful, ACM Queue, December 2004, http://queue.acm.org/detail.cfm?id=1036502

[Goralski2009] Goralski, W., The Illustrated network : How TCP/IP works in a modern network, Morgan Kaufmann, 2009

[KM1995] Kent, C. A. and Mogul, J. C. 1995. Fragmentation considered harmful. SIGCOMM Comput. Commun. Rev. 25, 1 (Jan. 1995), 75-87. DOI= http://doi.acm.org/10.1145/205447.205456

[KP91] Karn, P. and Partridge, C. 1991. Improving round-trip time estimates in reliable transport protocols. ACM Trans. Comput. Syst. 9, 4 (Nov. 1991), 364-373. DOI= http://doi.acm.org/10.1145/118544.118549

[KuroseRoss09] Kurose J. and Ross K., Computer networking : a top-down approach featuring the Internet, Addison-Wesley, 2009

[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. DOI= http://doi.acm.org/10.1145/217382.217465

[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. 2009. A brief history of the internet. SIGCOMM Comput. Commun. Rev. 39, 5 (Oct. 2009), 22-31. DOI= http://doi.acm.org/10.1145/1629607.1629613

[Paxson99] Paxson, V. End-to-end Internet packet dynamics. SIGCOMM Comput. Commun. Rev. 27, 4 (Oct. 1997), 139-152. DOI= http://doi.acm.org/10.1145/263109.263155

[Russel06] Russell A., Rough Consensus and Running Code and the Internet-OSI Standards War, IEEE Annals of the History of Computing, July-September 2006, http://www.computer.org/portal/cms_docs_annals/annals/content/promo2.pdf

[Sklower89] Sklower, K. 1989. Improving the efficiency of the OSI checksum calculation. SIGCOMM Comput. Commun. Rev. 19, 5 (Oct. 1989), 32-43. DOI= http://doi.acm.org/10.1145/74681.74684

[SGP98] Stone, J., Greenwald, M., Partridge, C., and Hughes, J. 1998. Performance of checksums and CRC's over real data. IEEE/ACM Trans. Netw. 6, 5 (Oct. 1998), 529-543. DOI= http://dx.doi.org/10.1109/90.731187

[SPMR09] Stigge, M., Plotz, H., Muller, W., Redlich, J., Reversing CRC - Theory and Practice. Berlin: Humboldt University Berlin. pp. 24. http://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/**SAR-PR-2006-05_**.pdf

[Selinger] Selinger, P., MD5 collision demo, http://www.mscs.dal.ca/~selinger/md5collision/

[Smm98] Semke, J., Mahdavi, J., and Mathis, M. 1998. Automatic TCP buffer tuning. SIGCOMM Comput. Commun. Rev. 28, 4 (Oct. 1998), 315-323. DOI= http://doi.acm.org/10.1145/285243.285292

[StevensTCP] Stevens, R., TCP/IP Illustrated : the Protocols, Addison-Wesley, 1994

[Thomborson1992] Thomborson, C., The V.42bis Standard for Data-Compressing Modems, IEEE Micro, September/October 1992 (vol. 12 no. 5), pp. 41-53, http://www.computer.org/portal/web/csdl/doi/10.1109/40.166712

[X200] ITU-T, recommendation X.200, Open Systems Interconnection - Model and Notation, 1994, http://www.itu.int/rec/T-REC-X.200-199407-I/en

[X224] ITU-T, recommendation X.224 : Information technology - Open Systems Interconnection - Protocol for providing the connection-mode transport service, 1995, http://www.itu.int/rec/T-REC-X.224-199511-I/en/

[Zimmermann80] Zimmermann, H., OSI Reference Model - The ISO Model of Architecture for Open Systems InterconnectionPDF (776 KB), IEEE Transactions on Communications, vol. 28, no. 4, April 1980, pp. 425 - 432. http://www.comsoc.org/livepubs/50_journals/pdf/RightsManagement_eid=136833.pdf

[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

[FLM2008] Fuller, V., Lear, E., Meyer, D., Reclassifying 240/4 as usable unicast address space, Internet draft, March 2008, workin progress, http://tools.ietf.org/html/draft-fuller-240space-02

[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, http://www.paris-traceroute.net/

[DT2007] Donnet, B. and Friedman, T.. Internet Topology Discovery: a Survey. IEEE Communications Surveys and Tutorials, 9(4):2-15, December 2007, http://inl.info.ucl.ac.be/publications/internet-topology-discovery-survey

[Stoll1988] Stoll, C. 1988. Stalking the wily hacker. Commun. ACM 31, 5 (May. 1988), 484-497. DOI= http://doi.acm.org/10.1145/42411.42412

