IPv6, the UDP checksum cannot be disabled. A detailed discussion of the implementation of the Internet checksum may be found in **RFC 1071**

Several types of applications rely on UDP. As a rule of thumb, UDP is used for applications where delay must be minimised or losses can be recovered by the application itself. A first class of the UDP-based applications are applications where the client sends a short request and expects a quick and short answer. The *DNS* is an example of a UDP application that is often used in the wide area. However, in local area networks, many distributed systems rely on Remote Procedure Call (*RPC*) that is often used on top of UDP. In Unix environments, the Network File System (*NFS*) is built on top of RPC and runs frequently on top of UDP. A second class of UDP-based applications are the interactive computer games that need to frequently exchange small messages, such as the player's location or their recent actions. Many of these games use UDP to minimise the delay and can recover from losses. A third class of applications are multimedia applications such as interactive Voice over IP or interactive Video over IP. These interactive applications expect a delay shorter than about 200 milliseconds between the sender and the receiver and can recover from losses directly inside the application.

3.11 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 [1]. However, the basics of the protocol remain and an implementation that only supports **RFC 793** should inter-operate 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*, *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 [2] have reported that TCP was responsible for more than 90% of the data exchanged in the global Internet.

To provide this service, TCP relies on a simple segment format that is shown in the figure below. Each TCP segment contains a header described below and, optionally, a payload. The default length of the TCP header is twenty bytes, but some TCP headers contain options.

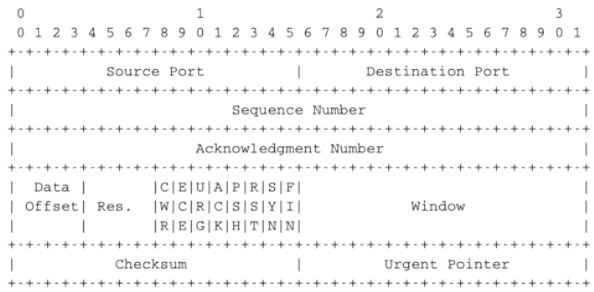


Fig. 3.20: TCP header format

A TCP header contains the following fields :

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

[2] 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

- Source and destination ports. The source and destination ports play an important role in TCP, as they allow the identification of the connection to which a TCP segment belongs. When a client opens a TCP connection, it typically selects an ephemeral TCP port number as its source port and contacts the server by using the server's port number. All the segments that are sent by the client on this connection have the same source and destination ports. The server sends segments that contain as source (resp. destination) port, the destination (resp. source) port of the segments sent by the client (see figure *Utilization of the TCP source and destination ports*). A TCP connection is always identified by four pieces of information :
  - the address of the client
  - the address of the server
  - the port chosen by the client
  - the port chosen by the server
- the *sequence number* (32 bits), *acknowledgement number* (32 bits) and *window* (16 bits) fields are used to provide a reliable data transfer, using a window-based protocol. In a TCP bytestream, each byte of the stream consumes one sequence number. Their utilisation will be described in more detail in section *TCP reliable data transfer*
- the *Urgent pointer* is used to indicate that some data should be considered as urgent in a TCP bytestream. However, it is rarely used in practice and will not be described here. Additional details about the utilisation of this pointer may be found in **RFC 793**, **RFC 1122** or *[Stevens1994]*
- the flags field contains a set of bit flags that indicate how a segment should be interpreted by the TCP entity receiving it :
  - the *SYN* flag is used during connection establishment
  - the *FIN* flag is used during connection release
  - the *RST* is used in case of problems or when an invalid segment has been received
  - when the *ACK* flag is set, it indicates that the *acknowledgment* field contains a valid number. Otherwise, the content of the *acknowledgment* field must be ignored by the receiver
  - the *URG* flag is used together with the *Urgent pointer*
  - the *PSH* flag is used as a notification from the sender to indicate to the receiver that it should pass all the data it has received to the receiving process. However, in practice TCP implementations do not allow TCP users to indicate when the *PSH* flag should be set and thus there are few real utilizations of this flag.
- the *checksum* field contains the value of the Internet checksum computed over the entire TCP segment and a pseudo-header as with UDP
- the *Reserved* field was initially reserved for future utilization. It is now used by **RFC 3168**.
- the *TCP Header Length* (THL) or *Data Offset* field is a four bits field that indicates the size of the TCP header in 32 bit words. The maximum size of the TCP header is thus 64 bytes.
- the *Optional header extension* is used to add optional information to the TCP header. Thanks to this header extension, it is possible to add new fields to the TCP header that were not planned in the original specification. This allowed TCP to evolve since the early eighties. The details of the TCP header extension are explained in sections *TCP connection establishment* and *TCP reliable data transfer*.

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.

3.11.1 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

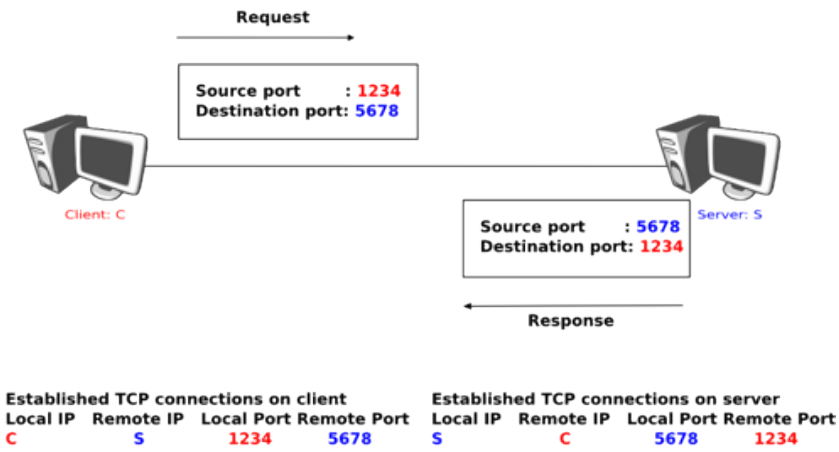


Fig. 3.21: Utilization of the TCP source and destination ports

communicating hosts negotiate the initial sequence number to be used in both directions of the connection. For this, each TCP entity maintains a 32 bits counter, which is supposed to be incremented by one at least every 4 microseconds and after each connection establishment [3]. When a client host wants to open a TCP connection with a server host, it creates a TCP segment with :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the client host's TCP entity

Upon reception of this segment (which is often called a *SYN segment*), the server host replies with a segment containing :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the server host's TCP entity
- the *ACK* flag set
- the *acknowledgment number* set to the *sequence number* of the received *SYN* segment incremented by 1 $(\mathrm{mod}\ 2^{32})$. 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 $(\mathrm{mod}\ 2^{32})$

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.

In the figure above, the connection is considered to be established by the client once it has received the *SYN+ACK* segment, while the server considers the connection to be established upon reception of the *ACK* segment. The first data segment sent by the client (server) has its *sequence number* set to *x+1* (resp. *y+1*).

Note: Computing TCP's initial sequence number

[3] 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**).

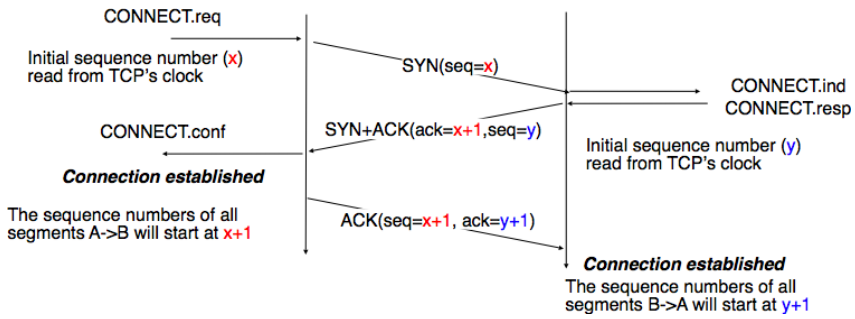


Fig. 3.22: Establishment of a TCP connection

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 [4]. 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 confirming the reception of the *SYN+ACK* segment sent by the server. Once the TCP connection is open, he can use it to send any command to the server. To counter this attack, current TCP implementations add randomness to the *ISN*. One of the solutions, proposed in **RFC 1948** is to compute the *ISN* as

```
ISN = M + H(localhost, localport, remotehost, remoteport, secret).
```

where *M* is the current value of the TCP clock and *H* is a cryptographic hash function. *localhost* and *remotehost* (resp. *localport* and *remoteport*) are the IP addresses (port numbers) of the local and remote host and *secret* is a random number only known by the server. This method allows the server to use different ISNs for different clients at the same time. Measurements performed with the first implementations of this technique showed that it was difficult to implement it correctly, but today's TCP implementation now generate good ISNs.

