3 - End-to-end Protocols

Sergi Robles

Sergi.Robles@uab.es

Department of Information and Communication Engineering
Universitat Autònoma de Barcelona

Computer Networks II, year 2008/09

Contingut

Contingut

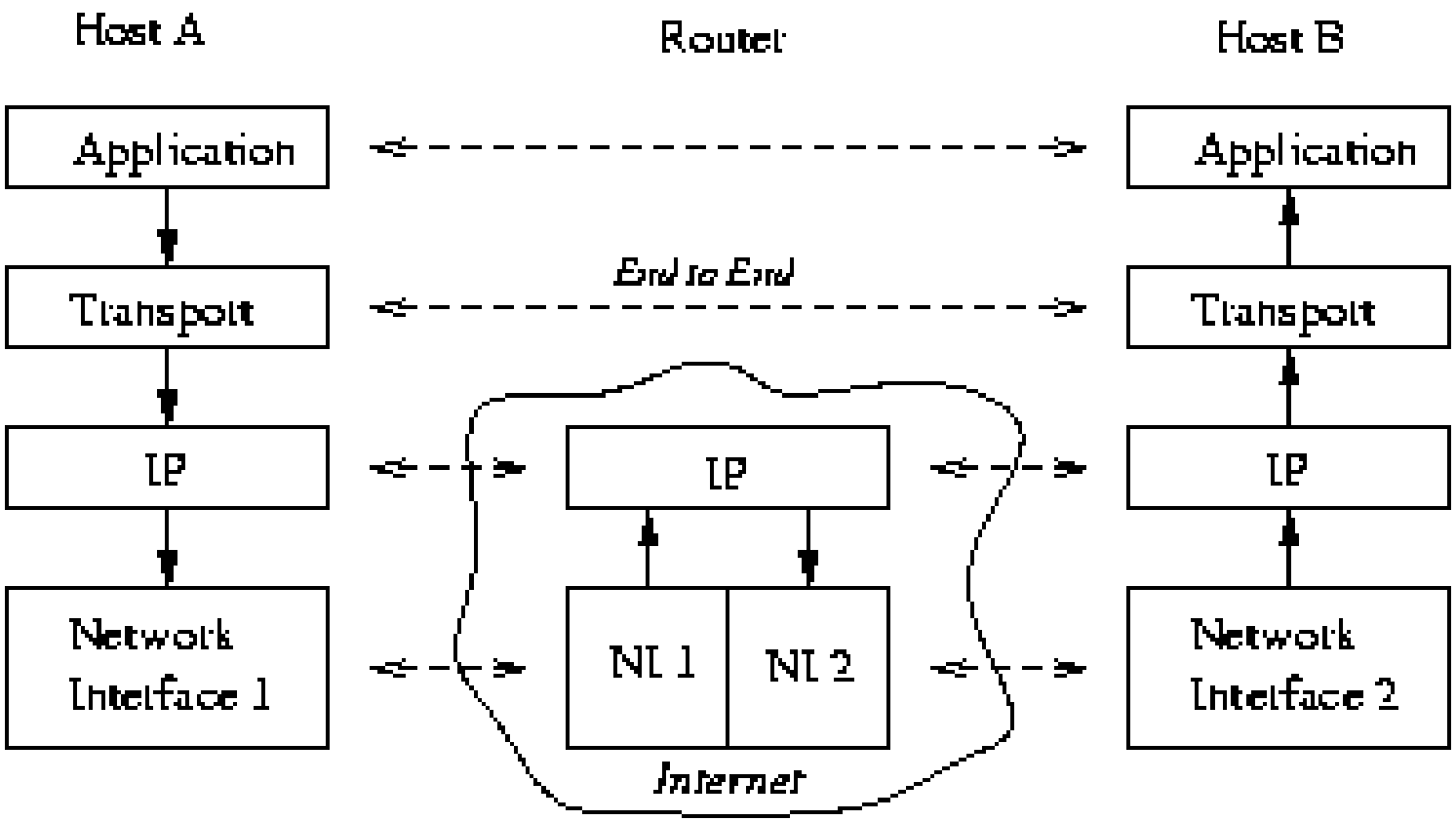
Principles of end-to-end communication

So far we have studied how the network interconnection protocol (Internet Protocol) allows a datagram to arrive from an origin to a destination (IP routing).

$\longrightarrow$ There is no information about the user or application receiving the datagram!

In this section we will see how the protocols from the TCP/IP suit distinguish between the applications which are sending and receiving messages independently.

In end-to-en protocols the communication is performed between the applications in the source and destination hosts.



The IP protocol provides an unreliable communication service. Should reliability is a requirement, it must be provided by either the application programmer or by other protocol.

TCP/IP provides two end-to-end protocols:

- Unreliable message transfer between applications (User Datagram Protocol, **UDP**)
- Reliable byte stream between applications (Transmission Control Protocol, **TCP**)

Contingut

We could consider applications as the ultimate destination for messages...

A host can be executing more than one program at a time. The IP address is not enough to identify them.

$\rightarrow$ We would need an application identifier (PID?).

Some problems arising when considering application identifiers:

- Processes are created and destroyed dynamically.
- Processes could be replaced seamlessly, without notifying the source of the communication.

Ports

- We must identify the service regardless of the specific process that is running it.
- If a program is providing more than one service, it is basic to specify which is the one required.

Ports

- We must identify the service regardless of the specific process that is running it.
- If a program is providing more than one service, it is basic to specify which is the one required.

Ports

Instead of thinking about a destination program we could think about having abstract access points at each host called **protocol ports**.

Each protocol port (or simply port) is identified by a positive integer (16 bits).

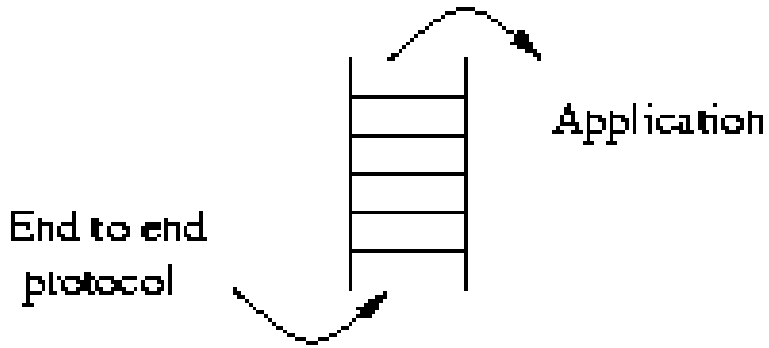
Analogy

IP Address

Port

Each Operating System provides to processes with an interface and mechanisms for using the ports.

Normally the OS assigns to the port a finite queue where all messages arriving for it are stored.



The access is usually synchronous (the process is blocked if data are read from an empty queue.).

The origin must know the IP address of the destination, and also the destination port, to send a message at transport level.

In order to receive replies, every message must include the source IP address and the source port.

There is two classes of ports:

Assigned ports Ports reserved for specific services. They are the same in all hosts. Some privileges might be required to assign them. Their range is **[1...1023]**. They are also called well-known ports.

Client application ports The assignment is dynamic, and they can be freely used. Their range is **[1024...65535]**.