[RE1989] Rochlis, J. A. and Eichin, M. W. 1989. With microscope and tweezers: the worm from MIT's perspective. Commun. ACM 32, 6 (Jun. 1989), 689-698. DOI= http://doi.acm.org/10.1145/63526.63528

[Cheswick1990] Cheswick, B., An Evening with Berferd In Which a Cracker is Lured, Endured, and Studied, Proc. Winter USENIX Conference, 1990, pp. 163-174, http://cheswick.com/ches/papers/berferd.pdf

[TE1993] Tsuchiya, P. F. and Eng, T. 1993. Extending the IP internet through address reuse. SIGCOMM Comput. Commun. Rev. 23, 1 (Jan. 1993), 16-33. DOI= http://doi.acm.org/10.1145/173942.173944

[Miyakawa2008] Miyakawa, S., From IPv4 only To v4/v6 Dual Stack, IETF72 IAB Technical Plenary, July 2008, http://www.nttv6.jp/~miyakawa/IETF72/IETF-IAB-TECH-PLENARY-NTT-miyakawa-extended.pdf

[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

[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, http://tools.ietf.org/html/draft-ietf-behave-v6v4-xlate-stateful-02

[Varghese2005] Varghese, G. , Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices, Morgan Kaufmann, 2005

[ISO10589] Information technology — Telecommunications and information exchange between systems — 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, http://standards.iso.org/ittf/PubliclyAvailableStandards/c030932_ISO_IEC_10589_2002(E).zip

[Malkin1999] Malkin, G., RIP: An Intra-Domain Routing Protocol, Addison Wesley, 1999

[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.

[Moy1998] Moy, J., OSPF: Anatomy of an Internet Routing Protocol, Addison Wesley, 1998

[ATLAS2009] Labovitz, C., Iekel-Johnson, S., McPherson, D., Oberheide, J., Jahanian F., Karir, M., ATLAS Internet Observatory 2009 Annual Report, presented at NANOG47, October 2009 http://www.nanog.org/meetings/nanog47/presentations/Monday/Labovitz_ObserveReport_N47_Mon.pdf

[KW2009] Katz, D., Ward, D., Bidirectional Forwarding Detection, Internet draft, http://tools.ietf.org/html/draft-ietf-bfd-base-09, Feb 2009, work in progress

[FFEB2005] Francois, P., Filsfils, C., Evans, J., and Bonaventure, O. 2005. Achieving sub-second IGP convergence in large IP networks. SIGCOMM Comput. Commun. Rev. 35, 3 (Jul. 2005), 35-44. DOI= http://doi.acm.org/10.1145/1070873.1070877

[VPD2004] Vasseur, J., Pickavet, M., and Demeester, P. 2004 Network Recovery: Protection and Restoration of Optical, Sonet-Sdh, Ip, and MPLS. Morgan Kaufmann Publishers Inc.

[WMS2004] White, R., Mc Pherson, D., Srihari, S., Practical BGP, Addison-Wesley, 2004, http://my.safaribooksonline.com/0321127005/

[Stewart1998] Stewart, J., BGP4: Inter-Domain Routing In The Internet, Addison-Wesley, 1998

[Garcia1993] Garcia-Lunes-Aceves, J., Loop-Free Routing Using Diffusing Computations, IEEE/ACM Transcations on Networking, Vol. 1, No, 1, Feb. 1993

[SARK2002] Subramanian, L., Agarwal, S., Rexford, J., Katz, R.. Characterizing the Internet hierarchy from multiple vantage points. In IEEE INFOCOM, 2002

[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. http://www.caida.org/outreach/papers/2002/Distance/

[GW2002] Griffin, T. and Wilfong, G. T. 2002. 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.

[GSW2002] Griffin, T. G., Shepherd, F. B., and Wilfong, G. 2002. The stable paths problem and interdomain routing. IEEE/ACM Trans. Netw. 10, 2 (Apr. 2002), 232-243.

[GW1999] Griffin, T. G. and Wilfong, G. 1999. An analysis of BGP convergence properties. SIGCOMM Comput. Commun. Rev. 29, 4 (Oct. 1999), 277-288. DOI= http://doi.acm.org/10.1145/316194.316231

[GGR2001] Gao, L., Griffin, T., Rexford, J., Inherently safe backup routing with BGP, Proc. IEEE INFOCOM, April 2001

[GR2001] Gao, L., Rexford, J., Stable Internet routing without global coordination, IEEE/ACM Transactions on Networking, December 2001, pp. 681-692

[COZ2008] Chi, Y., Oliveira, R., Zhang, L., Cyclops: The Internet AS-level Observatory, ACM SIGCOMM Computer Communication Review (CCR), October 2008

[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.