A server could, of course, refuse to open a TCP connection upon reception of a *SYN* segment. This refusal may be due to various reasons. There may be no server process that is listening on the destination port of the *SYN* segment. The server could always refuse connection establishments from this particular client (e.g. due to security reasons) or the server may not have enough resources to accept a new TCP connection at that time. In this case, the server would reply with a TCP segment having its *RST* flag set and containing the *sequence number* of the received *SYN* segment as its *acknowledgment number*. This is illustrated in the figure below. We discuss the other utilizations of the TCP *RST* flag later (see *TCP connection release*).

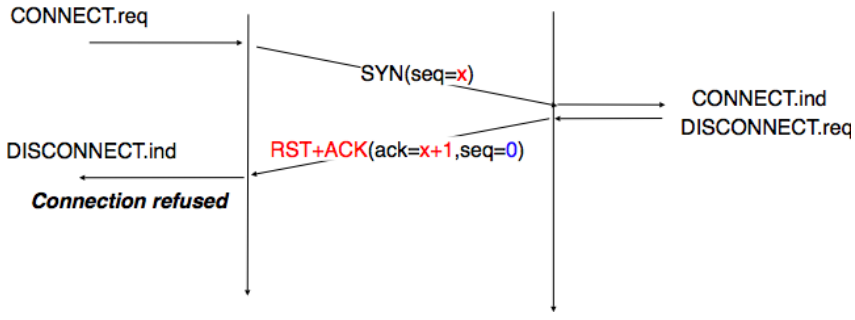


Fig. 3.23: TCP connection establishment rejected by peer

TCP connection establishment can be described as the four state Finite State Machine shown below. In this FSM, *!X* (resp. *?Y*) indicates the transmission of segment *X* (resp. reception of segment *Y*) during the corresponding

[4] On many departmental networks containing Unix workstations, it was common to allow users on one of the hosts to use rlogin **RFC 1258** to run commands on any of the workstations of the network without giving any password. In this case, the remote workstation "authenticated" the client host based on its IP address. This was a bad practice from a security viewpoint.

transition. *Init* is the initial state.

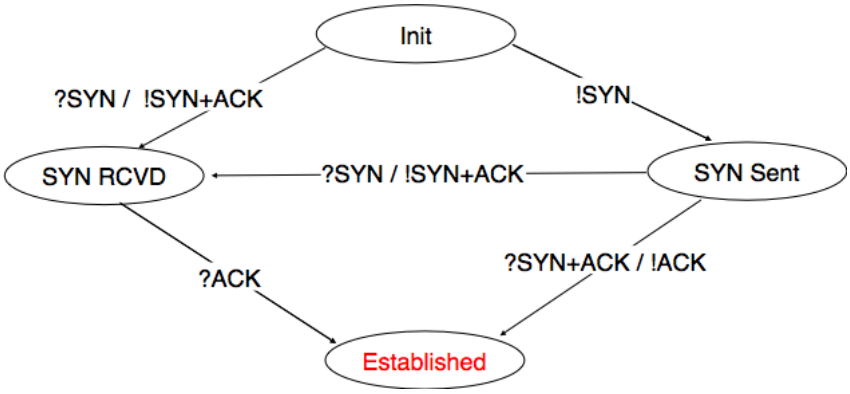


Fig. 3.24: TCP FSM for connection establishment

A client host starts in the *Init* state. It then sends a *SYN* segment and enters the *SYN Sent* state where it waits for a *SYN+ACK* segment. Then, it replies with an *ACK* segment and enters the *Established* state where data can be exchanged. On the other hand, a server host starts in the *Init* state. When a server process starts to listen to a destination port, the underlying TCP entity creates a TCP control block and a queue to process incoming *SYN* segments. Upon reception of a *SYN* segment, the server's TCP entity replies with a *SYN+ACK* and enters the *SYN RCVD* state. It remains in this state until it receives an *ACK* segment that acknowledges its *SYN+ACK* segment, with this it then enters the Established state.

Apart from these two paths in the TCP connection establishment FSM, there is a third path that corresponds to the case when both the client and the server send a *SYN* segment to open a TCP connection [5]. 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.

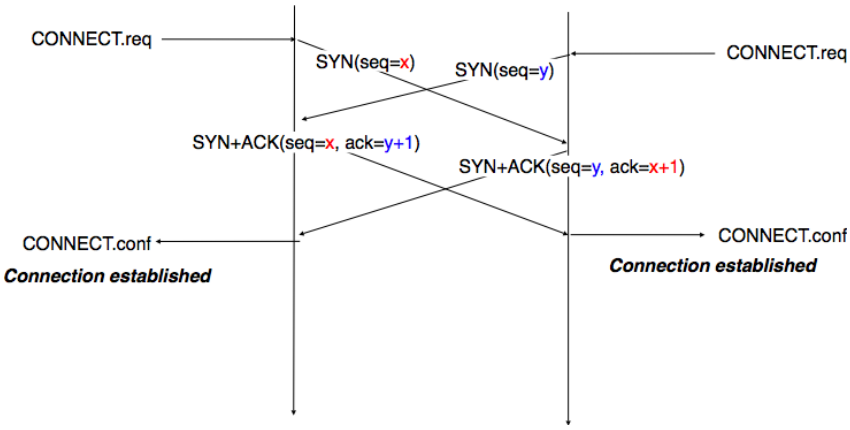


Fig. 3.25: Simultaneous establishment of a TCP connection

Denial of Service attacks

When a TCP entity opens a TCP connection, it creates a Transmission Control Block (*TCB*). The TCB contains the entire state that is maintained by the TCP entity for each TCP connection. During connection establishment, the TCB contains the local IP address, the remote IP address, the local port number, the remote port number, the

[5] 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.

current local sequence number, the last sequence number received from the remote entity. Until the mid 1990s, TCP implementations had a limit on the number of TCP connections that could be in the *SYN RCVD* state at a given time. Many implementations set this limit to about 100 TCBs. This limit was considered sufficient even for heavily load http servers given the small delay between the reception of a *SYN* segment and the reception of the *ACK* segment that terminates the establishment of the TCP connection. When the limit of 100 TCBs in the *SYN Rcvd* state is reached, the TCP entity discards all received TCP *SYN* segments that do not correspond to an existing TCB.

This limit of 100 TCBs in the *SYN Rcvd* state was chosen to protect the TCP entity from the risk of overloading its memory with too many TCBs in the *SYN Rcvd* state. However, it was also the reason for a new type of Denial of Service (DoS) attack **RFC 4987**. A DoS attack is defined as an attack where an attacker can render a resource unavailable in the network. For example, an attacker may cause a DoS attack on a 2 Mbps link used by a company by sending more than 2 Mbps of packets through this link. In this case, the DoS attack was more subtle. As a TCP entity discards all received *SYN* segments as soon as it has 100 TCBs in the *SYN Rcvd* state, an attacker simply had to send a few 100 *SYN* segments every second to a server and never reply to the received *SYN+ACK* segments. To avoid being caught, attackers were of course sending these *SYN* segments with a different address than their own IP address [6]. On most TCP implementations, once a TCB entered the *SYN Rcvd* state, it remained in this state for several seconds, waiting for a retransmission of the initial *SYN* segment. This attack was later called a *SYN flood* attack and the servers of the ISP named panix were among the first to be affected by this attack.

To avoid the *SYN flood* attacks, recent TCP implementations no longer enter the *SYN Rcvd* state upon reception of a *SYN segment*. Instead, they reply directly with a *SYN+ACK* segment and wait until the reception of a valid *ACK*. This implementation trick is only possible if the TCP implementation is able to verify that the received *ACK* segment acknowledges the *SYN+ACK* segment sent earlier without storing the initial sequence number of this *SYN+ACK* segment in a TCB. The solution to solve this problem, which is known as SYN cookies is to compute the 32 bits of the *ISN* as follows :

- the high order bits contain the low order bits of a counter that is incremented slowly
- the low order bits contain a hash value computed over the local and remote IP addresses and ports and a random secret only known to the server

The advantage of the SYN cookies is that by using them, the server does not need to create a *TCB* upon reception of the *SYN* segment and can still check the returned *ACK* segment by recomputing the *SYN cookie*. The main disadvantage is that they are not fully compatible with the TCP options. This is why they are not enabled by default on a typical system.

[6] Sending a packet with a different source IP address than the address allocated to the host is called sending a *spoofed packet*.

Retransmitting the first *SYN* segment

As IP provides an unreliable connectionless service, the *SYN* and *SYN+ACK* segments sent to open a TCP connection could be lost. Current TCP implementations start a retransmission timer when they send the first *SYN* segment. This timer is often set to three seconds for the first retransmission and then doubles after each retransmission **RFC 2988**. TCP implementations also enforce a maximum number of retransmissions for the initial *SYN* segment.

As explained earlier, TCP segments may contain an optional header extension. In the *SYN* and *SYN+ACK* segments, these options are used to negotiate some parameters and the utilisation of extensions to the basic TCP specification.