There is an important triad at the transport layer level:

Port - Service - Protocol

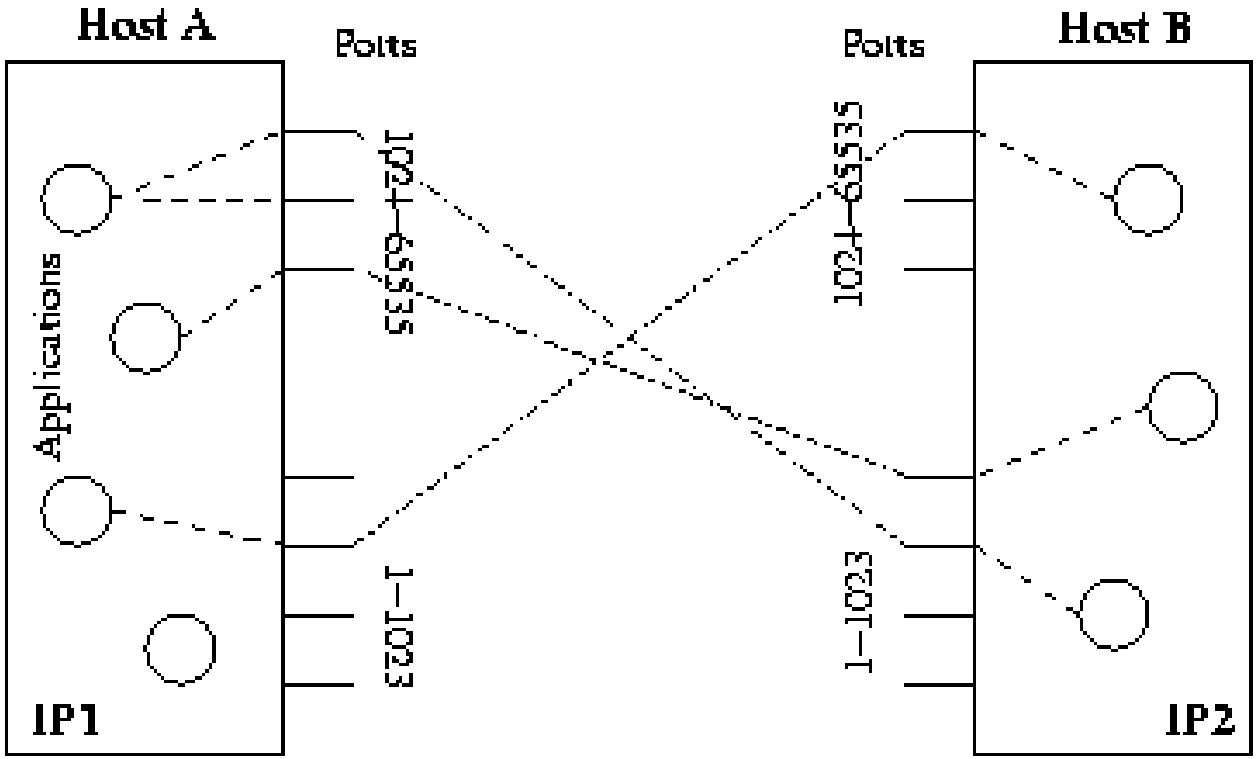
A service is provided by means of a protocol, which is associated to a certain port. Assigned ports are used for these services. This is a *de facto* standard, not a protocol obligation.

How can we know these associations?

Operating Systems normally have a table with these triads. In UNIX-like systems it is in the file

`/etc/services`

Port utilisation scheme:



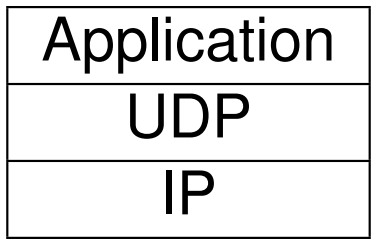
The port concept allows to send messages to applications on a particular host.

Contingut

User Datagram Protocol (UDP)

UDP

UDP (User Datagram Protocol) provides the basic mechanism used by application for sending datagrams to other applications.



→ UDP uses port numbers to distinguish the destination application on a host.

UDP services

UDP provides the same unreliable and connectionless datagram delivery service than IP:

- There are no ACK messages to make sure that the datagram has arrived.
- Messages are not ordered at arrival.
- The information flow between hosts is not controlled.

$\rightarrow$ Messages can be lost, duplicated, or arrive out of order, but messages going to different destinations in the same host are distinguished.

The responsibility of dealing with UDP problems goes to the application programmer.

Usually this is not done, since if reliability were required another protocol would be used.
There are some exceptions: TFTP, or reliability over UDP. Why?

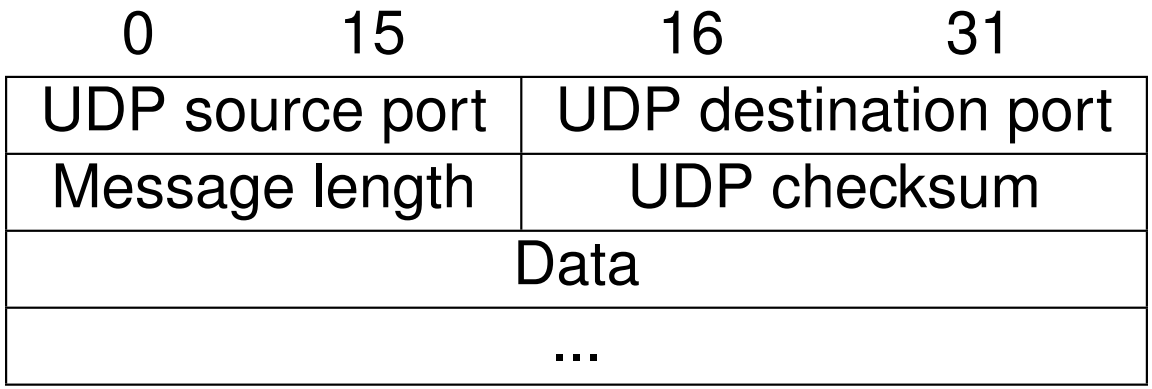
Reliability

Applications requiring reliability and using UDP are advisable in local highly reliable networks, but not in large internets. The UDP service will be as reliable as the network beneath.

Contingut

UDP Datagram

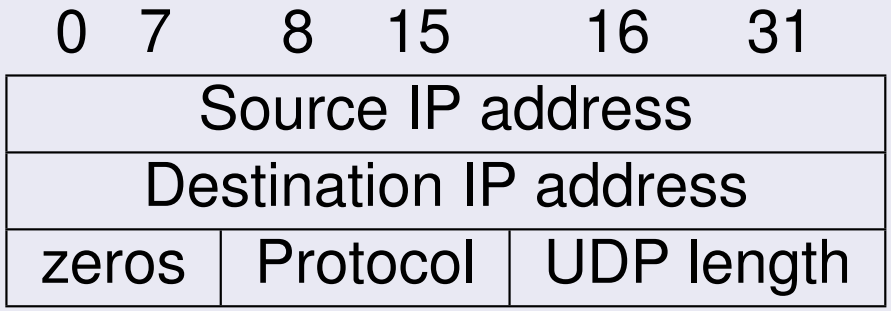
UDP messages are called User Datagrams. They have a 8 byte header and a data area:



Source and destination ports UDP protocol ports used for demultiplexing datagrams upon arrival. The source port is optional (set to zero if not used)

Message length Number of bytes of the UDP datagram, taking into account the header and user data. This will be 8 bytes at least.