[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

[MUF+2007] Mühlbauer, W., Uhlig, S., Fu, B., Meulle, M., and Maennel, O. 2007. 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. DOI= http://doi.acm.org/10.1145/1282380.1282398

[BMO2006] Bhatia, M., Manral, V., Ohara, Y., IS-IS and OSPF Difference Discussions, Internet draft, Jan. 2006, http://tools.ietf.org/html/draft-bhatia-manral-diff-isis-ospf-01, work in progress

[Perlman2000] Perlman, R., Interconnections : Bridges, routers, switches and internetworking protocols, 2nd edition, Addison Wesley, 2000

[FRT2002] Fortz, B. Rexford, J. ,Thorup, M., Traffic engineering with traditional IP routing protocols, IEEE Communication Magazine, October 2002

[KZ1989] Khanna, A. and Zinky, J. 1989. The revised ARPANET routing metric. SIGCOMM Comput. Commun. Rev. 19, 4 (Aug. 1989), 45-56. DOI= http://doi.acm.org/10.1145/75247.75252

[Dijkstra1959] Dijkstra, E. A Note on Two Problems in Connection with Graphs. Numerische Mathematik, 1:269-271, 1959.

[MRR1979] McQuillan, J. M., Richer, I., and Rosen, E. C. 1979. 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. DOI= http://doi.acm.org/10.1145/800092.802981

[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.

[Jacobson1988] Jacobson, V. 1988. 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. DOI= http://doi.acm.org/10.1145/52324.52356

[RJ1995] Ramakrishnan, K. K. and Jain, R. 1995. 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. DOI= http://doi.acm.org/10.1145/205447.205461

[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. DOI= http://doi.acm.org/10.1145/263932.264023

[Leboudec2008] Leboudec, J.-Y., Rate Adaptation Congestion Control and Fairness : a tutorial, Dec. 2008, http://ica1www.epfl.ch/PS_files/LEB3132.pdf

[BF1995] Bonomi, F. 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, DOI= 10.1109/65.372653

[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, DOI= 10.1109/65.372658

[BOP1994] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. 1994. 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. DOI= http://doi.acm.org/10.1145/190314.190317

[HRX2008] Ha, S., Rhee, I., and Xu, L. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. SIGOPS Oper. Syst. Rev. 42, 5 (Jul. 2008), 64-74. DOI= http://doi.acm.org/10.1145/1400097.1400105

[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, http://tools.ietf.org/html/draft-sridharan-tcpm-ctcp-02

[SMM1998] Semke, J., Mahdavi, J., and Mathis, M. 1998. Automatic TCP buffer tuning. SIGCOMM Comput. Commun. Rev. 28, 4 (Oct. 1998), 315-323. DOI= http://doi.acm.org/10.1145/285243.285292

[LSP1982] Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (Jul. 1982), 382-401. DOI= http://doi.acm.org/10.1145/357172.357176

[Mills2006] Mills, D.L. Computer Network Time Synchronization: the Network Time Protocol. CRC Press, March 2006, 304 pp.

[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

[SG1990] Scheifler, R., Gettys, J., X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD, X Version 11, Release 4, Digital Press, http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V51B_ACRO_SUP/XWINSYS.PDF

[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. DOI= http://doi.acm.org/10.1145/948109.948113

[Cohen1980] Cohen, D., On Holy Wars and a Plea for Peace, IEN 137, April 1980, http://www.ietf.org/rfc/ien/ien137.txt

[Unicode] The Unicode Consortium. The Unicode Standard, Version 5.0.0, defined by: The Unicode Standard, Version 5.0 (Boston, MA, Addison-Wesley, 2007. ISBN 0-321-48091-0), http://unicode.org/versions/Unicode5.0.0/

[Metcalfe1976] 18. Metcalfe and D. Boggs. Ethernet: Distributed packet-switching for local computer networks. Communications of the ACM, 19(7):395–404, 1976. http://www.acm.org/pubs/citations/journals/cacm/1976-19-7/p395-metcalfe/

[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. available from http://standards.ieee.org/getieee802/802.11.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/getieee802/802.5.html

[FDDI] ANSI. Information systems - fiber distributed data interface (FDDI) - token ring media access control (MAC). ANSI X3.139-1987 (R1997), 1997

[Benvenuti2005] 3. Benvenuti, Understanding Linux Network Internals, O'Reilly Media, 2005, http://oreilly.com/catalog/9780596002558

INDEX

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X