The first parameter which is negotiated during the establishment of a TCP connection is the Maximum Segment Size (*MSS*). The MSS is the size of the largest segment that a TCP entity is able to process. According to **RFC 879**, all TCP implementations must be able to receive TCP segments containing 536 bytes of payload. However, most TCP implementations are able to process larger segments. Such TCP implementations use the TCP MSS Option in the *SYN*/*SYN+ACK* segment to indicate the largest segment they are able to process. The MSS value indicates the maximum size of the payload of the TCP segments. The client (resp. server) stores in its *TCB* the MSS value announced by the server (resp. the client).

Another utilisation of TCP options during connection establishment is to enable TCP extensions. For example, consider **RFC 1323** (which is discussed in *TCP reliable data transfer*). **RFC 1323** defines TCP extensions to support timestamps and larger windows. If the client supports **RFC 1323**, it adds a **RFC 1323** option to its *SYN* segment. If the server understands this **RFC 1323** option and wishes to use it, it replies with an **RFC 1323** option in the *SYN+ACK* segment and the extension defined in **RFC 1323** is used throughout the TCP connection. Otherwise, if the server's *SYN+ACK* does not contain the **RFC 1323** option, the client is not allowed to use this extension and the corresponding TCP header options throughout the TCP connection. TCP's option mechanism is flexible and it allows the extension of TCP while maintaining compatibility with older implementations.

The TCP options are encoded by using a Type Length Value format where :

- the first byte indicates the *type* of the option.
- the second byte indicates the total length of the option (including the first two bytes) in bytes
- the last bytes are specific for each type of option

RFC 793 defines the Maximum Segment Size (MSS) TCP option that must be understood by all TCP implementations. This option (type 2) has a length of 4 bytes and contains a 16 bits word that indicates the MSS supported by the sender of the *SYN* segment. The MSS option can only be used in TCP segments having the *SYN* flag set.

RFC 793 also defines two special options that must be supported by all TCP implementations. The first option is *End of option*. It is encoded as a single byte having value *0x00* and can be used to ensure that the TCP header extension ends on a 32 bits boundary. The *No-Operation* option, encoded as a single byte having value *0x01*, can be used when the TCP header extension contains several TCP options that should be aligned on 32 bit boundaries. All other options [7] are encoded by using the TLV format.

Note: The robustness principle

The handling of the TCP options by TCP implementations is one of the many applications of the *robustness principle* which is usually attributed to Jon Postel and is often quoted as *"Be liberal in what you accept, and conservative in what you send"* **RFC 1122**

Concerning the TCP options, the robustness principle implies that a TCP implementation should be able to accept TCP options that it does not understand, in particular in received *SYN* segments, and that it should be able to parse any received segment without crashing, even if the segment contains an unknown TCP option. Furthermore, a server should not send in the *SYN+ACK* segment or later, options that have not been proposed by the client in the *SYN* segment.

3.11.2 TCP reliable data transfer

The original TCP data transfer mechanisms were defined in **RFC 793**. Based on the experience of using TCP on the growing global Internet, this part of the TCP specification has been updated and improved several times, always while preserving the backward compatibility with older TCP implementations. In this section, we review the main data transfer mechanisms used by TCP.

TCP is a window-based transport protocol that provides a bi-directional byte stream service. This has several implications on the fields of the TCP header and the mechanisms used by TCP. The three fields of the TCP header are :

- *sequence number*. TCP uses a 32 bits sequence number. The *sequence number* placed in the header of a TCP segment containing data is the sequence number of the first byte of the payload of the TCP segment.
- *acknowledgement number*. TCP uses cumulative positive acknowledgements. Each TCP segment contains the *sequence number* of the next byte that the sender of the acknowledgement expects to receive from the remote host. In theory, the *acknowledgement number* is only valid if the *ACK* flag of the TCP header is set. In practice almost all [8] TCP segments have their *ACK* flag set.

[7] The full list of all TCP options may be found at http://www.iana.org/assignments/tcp-parameters/
[8] In practice, only the *SYN* segment do not have their *ACK* flag set.

- *window*. a TCP receiver uses this 16 bits field to indicate the current size of its receive window expressed in bytes.

Note: The Transmission Control Block

For each established TCP connection, a TCP implementation must maintain a Transmission Control Block (*TCB*). A TCB contains all the information required to send and receive segments on this connection **RFC 793**. This includes [9] :

- the local IP address
- the remote IP address
- the local TCP port number
- the remote TCP port number
- the current state of the TCP FSM
- the *maximum segment size* (MSS)
- *snd.nxt* : the sequence number of the next byte in the byte stream (the first byte of a new data segment that you send uses this sequence number)
- *snd.una* : the earliest sequence number that has been sent but has not yet been acknowledged
- *snd.wnd* : the current size of the sending window (in bytes)
- *rcv.nxt* : the sequence number of the next byte that is expected to be received from the remote host
- *rcv.wnd* : the current size of the receive window advertised by the remote host
- *sending buffer* : a buffer used to store all unacknowledged data
- *receiving buffer* : a buffer to store all data received from the remote host that has not yet been delivered to the user. Data may be stored in the *receiving buffer* because either it was not received in sequence or because the user is too slow to process it

---

The original TCP specification can be categorised as a transport protocol that provides a byte stream service and uses *go-back-n*.

To send new data on an established connection, a TCP entity performs the following operations on the corresponding TCB. It first checks that the *sending buffer* does not contain more data than the receive window advertised by the remote host (*rcv.wnd*). If the window is not full, up to *MSS* bytes of data are placed in the payload of a TCP segment. The *sequence number* of this segment is the sequence number of the first byte of the payload. It is set to the first available sequence number : *snd.nxt* and *snd.nxt* is incremented by the length of the payload of the TCP segment. The *acknowledgement number* of this segment is set to the current value of *rcv.nxt* and the *window* field of the TCP segment is computed based on the current occupancy of the *receiving buffer*. The data is kept in the *sending buffer* in case it needs to be retransmitted later.

When a TCP segment with the *ACK* flag set is received, the following operations are performed. *rcv.wnd* is set to the value of the *window* field of the received segment. The *acknowledgement number* is compared to *snd.una*. The newly acknowledged data is removed from the *sending buffer* and *snd.una* is updated. If the TCP segment contained data, the *sequence number* is compared to *rcv.nxt*. If they are equal, the segment was received in sequence and the data can be delivered to the user and *rcv.nxt* is updated. The contents of the *receiving buffer* is checked to see whether other data already present in this buffer can be delivered in sequence to the user. If so, *rcv.nxt* is updated again. Otherwise, the segment's payload is placed in the *receiving buffer*.

Segment transmission strategies

In a transport protocol such as TCP that offers a bytestream, a practical issue that was left as an implementation choice in **RFC 793** is to decide when a new TCP segment containing data must be sent. There are two simple and

[9] 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.

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 would place each user byte in a segment containing 20 bytes of TCP header [11]. This is a huge overhead that is not acceptable in wide area networks. A second simple solution would be to only transmit a new TCP segment once the user has produced MSS bytes of data. This solution reduces the overhead, but at the cost of a potentially very high delay.

An elegant solution to this problem was proposed by John Nagle in **RFC 896**. John Nagle observed that the overhead caused by the TCP header was a problem in wide area connections, but less in local area connections where the available bandwidth is usually higher. He proposed the following rules to decide to send a new data segment when a new data has been produced by the user or a new ack segment has been received

```
if rcv.wnd>= MSS and len(data) >= MSS :
  send one MSS-sized segment
else
  if there are unacknowledged data:
    place data in buffer until acknowledgement has been received
  else
    send one TCP segment containing all buffered data
```

The first rule ensures that a TCP connection used for bulk data transfer always sends full TCP segments. The second rule sends one partially filled TCP segment every round-trip-time.

This algorithm, called the Nagle algorithm, takes a few lines of code in all TCP implementations. These lines of code have a huge impact on the packets that are exchanged in TCP/IP networks. Researchers have analysed the distribution of the packet sizes by capturing and analysing all the packets passing through a given link. These studies have shown several important results :

- in TCP/IP networks, a large fraction of the packets are TCP segments that contain only an acknowledgement. These packets usually account for 40-50% of the packets passing through the studied link
- in TCP/IP networks, most of the bytes are exchanged in long packets, usually packets containing about 1440 bytes of payload which is the default MSS for hosts attached to an Ethernet network, the most popular type of LAN

Recent measurements indicate that these packet size distributions are still valid in today's Internet, although the packet distribution tends to become bimodal with small packets corresponding to TCP pure acks and large 1440-bytes packets carrying most of the user data *[SMASU2012]*.

3.11.3 TCP windows

From a performance point of view, one of the main limitations of the original TCP specification is the 16 bits *window* field in the TCP header. As this field indicates the current size of the receive window in bytes, it limits the TCP receive window at 65535 bytes. This limitation was not a severe problem when TCP was designed since at that time high-speed wide area networks offered a maximum bandwidth of 56 kbps. However, in today's network, this limitation is not acceptable anymore. The table below provides the rough [12] 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

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

[12] 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.