UDP checksum This field is optional. If it is not used, should be set to zero. In order to calculate the checksum a pseudo-header is created (it will not be sent) for verifying the integrity of the destination data.



The protocol field in this case is 17 (UDP), which is the one in the IP header. The length of the UDP message does not include the pseudo-header.

The checksum is calculated like this: the pseudo-header is created and added before the UDP header. The checksum field is set to zero, and then the checksum calculated.

The checksum is performed on the headers and the user data field.

UDP requires to know the IP address for calculating the checksum... This is contradictory to the concept of layered protocols, isn't it?

UDP requires to know the IP address for calculating the checksum... This is contradictory to the concept of layered protocols, isn't it?

→ Totally! UDP violates the basic premise of independence of functionality on the layers of protocols. This is done for practical reasons.

It is really the UDP layer which prepares the IP datagram, and afterwards it is passed to IP which will continue filling the remaining fields.

Contingut

UDP functionality

Functions of UDP:

1. Receive a message from an application.
2. Create a user datagram.
3. Calculates the checksum (optional).
4. Pass the datagram to IP.

Multiplexing and demultiplexing operation in UDP:

- The application has to negotiate with the operating system to get a port before sending messages to UDP. The OS assigns a queue to store all messages arriving to this port.
- When a UDP message arrives the port number is used to put it in the corresponding queue. If there is no queue assigned to the port, the message is dropped and an error message is created (ICMP).
- If the queue is full the message is discarded.

UDP is a good choice is the application requires speed before reliability.

Also if independence between application is needed, generally between client and server. UDP is a stateless protocol.

Applications:

NFS, NIS, TFTP, Network Games, audio and video streaming, VoIP, among others.

Contingut

Transmission Control Protocol (TCP)

We have seen UDP so far, an unreliable messaging service.

In this section we will see a reliable stream service between applications. The protocol providing this service is **TCP** (*Transmission Control Protocol*).

TCP is designed as a general purpose protocol.

Until now, using only an unreliable message delivering service we have accepted:

- Datagrams may be destroyed or lost (transmission errors, hardware failures, ...).
- Datagrams might arrive out of order, delayed, or even duplicated.
- Sender and receiver may have different message processing speeds, being this the cause of many losses.

If we want to send many data and want reliability, we had to do a control from the application. But then:

- The same control code would be in many applications.
- Many technical details would be required to program applications.

We need general purpose mechanisms allowing the reliable transmission of bytes at the operating system level.

Properties of the reliable service

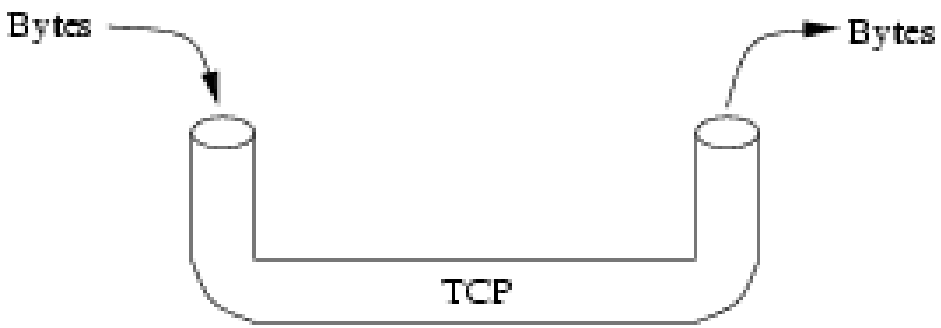
The reliable transmission service has 5 important properties:

TCP properties

1. Stream orientation.
2. Virtual connection.
3. Buffered transfer.
4. Unstructured stream.
5. Full duplex connection.

1. Stream orientation When two application programs transfer data, we think of the data as a stream of bits, divided into bytes.

   - The stream going out from a source application and the one arriving to the destination application are **always** exactly the same.



2. Virtual connection Making a stream transfer implies **establishing a connection**. The ends must agree this connection.

During transfer, protocol software verify that data is received correctly. If the communication fails, the application programs are reported.

There is always three stages:

- 1. Connection establishment
- 2. Data transfer
- 3. Connection release

3. Buffered transfer The source application passes data to the protocol software. This software decides when the data will be sent. Meanwhile, data is in a buffer.

- Writing and reading from this buffer are blocking operations: if it is full the writing will be blocked, and if it is empty the read will be blocked.
- There is a "*push*" mechanism for forcing a transfer of the data in the buffer.
- If the receiver cannot cope with the amount of data the protocol will make the sender stop.

4. Unstructured stream For the service, all is a stream of bytes. The structure of data must be provided by the application using it.

- Application programs must agree on stream format before they initiate a connection.

5. Full duplex connection The stream service allow concurrent transfer in both directions.

- Applications see two independent streams flowing in opposite directions (one can send control information and the other data).

Contingut

Mechanisms for reliability

How can be reliability provided if we only have unreliable services below?

→ There are several mechanisms providing this reliability. We will analyse the two used by the TCP protocol.

Mechanisms for reliability in TCP

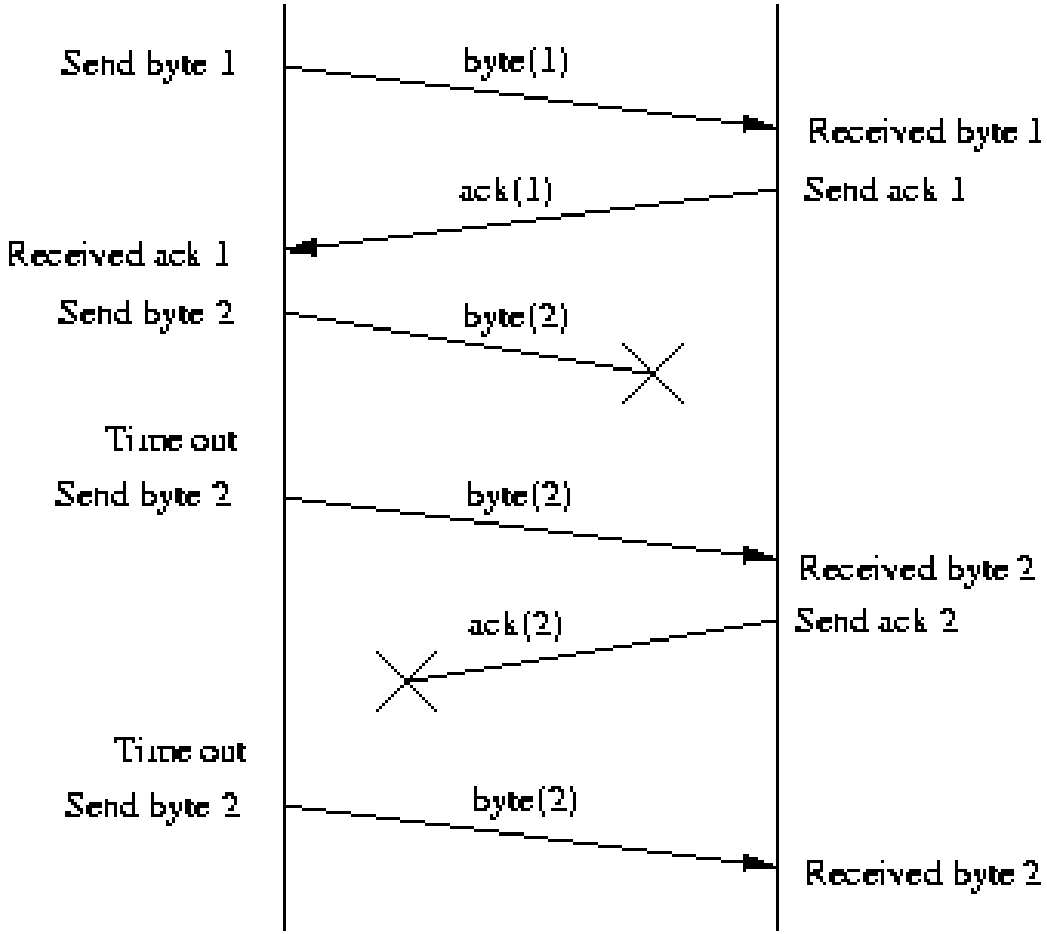
- Positive acknowledgement with retransmission
- Sliding Window

Positive acknowledgement with retransmission

Most protocols with support for reliability use this technique.

1. Data is sent.
2. If they are received, an ACK message is sent as an acknowledgement response.
3. No more data is sent until the ACK message is received.
4. If the ACK is not received after a given time, data is sent again.

To avoid duplication of messages (data and ACKs), they are numbered. The ACK-1 message indicates that the first data segment has been received.



Sliding Window

Using the positive acknowledgement with retransmission there is a loss of time: while acknowledgements are being waited nothing else is done.

$\longrightarrow$ This means a low usage of the bandwidth.

Remember we have a full duplex channel for which information can be sent and received at the same time!

There is another mechanism for a more efficient transmission: the sliding window.

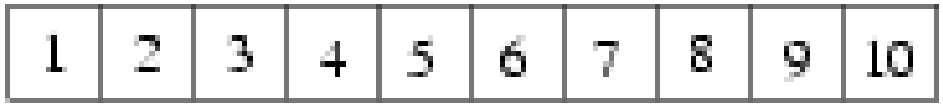
The sliding window technique is more complex than the simple positive acknowledge.

Before receiving the ACK message **more data can be sent**!

→ A better utilisation of the bandwidth will be achieved.

The sender sends data, but ignoring if the other end is receiving them or not. How data loss is controlled?

To illustrate how a sliding window operates we take the bytes to be sent as a numbered sequence of elements:



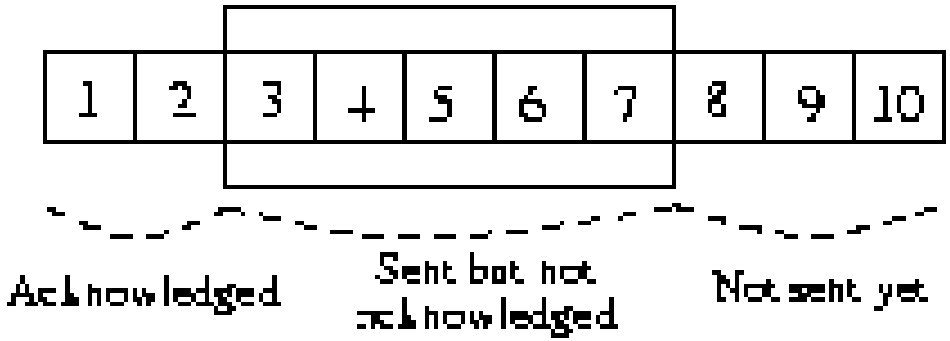
Window

A "**window**" is a subset of consecutive elements in this sequence that can be sent without waiting for acknowledgement.

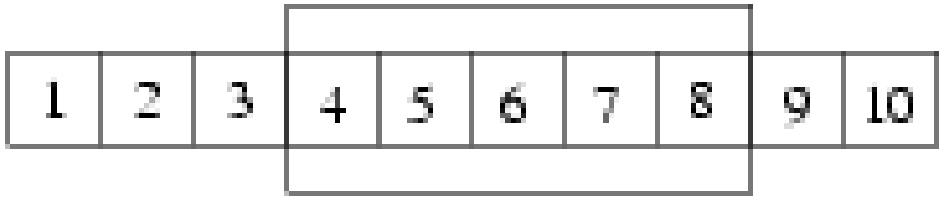
For instance, elements from 3 to 7, both included.

There are three types of elements: The already sent and acknowledged, the sent but not yet acknowledged, and those not sent yet.

The window enclose the bytes pending to be acknowledged.

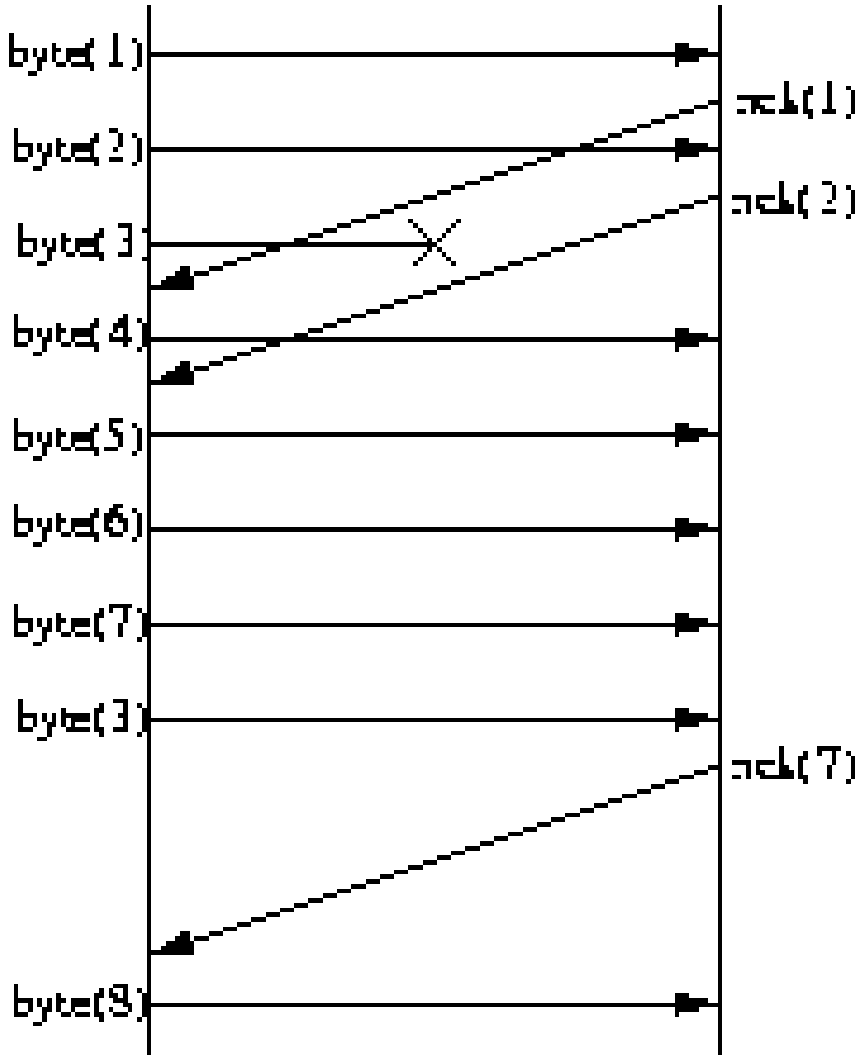


When the acknowledge corresponding to the first element arrives, the window slides admitting a new element to be sent:



The window in this case has a length of 5. The length of the window specifies how many unacknowledged elements there can be, at most.