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 [13]. Today's high performance implementations are able to automatically adjust the size of the sending and receiving buffer to better support high bandwidth flows *[SMM1998]*

3.11.4 TCP's retransmission timeout

In a go-back-n transport protocol such as TCP, the retransmission timeout must be correctly set in order to achieve good performance. If the retransmission timeout expires too early, then bandwidth is wasted by retransmitting segments that have already been correctly received; whereas if the retransmission timeout expires too late, then bandwidth is wasted because the sender is idle waiting for the expiration of its retransmission timeout.

A good setting of the retransmission timeout clearly depends on an accurate estimation of the round-trip-time of each TCP connection. The round-trip-time differs between TCP connections, but may also change during the lifetime of a single connection. For example, the figure below shows the evolution of the round-trip-time between two hosts during a period of 45 seconds.

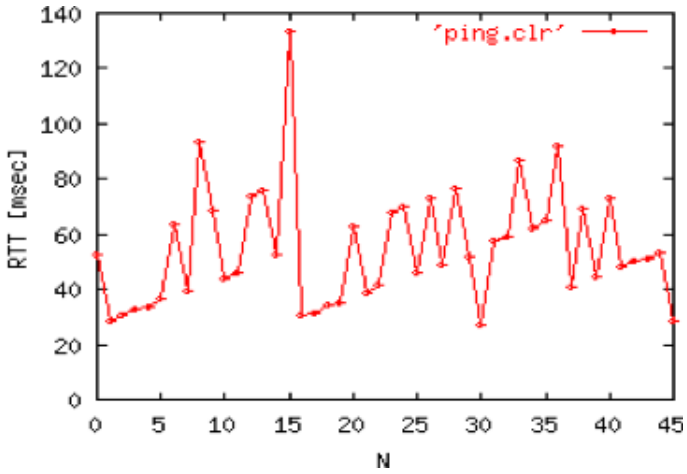


Fig. 3.26: Evolution of the round-trip-time between two hosts

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

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

[14] 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**

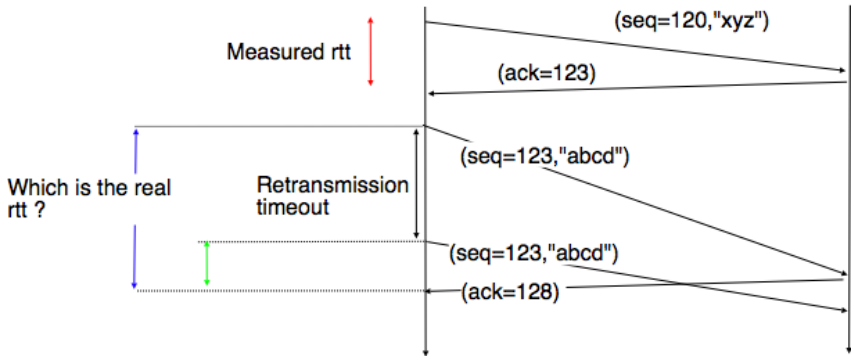


Fig. 3.27: How to measure the round-trip-time ?

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 bit timestamps in each TCP segment that it sends. The first timestamp, TS Value (*TSval*) is chosen by the sender of the segment. It could for example be the current value of its real-time clock [15]. The second value, TS Echo Reply (*TSecr*), is the last *TSval* that was received from the remote host and stored in the *TCB*. The figure below shows how the utilization of this timestamp option allows for the disambiguation of the round-trip-time measurement when there are retransmissions.

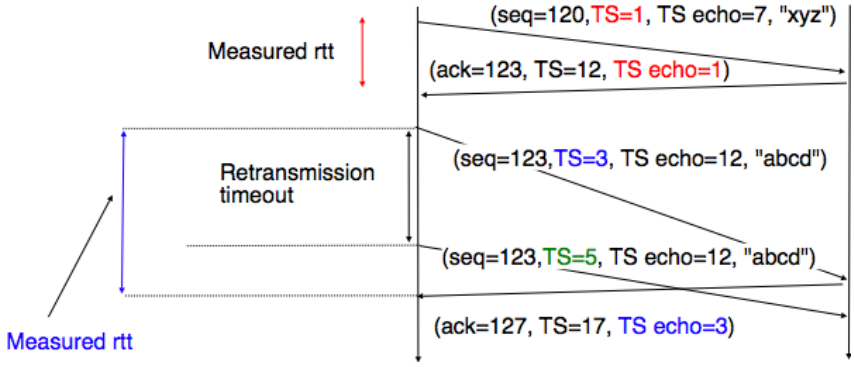


Fig. 3.28: Disambiguating round-trip-time measurements with the **RFC 1323** timestamp option

Once the round-trip-time measurements have been collected for a given TCP connection, the TCP entity must compute the retransmission timeout. As the round-trip-time measurements may change during the lifetime of a connection, the retransmission timeout may also change. At the beginning of a connection [16] , the TCP entity that sends a *SYN* segment does not know the round-trip-time to reach the remote host and the initial retransmission timeout is usually set to 3 seconds **RFC 2988**.

The original TCP specification proposed in **RFC 793** to include two additional variables in the TCB :

- *srtt* : the smoothed round-trip-time computed as $srrt = (\alpha \times srtt) + ((1 - \alpha) \times rtt)$ where *rtt* is the round-trip-time measured according to the above procedure and α a smoothing factor (e.g. 0.8 or 0.9)
- *rto* : the retransmission timeout is computed as $rto = \min(60, max(1, \beta \times srtt))$ where β is used to take

[15] Some security experts have raised concerns that using the real-time clock to set the *TSval* in the timestamp option can leak information such as the system's up-time. Solutions proposed to solve this problem may be found in *[CNPI09]*

[16] 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.

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 follows :

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

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

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

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

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

The proposed values for the parameters are $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$. This allows a TCP implementation, implemented in the kernel, to perform the *rtt* computation by using shift operations instead of the more costly floating point operations *[Jacobson1988]*. The figure below illustrates the computation of the *rto* upon *rtt* changes.

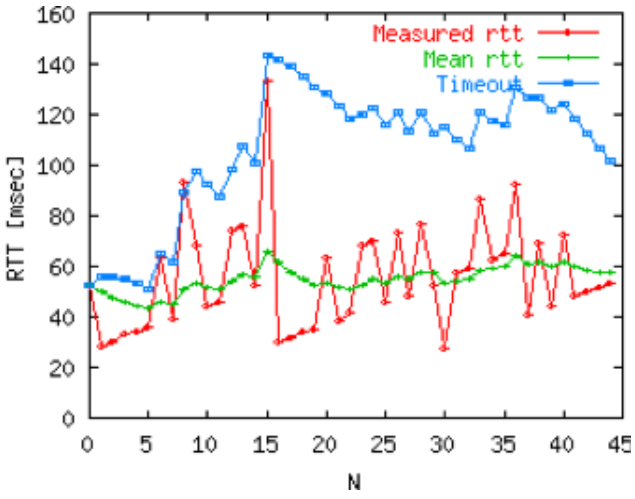


Fig. 3.29: Example computation of the *rto*

3.11.5 Advanced retransmission strategies

The default go-back-n retransmission strategy was defined in **RFC 793**. When the retransmission timer expires, TCP retransmits the first unacknowledged segment (i.e. the one having sequence number *snd.una*). After each expiration of the retransmission timeout, **RFC 2988** recommends to double the value of the retransmission timeout. This is called an *exponential backoff*. This doubling of the retransmission timeout after a retransmission was included in TCP to deal with issues such as network/receiver overload and incorrect initial estimations of the retransmission timeout. If the same segment is retransmitted several times, the retransmission timeout is doubled after every retransmission until it reaches a configured maximum. **RFC 2988** suggests a maximum retransmission timeout of at least 60 seconds. Once the retransmission timeout reaches this configured maximum, the remote host is considered to be unreachable and the TCP connection is closed.

This retransmission strategy has been refined based on the experience of using TCP on the Internet. The first refinement was a clarification of the strategy used to send acknowledgements. As TCP uses piggybacking, the

easiest and less costly method to send acknowledgements is to place them in the data segments sent in the other direction. However, few application layer protocols exchange data in both directions at the same time and thus this method rarely works. For an application that is sending data segments in one direction only, the remote TCP entity returns empty TCP segments whose only useful information is their acknowledgement number. This may cause a large overhead in wide area network if a pure *ACK* segment is sent in response to each received data segment. Most TCP implementations use a *delayed acknowledgement* strategy. This strategy ensures that piggybacking is used whenever possible, otherwise pure *ACK* segments are sent for every second received data segments when there are no losses. When there are losses or reordering, *ACK* segments are more important for the sender and they are sent immediately **RFC 813 RFC 1122**. This strategy relies on a new timer with a short delay (e.g. 50 milliseconds) and one additional flag in the TCB. It can be implemented as follows :

```
reception of a data segment:
   if pkt.seq==rcv.nxt:   # segment received in sequence
      if delayedack :
         send pure ack segment
         cancel acktimer
         delayedack=False
      else:
         delayedack=True
         start acktimer
   else:                      # out of sequence segment
      send pure ack segment
      if delayedack:
         delayedack=False
         cancel acktimer

transmission of a data segment:  # piggyback ack
   if delayedack:
      delayedack=False
      cancel acktimer

acktimer expiration:
   send pure ack segment
   delayedack=False
```

Due to this delayed acknowledgement strategy, during a bulk transfer, a TCP implementation usually acknowledges every second TCP segment received.

The default go-back-n retransmission strategy used by TCP has the advantage of being simple to implement, in particular on the receiver side, but when there are losses, a go-back-n strategy provides a lower performance than a selective repeat strategy. The TCP developers have designed several extensions to TCP to allow it to use a selective repeat strategy while maintaining backward compatibility with older TCP implementations. These TCP extensions assume that the receiver is able to buffer the segments that it receives out-of-sequence.

The first extension that was proposed is the fast retransmit heuristic. This extension can be implemented on TCP senders and thus does not require any change to the protocol. It only assumes that the TCP receiver is able to buffer out-of-sequence segments.

From a performance point of view, one issue with TCP's *retransmission timeout* is that when there are isolated segment losses, the TCP sender often remains idle waiting for the expiration of its retransmission timeouts. Such isolated losses are frequent in the global Internet *[Paxson99]*. A heuristic to deal with isolated losses without waiting for the expiration of the retransmission timeout has been included in many TCP implementations since the early 1990s. To understand this heuristic, let us consider the figure below that shows the segments exchanged over a TCP connection when an isolated segment is lost.

As shown above, when an isolated segment is lost the sender receives several *duplicate acknowledgements* since the TCP receiver immediately sends a pure acknowledgement when it receives an out-of-sequence segment. A duplicate acknowledgement is an acknowledgement that contains the same *acknowledgement number* as a previous segment. A single duplicate acknowledgement does not necessarily imply that a segment was lost, as a simple reordering of the segments may cause duplicate acknowledgements as well. Measurements *[Paxson99]* have shown that segment reordering is frequent in the Internet. Based on these observations, the *fast retransmit* heuristic has been included in most TCP implementations. It can be implemented as follows :