The ACK message corresponding to the *n* unit of data indicates that all the previous elements have been received correctly.



- Both the sender and the receiver must have sliding windows to be aware of the data that has already arrived and the data that must be retransmitted.

If the sliding window size is 1, the mechanism is equivalent to the positive acknowledgement with retransmission.

The size of the window is a key parameter to better the transmission rate: if it was large enough all waiting time could be eliminated.

$\rightarrow$ The data transfer can be as fast as the network permits.

The sliding window mechanism in TCP is a bit different from we have just seen, in order to adjust to the features of this protocol:

- ACK numbering is not related to the number of datagram or segment.

  This is due to the variable size of segments and because the retransmitted segments could contain more data than the original.

  The ACK numbers in TCP are related to stream positions.

- The receiver always acknowledges the largest contiguous prefix of the stream that was correctly received.
- The acknowledgement refers always to the next byte in the stream to be received.

  This acknowledgement scheme is called accumulative.

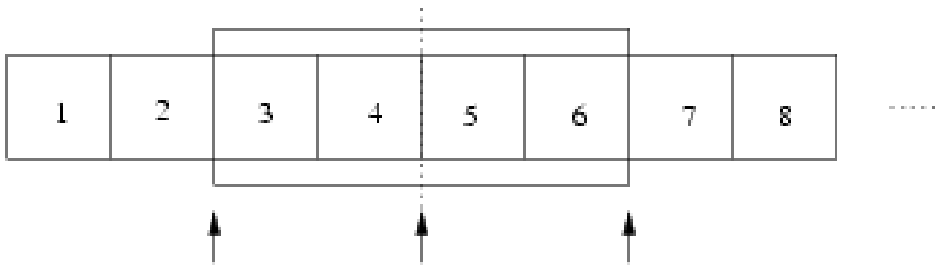
  The accumulative acknowledgement has pros and cons:
  - Pros: ACK easy to create, no ambiguity, loss does not force retransmission.
  - Cons: The sender is not aware of what has been received.

TCP stream

From the point of view of TCP the data stream is a sequence of bytes, that it divides into **segments**, usually sent in only one IP datagram.

TCP uses a specialised sliding window mechanism.

The sender keeps three pointers associated to every window: start of window (separating bytes already acknowledged), window boundary (separating the bytes that can be sent without waiting for the acknowledgement), and already sent bytes.



TCP allows to modify the window size during a transmission (**variable size**).

Every acknowledge message has a window advertisement specifying how many bytes can be received:

- If it becomes bigger, the sender increases the size of the sliding window.
- If it decreases, the sender sends bytes until the boundary of the window.

If the window reaches a size of zero bytes, the transmission is stalled.

Contingut

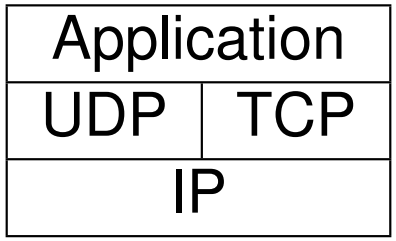
Transmission Control Protocol

TCP is the reliable stream service provided by TCP/IP. It is a protocol, not a program!

What is specified in TCP?

- Format of data and acknowledgement.
- Procedures for ensuring the correct arrival of data.
- Destination distinction in the same host.
- Recovery of errors and duplicated information.
- Establishment and closing of the connection.

TCP is placed over IP, just like UDP.



The port concept is also in TCP.

Connection

The basic abstraction in TCP is the **connection**, not the protocol port. Each connection is identified by two **endpoints**.

What is an **endpoint**?

→ Each *endpoint* corresponds to an end of the connection and is represented by a pair of integers `(host,port)`, where `host` is the IP address and `port` is the TCP port in this host.

Examples

- These examples are connections between a host at the Autònoma and the Google (1) and between a host at the IIIA and the Google (2):

(1): `(158.109.10.12,34521)-(216.239.33.99,80)`
(2): `(158.109.36.12,24135)-(216.239.33.99,80)`

The two connection examples could be simultaneous.

→TCP identifies the connection by the pair of endpoints: a TCP port can be shared among more than one connection in the same host.

Concurrency

TCP allows providing concurrent services to multiple simultaneous connections not requiring unique local ports.

In order to establish a connection, both ends must reach and agreement: the one receiving the connection does a **passive opening** and the one initiating it does a **active opening**.

Every connection has 4 sliding windows associated (two simultaneous connections (full duplex), with two windows each: one for sending and one for receiving).

Stages of the TCP protocol

- 1. Connection establishment.
- 2. Reliable transmission of data.
  - Numbering of bytes.
  - Acknowledge bytes.
  - Flow control with sliding windows.
  - Variable window size.
- 3. Release the connection.

Contingut

Segment

The format of the TCP segment

The basic transfer unit of TCP is the **segment.**

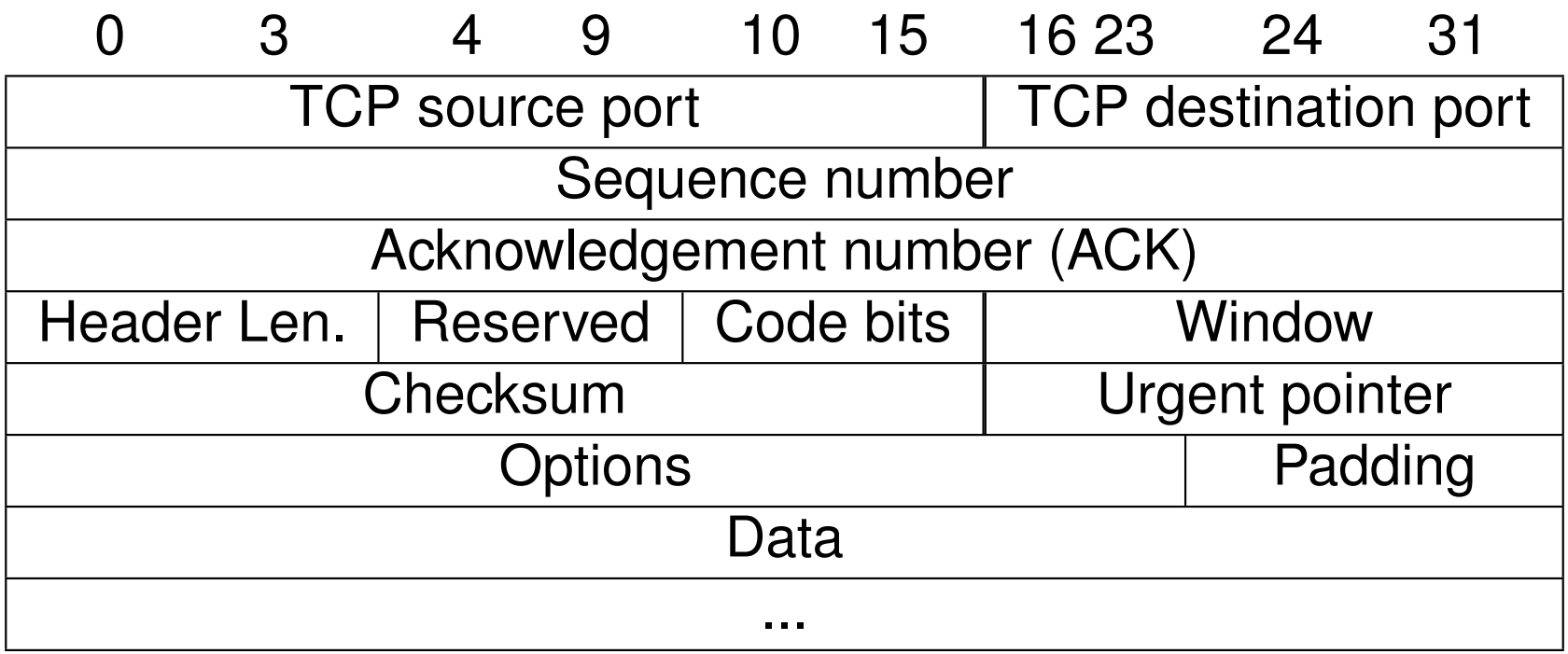
Segments are interchanged between sender and receiver for