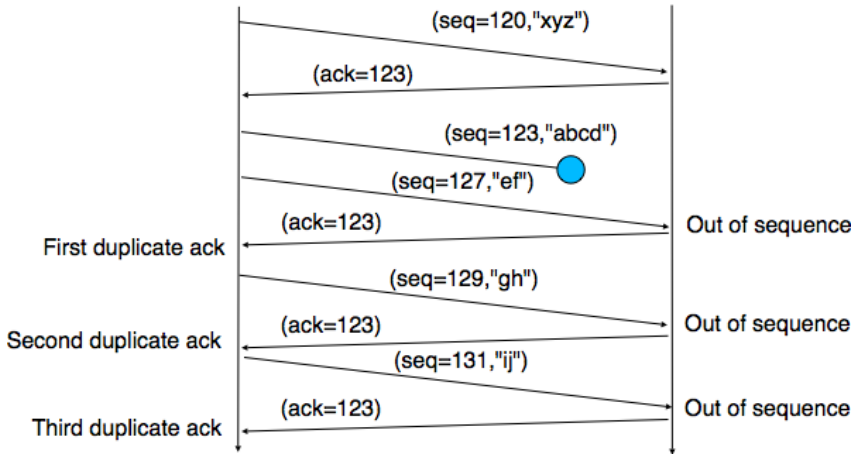


Fig. 3.30: Detecting isolated segment losses

```
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* heuristic improves the TCP performance provided that isolated segments are lost and the current window is large enough to allow the sender to send three duplicate acknowledgements.

The figure below illustrates the operation of the *fast retransmit* heuristic.

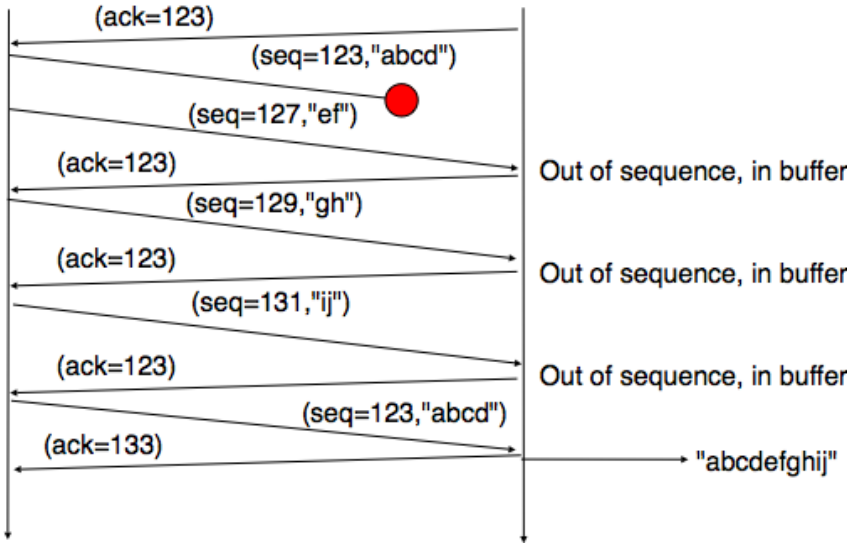


Fig. 3.31: TCP fast retransmit heuristics

When losses are not isolated or when the windows are small, the performance of the *fast retransmit* heuristic decreases. In such environments, it is necessary to allow a TCP sender to use a selective repeat strategy instead of the default go-back-n strategy. Implementing selective-repeat requires a change to the TCP protocol as the receiver needs to be able to inform the sender of the out-of-order segments that it has already received. This can be done by using the Selective Acknowledgements (SACK) option defined in **RFC 2018**. This TCP option is

negotiated during the establishment of a TCP connection. If both TCP hosts support the option, SACK blocks can be attached by the receiver to the segments that it sends. SACK blocks allow a TCP receiver to indicate the blocks of data that it has received correctly but out of sequence. The figure below illustrates the utilisation of the SACK blocks.

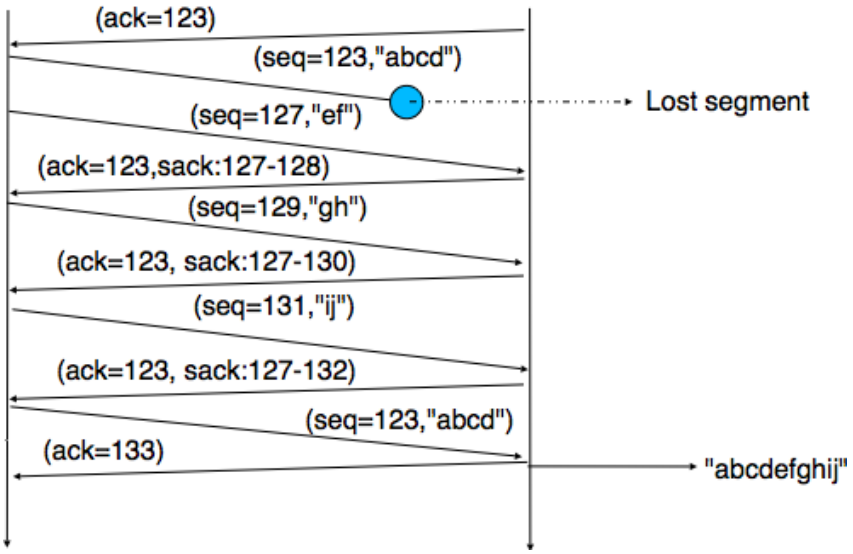


Fig. 3.32: TCP selective acknowledgements

An SACK option contains one or more blocks. A block corresponds to all the sequence numbers between the *left edge* and the *right edge* of the block. The two edges of the block are encoded as 32 bit numbers (the same size as the TCP sequence number) in an SACK option. As the SACK option contains one byte to encode its type and one byte for its length, a SACK option containing b blocks is encoded as a sequence of $2 + 8 \times b$ bytes. In practice, the size of the SACK option can be problematic as the optional TCP header extension cannot be longer than 44 bytes. As the SACK option is usually combined with the **RFC 1323** timestamp extension, this implies that a TCP segment cannot usually contain more than three SACK blocks. This limitation implies that a TCP receiver cannot always place in the SACK option that it sends, information about all the received blocks.

To deal with the limited size of the SACK option, a TCP receiver currently having more than 3 blocks inside its receiving buffer must select the blocks to place in the SACK option. A good heuristic is to put in the SACK option the blocks that have most recently changed, as the sender is likely to be already aware of the older blocks.

When a sender receives an SACK option indicating a new block and thus a new possible segment loss, it usually does not retransmit the missing segments immediately. To deal with reordering, a TCP sender can use a heuristic similar to *fast retransmit* by retransmitting a gap only once it has received three SACK options indicating this gap. It should be noted that the SACK option does not supersede the *acknowledgement number* of the TCP header. A TCP sender can only remove data from its sending buffer once they have been acknowledged by TCP's cumulative acknowledgements. This design was chosen for two reasons. First, it allows the receiver to discard parts of its receiving buffer when it is running out of memory without loosing data. Second, as the SACK option is not transmitted reliably, the cumulative acknowledgements are still required to deal with losses of *ACK* segments carrying only SACK information. Thus, the SACK option only serves as a hint to allow the sender to optimise its retransmissions.

3.11.6 TCP connection release

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

- graceful connection release, where each TCP user can release its own direction of data transfer after having transmitted all data
- abrupt connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection (e.g. because the remote host does not reply anymore or due to lack of resources)

The abrupt connection release mechanism is very simple and relies on a single segment having the *RST* bit set. A TCP segment containing the *RST* bit can be sent for the following reasons :

- a non-*SYN* segment was received for a non-existing TCP connection **RFC 793**
- by extension, some implementations respond with an *RST* segment to a segment that is received on an existing connection but with an invalid header **RFC 3360**. This causes the corresponding connection to be closed and has caused security attacks **RFC 4953**
- by extension, some implementations send an *RST* segment when they need to close an existing TCP connection (e.g. because there are not enough resources to support this connection or because the remote host is considered to be unreachable). Measurements have shown that this usage of TCP *RST* is widespread *[AW05]*

When an *RST* segment is sent by a TCP entity, it should contain the current value of the *sequence number* for the connection (or 0 if it does not belong to any existing connection) and the *acknowledgement number* should be set to the next expected in-sequence *sequence number* on this connection.

Note: TCP *RST* wars

TCP implementers should ensure that two TCP entities never enter a TCP *RST* war where host *A* is sending a *RST* segment in response to a previous *RST* segment that was sent by host *B* in response to a TCP *RST* segment sent by host *A* ... To avoid such an infinite exchange of *RST* segments that do not carry data, a TCP entity is *never* allowed to send a *RST* segment in response to another *RST* segment.

The normal way of terminating a TCP connection is by using the graceful TCP connection release. This mechanism uses the *FIN* flag of the TCP header and allows each host to release its own direction of data transfer. As for the *SYN* flag, the utilisation of the *FIN* flag in the TCP header consumes one sequence number. The figure *FSM for TCP connection release* shows the part of the TCP FSM used when a TCP connection is released.

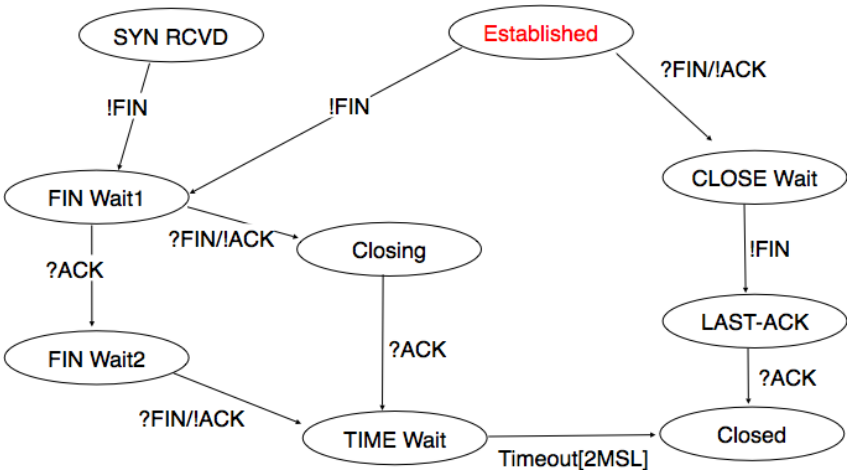


Fig. 3.33: FSM for TCP connection release

Starting from the *Established* state, there are two main paths through this FSM.

The first path is when the host receives a segment with sequence number *x* and the *FIN* flag set. The utilisation of the *FIN* flag indicates that the byte before *sequence number x* was the last byte of the byte stream sent by the remote host. Once all of the data has been delivered to the user, the TCP entity sends an *ACK* segment whose *ack* field is set to $(x+1) \pmod{2^{32}}$ to acknowledge the *FIN* segment. The *FIN* segment is subject to the same retransmission mechanisms as a normal TCP segment. In particular, its transmission is protected by the retransmission timer. At this point, the TCP connection enters the *CLOSE_WAIT* state. In this state, the host can still send data to the remote host. Once all its data have been sent, it sends a *FIN* segment and enter the *LAST_ACK* state. In this state, the TCP entity waits for the acknowledgement of its *FIN* segment. It may still retransmit unacknowledged data segments e.g. if the retransmission timer expires. Upon reception of the acknowledgement for the *FIN* segment, the TCP connection is completely closed and its *TCB* can be discarded.

The second path is when the host has transmitted all data. Assume that the last transmitted sequence number is z. Then, the host sends a *FIN* segment with sequence number $(z + 1) \pmod{2^{32}}$ and enters the *FIN_WAIT1* state. It this state, it can retransmit unacknowledged segments but cannot send new data segments. It waits for an acknowledgement of its *FIN* segment (i.e. sequence number $(z+1) \pmod{2^{32}}$), but may receive a *FIN* segment sent by the remote host. In the first case, the TCP connection enters the *FIN_WAIT2* state. In this state, new data segments from the remote host are still accepted until the reception of the *FIN* segment. The acknowledgement for this *FIN* segment is sent once all data received before the *FIN* segment have been delivered to the user and the connection enters the *TIME_WAIT* state. In the second case, a *FIN* segment is received and the connection enters the *Closing* state once all data received from the remote host have been delivered to the user. In this state, no new data segments can be sent and the host waits for an acknowledgement of its *FIN* segment before entering the *TIME_WAIT* state.

The *TIME_WAIT* state is different from the other states of the TCP FSM. A TCP entity enters this state after having sent the last *ACK* segment on a TCP connection. This segment indicates to the remote host that all the data that it has sent have been correctly received and that it can safely release the TCP connection and discard the corresponding *TCB*. After having sent the last *ACK* segment, a TCP connection enters the *TIME_WAIT* and remains in this state for $2 * MSL$ seconds. During this period, the TCB of the connection is maintained. This ensures that the TCP entity that sent the last *ACK* maintains enough state to be able to retransmit this segment if this *ACK* segment is lost and the remote host retransmits its last *FIN* segment or another one. The delay of $2 * MSL$ seconds ensures that any duplicate segments on the connection would be handled correctly without causing the transmission of an *RST* segment. Without the *TIME_WAIT* state and the $2 * MSL$ seconds delay, the connection release would not be graceful when the last *ACK* segment is lost.

Note: TIME_WAIT on busy TCP servers

The $2 * MSL$ seconds delay in the *TIME_WAIT* state is an important operational problem on servers having thousands of simultaneously opened TCP connections *[FTY99]*. Consider for example a busy web server that processes 10.000 TCP connections every second. If each of these connections remain in the *TIME_WAIT* state for 4 minutes, this implies that the server would have to maintain more than 2 million TCBs at any time. For this reason, some TCP implementations prefer to perform an abrupt connection release by sending a *RST* segment to close the connection *[AW05]* and immediately discard the corresponding *TCB*. However, if the *RST* segment is lost, the remote host continues to maintain a *TCB* for a connection no longer exists. This optimisation reduces the number of TCBs maintained by the host sending the *RST* segment but at the potential cost of increased processing on the remote host when the *RST* segment is lost.

3.12 The Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) **RFC 4960** was defined in the late 1990s, early 2000s as an alternative to the Transmission Control Protocol. The initial design of SCTP was motivated by the need to efficiently support signaling protocols that are used in Voice over IP networks. These signaling protocols allow to create, control and terminate voice calls. They have different requirements than regular applications like email, http that are well served by TCP's bytestream service.

One of the first motivations for SCTP was the need to efficiently support multihomed hosts, i.e. hosts equipped with two or more network interfaces. The Internet architecture and TCP in particular were not designed to handle efficiently such hosts. On the Internet, when a host is multihomed, it needs to use several IP addresses, one per interface. Consider for example a smartphone connected to both WiFi and 3G. The smartphone uses one IP address on its WiFi interface and a different one on its 3G interface. When it establishes a TCP connection through its WiFi interface, this connection is bound to the IP address of the WiFi interface and the segments corresponding to this connection must always be transmitted through the WiFi interface. If the WiFi interface is not anymore connected to the network (e.g. because the smartphone user moved), the TCP connection stops and need to be explicitly reestablished by the application over the 3G interface. SCTP was designed to support seamless failover from one interface to another during the lifetime of a connection. This is a major change compared to TCP [1].

[1] Recently, the IETF approved the Multipath TCP extension **RFC 6824** that allows TCP to efficiently support multihomed hosts. A detailed presentation of Multipath TCP is outside the scope of this document, but may be found in *[RIB2013]* and on http://www.multipath-tcp.org

router or the address of the DNS resolver. The DHCP reply also specifies the lifetime of the address allocation. This forces the host to renew its address allocation once it expires. Thanks to the limited lease time, IP addresses are automatically returned to the pool of addresses when hosts are powered off.

Both SLAAC and DHCPv6 can be extended to provide additional information beyond the IPv6 prefix/address. For example, **RFC 6106** defines options for the ICMPv6 ND message that can carry the IPv6 address of the recursive DNS resolver and a list of default domain search suffixes. It is also possible to combine SLAAC with DHCPv6. **RFC 3736** defines a stateless variant of DHCPv6 that can be used to distribute DNS information while SLAAC is used to distribute the prefixes.

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

3.16 Routing in IP networks

In a large IP network such as the global Internet, routers need to exchange routing information. The Internet is an interconnection of networks, often called domains, that are under different responsibilities. As of this writing, the Internet is composed on more than 40,000 different domains and this number is still growing [1]. A domain can be a small enterprise that manages a few routers in a single building, a larger enterprise with a hundred routers at multiple locations, or a large Internet Service Provider managing thousands of routers. Two classes of routing protocols are used to allow these domains to efficiently exchange routing information.

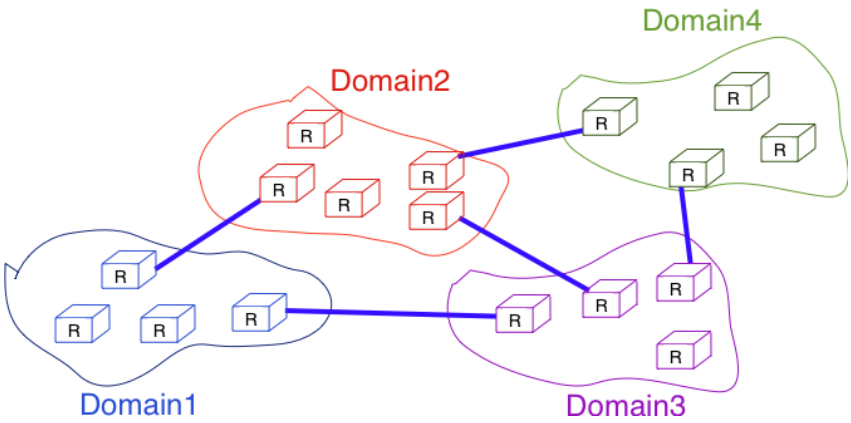


Fig. 3.60: Organisation of a small Internet

The first class of routing protocols are the *intradomain routing protocols* (sometimes also called the interior gateway protocols or *IGP*). An intradomain routing protocol is used by all routers inside a domain to exchange routing information about the destinations that are reachable inside the domain. There are several intradomain routing protocols. Some domains use *RIP*, which is a distance vector protocol. Other domains use link-state routing protocols such as *OSPF* or *IS-IS*. Finally, some domains use static routing or proprietary protocols such as *IGRP* or *EIGRP*.

These intradomain routing protocols usually have two objectives. First, they distribute routing information that corresponds to the shortest path between two routers in the domain. Second, they should allow the routers to quickly recover from link and router failures.

The second class of routing protocols are the *interdomain routing protocols* (sometimes also called the exterior gateway protocols or *EGP*). The objective of an interdomain routing protocol is to distribute routing information between domains. For scalability reasons, an interdomain routing protocol must distribute aggregated routing information and considers each domain as a black box.

A very important difference between intradomain and interdomain routing are the *routing policies* that are used by each domain. Inside a single domain, all routers are considered equal, and when several routes are available

[1] See http://bgp.potaroo.net/index-as.html for reports on the evolution of the number of Autonomous Systems over time.

to reach a given destination prefix, the best route is selected based on technical criteria such as the route with the shortest delay, the route with the minimum number of hops or the route with the highest bandwidth.

When we consider the interconnection of domains that are managed by different organisations, this is no longer true. Each domain implements its own routing policy. A routing policy is composed of three elements : an *import filter* that specifies which routes can be accepted by a domain, an *export filter* that specifies which routes can be advertised by a domain and a ranking algorithm that selects the best route when a domain knows several routes towards the same destination prefix. As we will see later, another important difference is that the objective of the interdomain routing protocol is to find the *cheapest* route towards each destination. There is only one interdomain routing protocol : *BGP*.

3.17 Intradomain routing

In this section, we briefly describe the key features of the two main intradomain unicast routing protocols : RIP and OSPF. The basic principles of distance vector and link-state routing have been presented earlier.

3.17.1 RIP

The Routing Information Protocol (RIP) is the simplest routing protocol that was standardised for the TCP/IP protocol suite. RIP is defined in **RFC 2453**. Additional information about RIP may be found in *[Malkin1999]*

RIP routers periodically exchange RIP messages. The format of these messages is shown below. A RIP message is sent inside a UDP segment whose destination port is set to *521*. A RIP message contains several fields. The *Cmd* field indicates whether the RIP message is a request or a response. When a router boots, its routing table is empty and it cannot forward any packet. To speedup the discovery of the network, it can send a request message to the RIP IPv6 multicast address, `FF02::9`. All RIP routers listen to this multicast address and any router attached to the subnet will reply by sending its own routing table as a sequence of RIP messages. In steady state, routers multicast one of more RIP response messages every 30 seconds. These messages contain the distance vectors that summarize the router's routing table. The current version of RIP is version 2 defined in **RFC 2453** for IPv4 and **RFC 2080** for IPv6.

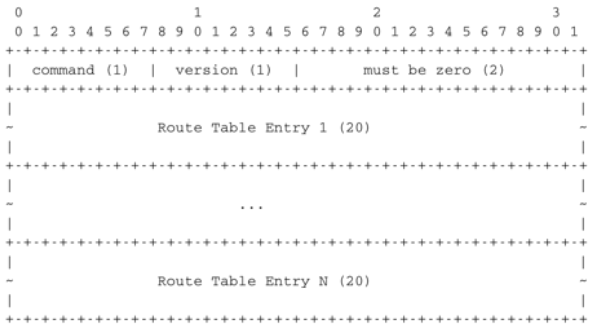


Fig. 3.61: The RIP message format

Each RIP message contains a set of route entries. Each route entry is encoded as a 20 bytes field whose format is shown below. RIP was initially designed to be suitable for different network layer protocols. Some implementations of RIP were used in XNS or IPX networks **RFC 2453**. The format of the route entries used by **RFC 2080** is shown below. *Plen* is the length of the subnet identifier in bits and the metric is encoded as one byte. The maximum metric supported by RIP is *15*.

Note: A note on timers

The first RIP implementations sent their distance vector exactly every 30 seconds. This worked well in most networks, but some researchers noticed that routers were sometimes overloaded because they were processing too many distance vectors at the same time *[FJ1994]*. They collected packet traces in these networks and found that after some time the routers' timers became synchronised, i.e. almost all routers were sending their distance