- Establishing connection,
- Transferring data,
- Sending acknowledgements,
- Advertising window sizes,
- Closing connections.

Because TCP makes use of **piggybacking**, the acknowledgement of data in one direction can be sent on a data segment in the opposite direction.

Segment Format

The TCP segment consists of some headers and application data:



Source and destination ports These are the port numbers specifying the applications at the ends of the connection.

Sequence number Specify the position in the stream of the data in the segment.

Acknowledgement number This is the byte number expected next (related to the data stream in the opposite direction).

Header length The number of 32-bit words in the header of the segment (this is needed due to the variable length of the options).

Reserved Reserved bits for a future use.

Code bits This code specifies the content and purpose of the segment. Ordered from left to right they are:

- **URG** Set if the urgent pointer field is used.
- **ACK** Set if the acknowledgement field is used.
- **PSH** Set if segment is requesting a data PUSH (data must be delivered immediately to the application).
- **RST** Set to request a connection reset (due to sequence problems, for instance).
- **SYN** Set to synchronise sequence numbers.
- **FIN** Set to indicate the end of the data stream.

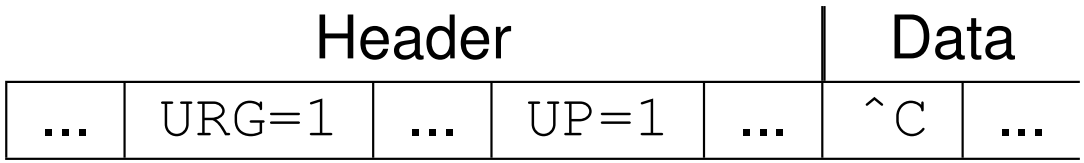
Window Advertisement of the buffer size (window size). This is in all segments, even on those carrying only an acknowledgement.

The window advertisement may be different in all segments, allowing the adaption to the current circumstances.

Checksum The checksum is calculated exactly in the same fashion as it is calculated for UDP datagrams. The required pseudo-header is the same used for UDP. The only difference is the protocol field: here it is filled with a **6** (TCP).

Urgent Pointer Although TCP transmits a seamless stream, it is useful sometimes to insert urgent data at some point (e.g. signals). When the URG bit is set, this pointer indicates the end of inserted urgent data.

$\rightarrow$ An example of the utilisation of this is during the transmission of a film on a file transfer service. If a Control-C is sent, we want the transmission to be aborted at once, not later!



Options TCP options. The most used option is the negotiation of the **Maximum Segment Size** (**MSS**). $0 \leq segment \leq MSS$

Segments can be of different lengths, but the ends must agree a maximum size. This size is very important to achieve a good utilisation of the network bandwidth.

- If it is very small, there will be an underutilisation of the network bandwidth.
- If it is very large, there will be a lot of fragmentation in IP. This will lead to problems, because if a segment's fragment is lost, the segment can not be restored.

Options (cont.)

- If both ends of the communication are in the same network, it will be good to have a MSS such that the datagram containing it is as big as the MTU.
- If they are not in the same network then either they find the smallest MTU of the networks in between or the choose the standard size of 536 bytes (576 less IP and TCP headers).

Which is the optimal MSS?

Options (cont.)

- If both ends of the communication are in the same network, it will be good to have a MSS such that the datagram containing it is as big as the MTU.
- If they are not in the same network then either they find the smallest MTU of the networks in between or the choose the standard size of 536 bytes (576 less IP and TCP headers).

Which is the optimal MSS?

Optimal MSS

$\rightarrow$ In theory, it is a size that makes the IP datagram to be as large as possible without requiring fragmentation at any point of the path between the ends of the communication.

Options (cont.)

The optimal MSS is hard to calculate, due to:

- Most TCP implementations lack from MTU discovery mechanisms.
- Paths between hosts may change dynamically.
- It also depends on the headers of the protocols at lower levels.

In practise, sender and receiver send each other their respective MTU, and the smallest is used.

Contingut

Connection establishment

Battle of Hannibal's Carthaginians against the Romans in the Alps (*circa* 217BC)

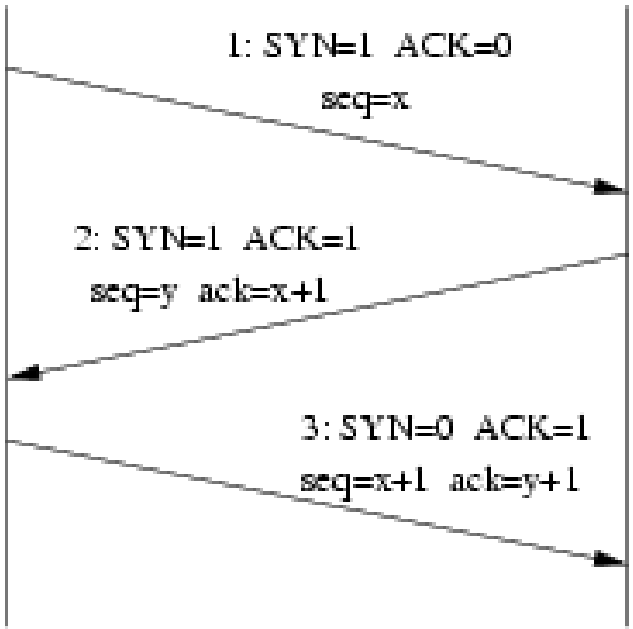
Connection establishment

To establish a connection, TCP solves the problem by using a **three way handshake**:

3-way Handshake

- 1. The first segment sent has the SYN bit set, and the ACK bit not set.
- 2. The second segment of the establishment has the SYN and ACK bits set, specifying an acknowledgement of the first segment.
- 3. The last segments has only the ACK bit set and it is used to inform the destination that an agreement has been reached and therefore the connection is established.

During the establishment of the TCP connection the sequence numbers to be used in both directions are agreed.



$\rightarrow$ The sequence numbers are chosen randomly to avoid confusion with other connections. $x, y \in \{0, 2^{32} - 1\}$

Normally one end of the connection passively waits for the establishment of the connection and the other initiates it.

→ The handshake protocol is also designed for accepting an establishment when both ends attempt to initiate it simultaneously!

The connection can be established from either end (or even from both).

Connection control

Once the connection has been established, data can flow in both directions. **There is no master and slave**.

The three-way handshake accomplishes two important functions:

- It guarantees that both sides of the communication are ready to receive data (and they know that the other part is ready as well).
- It allows both parts to agree on initial sequence numbers.

The initial sequence numbers are chosen at random, and they are used to identify the bytes of the streams.

$\rightarrow$ Sequence numbers cannot be always the same!! This would arise problems if a connection is re-started after an unexpected closing.

Segments used during the handshake are the same as the segments used along the connection.

→ These segments can carry data too!!

How is it possible to send data is the connection is not yet established?

Segments used during the handshake are the same as the segments used along the connection.

→ These segments can carry data too!!

How is it possible to send data is the connection is not yet established?

→ TCP keeps these data until the establishment has succeed.

Connection protocol analysis

- If the first segment (initial SYN) or the second (first ACK) is lost nothing happens: → after a while the segment will be **sent again**.
- If the segment is duplicated there is no problem neither: → with two segments with the same sequence number, **one is discarded**.
- If the second ACK is lost (third step of the handshake), the second SYN will be **sent again**.

Connection closing

There are two ways of closing a connection: with the closing operation (gracefully), or sending a reset. The first way should be the usual.

1- Graceful termination protocol

Because in a TCP connection the are two simultaneous streams involved the termination is performed separately.

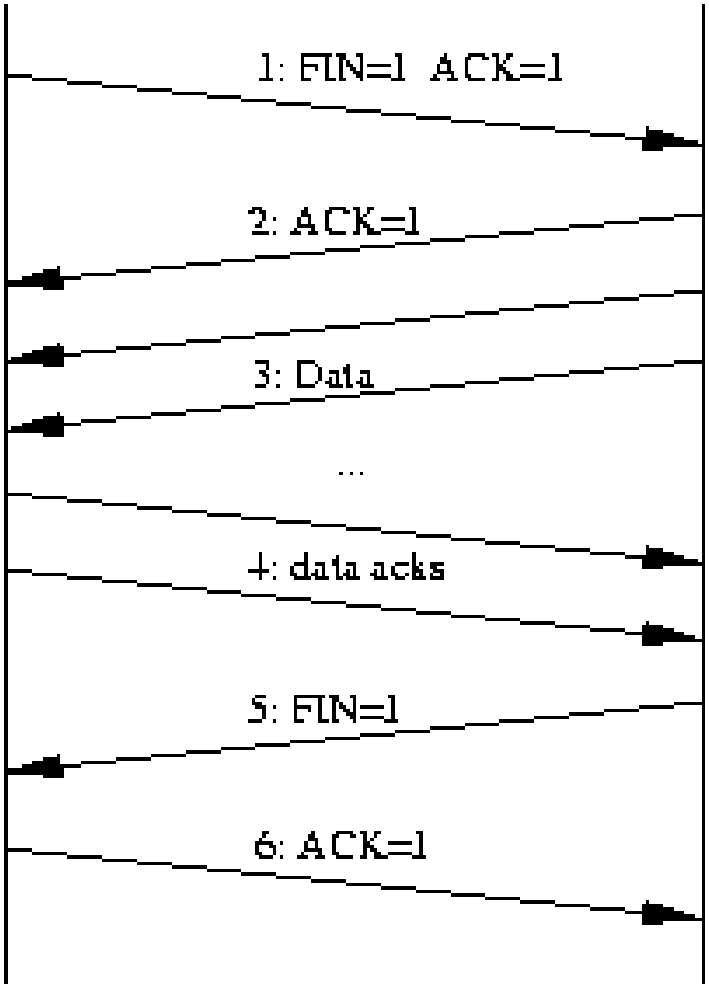
The segments to indicate the closing of the connection have the FIN bit set.

- When a data stream is over the connection in this direction is closed: the last ACK is waited for and a FIN is sent.

  After the receiving of the segment with the FIN tcp, TCP will refuse all data segments coming from this direction of the communication.

- The other end of the connection sends an ACK, finishes sending the pending data (waiting the acknowledgements) and finally sends a FIN. The connection is over when the ACK of this FIN is received.

Connection termination scheme



- In between the reception of the first FIN and the sending of the second one all pending data of the stream in the opposite direction are sent.
- Between the first FIN and the second there is, at least, one ACK without FIN.

Analysis of the termination protocol

- If a segment with the FIN bit set is lost, or if the ACK corresponding to the first FIN is lost, nothing happens: → After a while the segment **is sent again**.
- If the segment is duplicated it is not important also: → duplicated segments are always **discarded**
- If the last ACK is lost (last segment of the connection closing)... is the FIN sent again?.

How do we know that the other end has received the last ACK??

→ Problem of Carthaginians and Romans.

Abrupt termination

Connections should be closed with the closing procedure described above.

There could be situation in which it is necessary to force an immediate shutdown of the connection: → The reset (RST) bit is used for this.

Abrupt termination

- When an end sends a segment with the RST bit set, the other side shuts down the connection immediately.
- Both simultaneous TCP streams are broken and all the resources in use are released.
- TCP informs the application that there has been a *reset*.

Example of a full connection

This example shown a connection to an echo service in TCP (well-known port 7). The client has sent the word "`Hello!`" and the server has replied with the same word.

```
32833 > 7 [SYN] Seq=4256640492 Ack=0 Win=6024 Len=0
7 > 32833 [SYN, ACK] Seq=4249602681 Ack=4256640493 Win=5988 Len=0
32833 > 7 [ACK] Seq=4256640493 Ack=4249602682 Win=6024 Len=0
32833 > 7 [PSH, ACK] Seq=4256640493 Ack=4249602682 Win=6024 Len=7
7 > 32833 [ACK] Seq=4249602682 Ack=4256640500 Win=5988 Len=0
7 > 32833 [PSH, ACK] Seq=4249602682 Ack=4256640500 Win=5988 Len=7
32833 > 7 [ACK] Seq=4256640500 Ack=4249602689 Win=6024 Len=0
32833 > 7 [FIN, ACK] Seq=4256640500 Ack=4249602689 Win=6024 Len=0
7 > 32833 [FIN, ACK] Seq=4249602689 Ack=4256640501 Win=5988 Len=0
32833 > 7 [ACK] Seq=4256640501 Ack=4249602690 Win=6024 Len=0
```

Contingut

Classic attacks for TCP

SYN flooding

Denial of Service (DoS) attack.

Many connection establishment requests (SYN segments) are sent to the same host. The receiver has a limited space to keep simultaneous requests. When this space is all used, no more requests are accepted.

RST flooding

DoS and finishes active connections.

Segments with the RST bit set are sent to a host. The established connections are finished at once. If the attack is continuous no new connections can be established.

Session hijacking

A session (or a X window) can be hijacked.

By analysing TCP traffic it is possible to see the sequence numbers used by a connection. This attack consists of a third party going on with an already established connection, injecting any data.