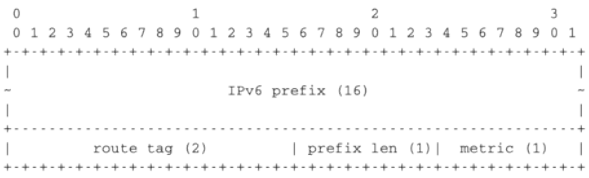


Fig. 3.62: Format of the RIP IPv6 route entries

vectors at almost the same time. This synchronisation of the transmission times of the distance vectors caused an overload on the routers' CPU but also increased the convergence time of the protocol in some cases. This was mainly due to the fact that all routers set their timers to the same expiration time after having processed the received distance vectors. Sally Floyd and Van Jacobson proposed in *[FJ1994]* a simple solution to solve this synchronisation problem. Instead of advertising their distance vector exactly after 30 seconds, a router should send its next distance vector after a delay chosen randomly in the [15,45] interval **RFC 2080**. This randomisation of the delays prevents the synchronisation that occurs with a fixed delay and is now a recommended practice for protocol designers.

3.17.2 OSPF

Link-state routing protocols are used in IP networks. Open Shortest Path First (OSPF), defined in **RFC 2328**, is the link state routing protocol that has been standardised by the IETF. The last version of OSPF, which supports IPv6, is defined in **RFC 5340**. OSPF is frequently used in enterprise networks and in some ISP networks. However, ISP networks often use the IS-IS link-state routing protocol *[ISO10589]* , which was developed for the ISO CLNP protocol but was adapted to be used in IP **RFC 1195** networks before the finalisation of the standardisation of OSPF. A detailed analysis of ISIS and OSPF may be found in *[BMO2006]* and *[Perlman2000]*. Additional information about OSPF may be found in *[Moy1998]*.

Compared to the basics of link-state routing protocols that we discussed in section *Link state routing*, there are some particularities of OSPF that are worth discussing. First, in a large network, flooding the information about all routers and links to thousands of routers or more may be costly as each router needs to store all the information about the entire network. A better approach would be to introduce hierarchical routing. Hierarchical routing divides the network into regions. All the routers inside a region have detailed information about the topology of the region but only learn aggregated information about the topology of the other regions and their interconnections. OSPF supports a restricted variant of hierarchical routing. In OSPF's terminology, a region is called an *area*.

OSPF imposes restrictions on how a network can be divided into areas. An area is a set of routers and links that are grouped together. Usually, the topology of an area is chosen so that a packet sent by one router inside the area can reach any other router in the area without leaving the area [2] . An OSPF area contains two types of routers **RFC 2328**:

- Internal router : A router whose directly connected networks belong to the area
- Area border routers : A router that is attached to several areas.

For example, the network shown in the figure below has been divided into three areas : *area 1*, containing routers *R1*, *R3*, *R4*, *R5* and *RA*, *area 2* containing *R7*, *R8*, *R9*, *R10*, *RB* and *RC*. OSPF areas are identified by a 32 bit integer, which is sometimes represented as an IP address. Among the OSPF areas, *area 0*, also called the *backbone area* has a special role. The backbone area groups all the area border routers (routers *RA*, *RB* and *RC* in the figure below) and the routers that are directly connected to the backbone routers but do not belong to another area (router *RD* in the figure below). An important restriction imposed by OSPF is that the path between two routers that belong to two different areas (e.g. *R1* and *R8* in the figure below) must pass through the backbone area.

Inside each non-backbone area, routers distribute the topology of the area by exchanging link state packets with the other routers in the area. The internal routers do not know the topology of other areas, but each router knows how to reach the backbone area. Inside an area, the routers only exchange link-state packets for all destinations

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

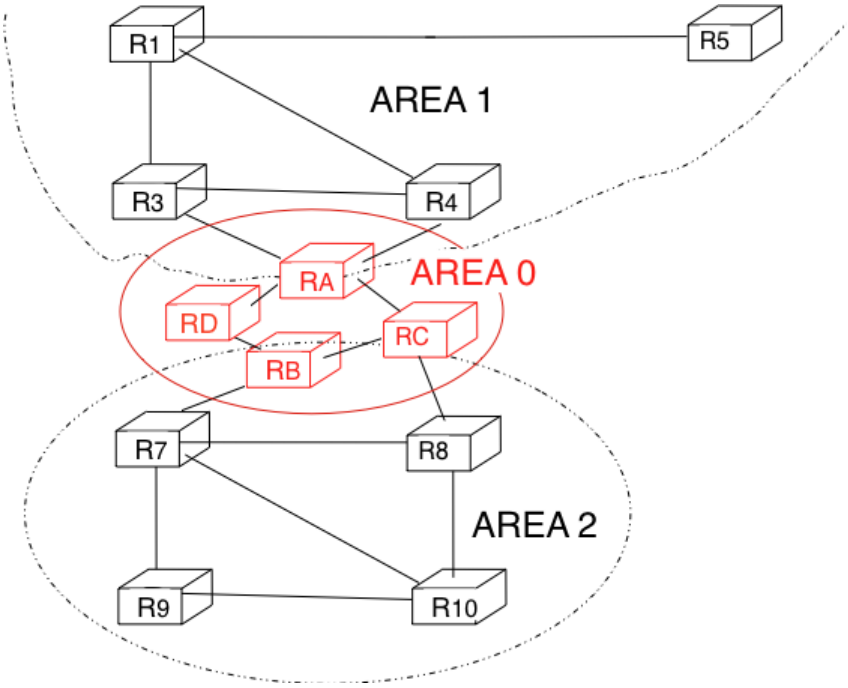


Fig. 3.63: OSPF areas

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.

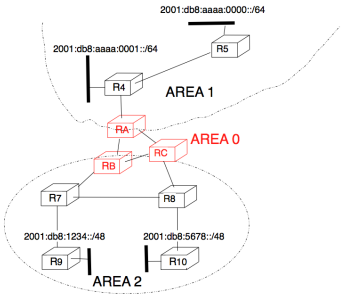


Fig. 3.64: Hierarchical routing with OSPF

Let us first consider OSPF routing inside *area 2*. All routers in the area learn a route towards *2001:db8:1234::/48* and *2001:db8:5678::/48*. The two area border routers, *RB* and *RC*, create network summary advertisements. Assuming that all links have a unit link metric, these would be:

- *RB* advertises *2001:db8:1234::/48* at a distance of *2* and *2001:db8:5678::/48* at a distance of *3*
- *RC* advertises *2001:db8:5678::/48* at a distance of *2* and *2001:db8:1234::/48* at a distance of *3*

These summary advertisements are flooded through the backbone area attached to routers *RB* and *RC*. In its routing table, router *RA* selects the summary advertised by *RB* to reach *2001:db8:1234::/48* and the summary advertised by *RC* to reach *2001:db8:5678::/48*. Inside *area 1*, router *RA* advertises a summary indicating that *2001:db8:1234::/48* and *2001:db8:5678::/48* are both at a distance of *3* from itself.

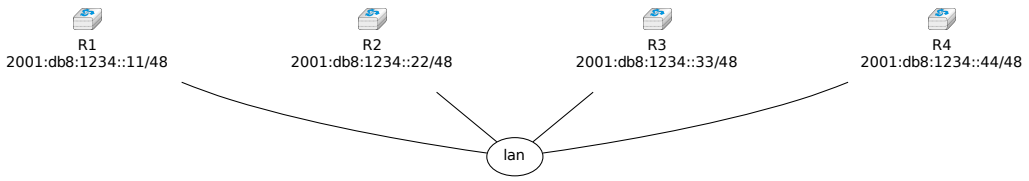
On the other hand, consider the prefixes *2001:db8:aaaa:0000::/64* and *2001:db8:aaaa:0001::/64* that are inside *area 1*. Router *RA* is the only area border router that is attached to this area. This router can create two different network summary advertisements :

- *2001:db8:aaaa:0000::/64* at a distance of *1* and *2001:db8:aaaa:0001::/64* at a distance of *2* from *RA*

- *2001:db8:aaaa:0000::/63* at a distance of *2* from *RA*

The first summary advertisement provides precise information about the distance used to reach each prefix. However, all routers in the network have to maintain a route towards *2001:db8:aaaa:0000::/64* and a route towards *2001:db8:aaaa:0001::/64* that are both via router *RA*. The second advertisement would improve the scalability of OSPF by reducing the number of routes that are advertised across area boundaries. However, in practice this requires manual configuration on the border routers.

The second OSPF particularity that is worth discussing is the support of Local Area Networks (LAN). As shown in the example below, several routers may be attached to the same LAN.



A first solution to support such a LAN with a link-state routing protocol would be to consider that a LAN is equivalent to a full-mesh of point-to-point links as if each router can directly reach any other router on the LAN. However, this approach has two important drawbacks :

1. Each router must exchange HELLOs and link state packets with all the other routers on the LAN. This increases the number of OSPF packets that are sent and processed by each router.
2. Remote routers, when looking at the topology distributed by OSPF, consider that there is a full-mesh of links between all the LAN routers. Such a full-mesh implies a lot of redundancy in case of failure, while in practice the entire LAN may completely fail. In case of a failure of the entire LAN, all routers need to detect the failures and flood link state packets before the LAN is completely removed from the OSPF topology by remote routers.

To better represent LANs and reduce the number of OSPF packets that are exchanged, OSPF handles LAN differently. When OSPF routers boot on a LAN, they elect [3] one of them as the *Designated Router (DR)* **RFC 2328**. The *DR* router *represents* the local area network, and advertises the LAN's subnet. Furthermore, LAN routers only exchange HELLO packets with the *DR*. Thanks to the utilisation of a *DR*, the topology of the LAN appears as a set of point-to-point links connected to the *DR* router.

Note: How to quickly detect a link failure ?

Network operators expect an OSPF network to be able to quickly recover from link or router failures *[VPD2004]*. In an OSPF network, the recovery after a failure is performed in three steps *[FFEB2005]* :

- the routers that are adjacent to the failure detect it quickly. The default solution is to rely on the regular exchange of HELLO packets. However, the interval between successive HELLOs is often set to 10 seconds... Setting the HELLO timer down to a few milliseconds is difficult as HELLO packets are created and processed by the main CPU of the routers and these routers cannot easily generate and process a HELLO packet every millisecond on each of their interfaces. A better solution is to use a dedicated failure detection protocol such as the Bidirectional Forwarding Detection (BFD) protocol defined in *[KW2009]* that can be implemented directly on the router interfaces. Another solution to be able to detect the failure is to instrument the physical and the datalink layer so that they can interrupt the router when a link fails. Unfortunately, such a solution cannot be used on all types of physical and datalink layers.
- the routers that have detected the failure flood their updated link state packets in the network
- all routers update their routing table

---

[3] 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.