It is normally used to execute arbitrary commands in authenticated sessions.

IP Spoofing

IP address forgery.

It allows a remote attacker, and without the need of a sniffer, to pretend it has a local address.

It can be used for executing trusted services (R commands) or to go across firewalls..

It requires sequence number “predictability”.

Other attacks

Smurf

Fraggle

Snork

Contingut

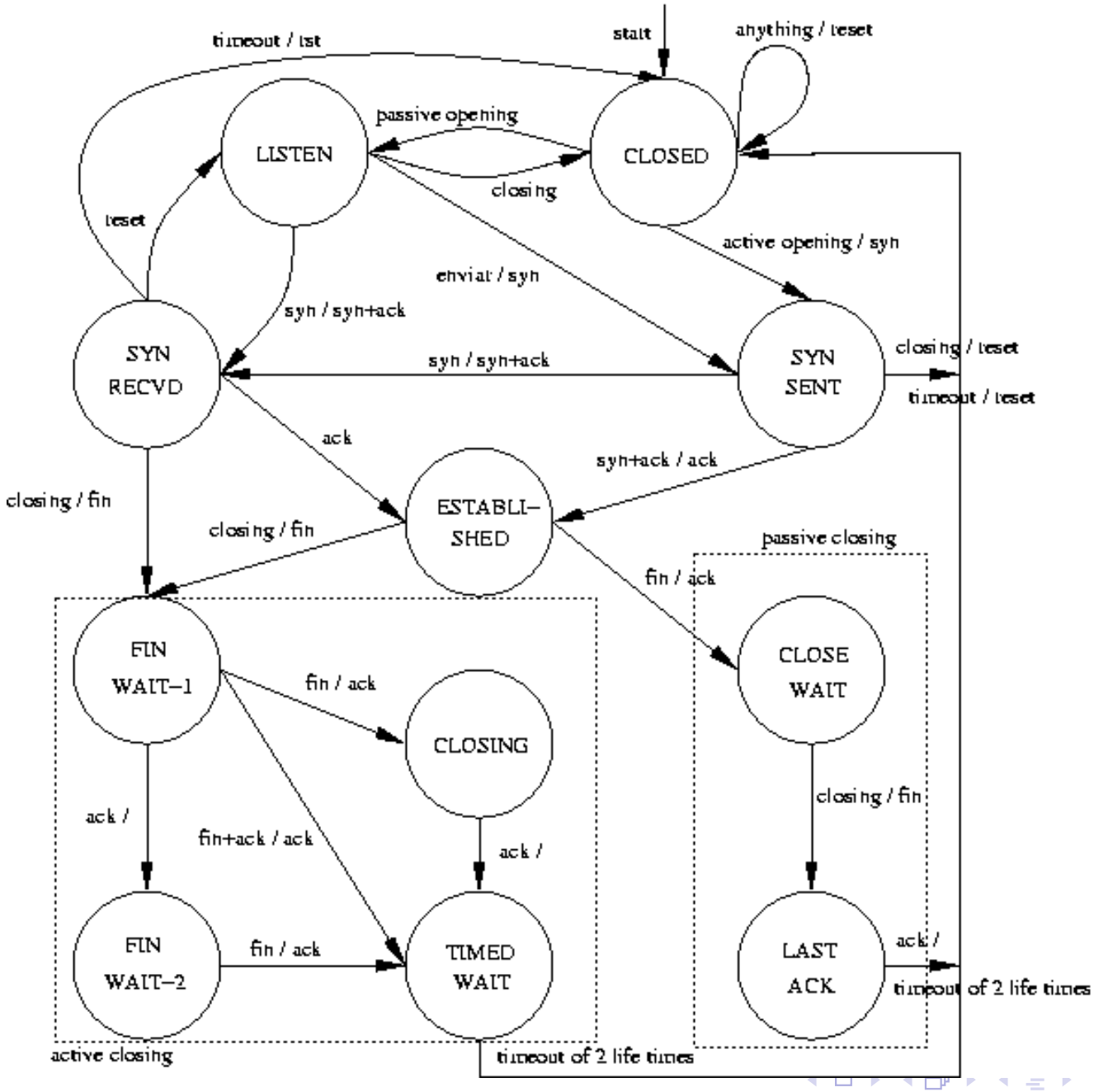
TCP state machine

Like most protocols, the operation of TCP can be easily modelled with a Finite State Machine.

11 states FSM of TCP

CLOSED,	**ESTABLISHED**,	**SYN RECVD**,	**SYN SENT**,
LISTEN,	**CLOSE WAIT**,	**LAST ACK**,	**FIN WAIT-1**,
CLOSING,	**FIN WAIT-2**,	**TIMED WAIT**.	

The initial state is CLOSED.


timeout / 1st
start
anything / reset
passive opening
LISTEN
CLOSED
closing
reset
active opening / syn
enviat / syn
syn / syn+ack
SYN RECVD
syn / syn+ack
SYN SENT
closing / reset
timeout / reset
ack
syn+ack / ack
closing / fin
ESTABLI–SHED
closing / fin
passive closing
fin / ack
FIN WAIT–1
fin / ack
CLOSING
CLOSE WAIT
ack /
fin+ack / ack
ack /
closing / fin
FIN WAIT–2
fin / ack
TIMED WAIT
LAST ACK
ack /
timeout of 2 life times
active closing
timeout of 2 life times

Contingut

Applications

netstat

The application **netstat** shows the current connections in a host together with the TCP state in which they are and the bytes in the queues:

```
# netstat -tan
Active Internet connections (servers and established)
```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:6000	0.0.0.0:*	LISTEN
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:6010	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:22	127.0.0.1:3276	ESTAB.
tcp	0	0	127.0.0.1:3276	127.0.0.1:22	ESTAB.

TCP fingerprint

The TCP standard does not specify what to do in many situations. By observing the response of a host to these situations it is possible to find out what Operating System it is running.

```
# TEST DESCRIPTION:
# Tseq is the TCP sequenceability test
# T1 is a SYN packet with a bunch of TCP options to open
port
# T2 is a NULL packet w/options to open port
# T3 is a SYN|FIN|URG|PSH packet w/options to open port
# T4 is an ACK to open port w/options
# T5 is a SYN to closed port w/options
# T6 is an ACK to closed port w/options
# T7 is a FIN|PSH|URG to a closed port w/options
# PU is a UDP packet to a closed port
```

The `nmap` program, besides performing a myriad other things, analyses a hosts and shows its TCP fingerprint and therefore the OS it is running in most cases.

```
# nmap -O -v 127.0.0.1
Starting nmap V. 2.54BETA31 ( www.insecure.org/nmap/ )
Host marisma (127.0.0.1) appears to be up ...  good.
For OSScan assuming that port 22 is open and port 1 is
closed and neither are firewalled
Interesting ports on marisma (127.0.0.1):
(The 1552 ports scanned but not shown below are in
state:  closed)
Port State Service
22/tcp open ssh
6000/tcp open X11
Remote operating system guess:  Linux Kernel 2.4.0 -
2.4.17 (X86)
TCP Sequence Prediction:  Class=random positive incs.
Difficulty=3384336 (Good luck!)
Nmap run completed - 1 IP address (1 host up) scanned in
1 second
```