Transport Protocols

Claude Chaudet
Tel : 71 51
Claude.Chaudet@enst.fr

Outline

- TCP, UDP: Applicative Multiplexing
  - Ports number
- TCP: End-to-end reliability
  - End-to-end acknowledgments
- Opening and closing sessions
- TCP: Congestion control

Standard Transport Protocols

- Several transport protocols exist in the Internet world

- UDP (User Datagram Protocol)
  - minimalistic, almost only port numbers
- TCP (Transmission Control Protocol)
  - Connected service: reliable, session management, segments order preserved
  - Congestion control

- DCCP (Datagram Congestion Control Protocol)
  - Congestion control over UDP
- SCTP (Stream Control Transmission Protocol)
  - Configurable Transport protocol

TCP, UDP: Applicative Multiplexing

TELECOM ParisTech

Applicative multiplexing

- Lower layers (physical, link, network) are there to carry a packet from a source to a destination.
- Transport layer does the rest, once the packet has arrived
- Last level of addressing: **port number**
  - Identifies an application / a service

Service	HTTP	FTP	SMTP	ssh
Application	apache	ftpd	exim	sshd
Port	80	21	25	22

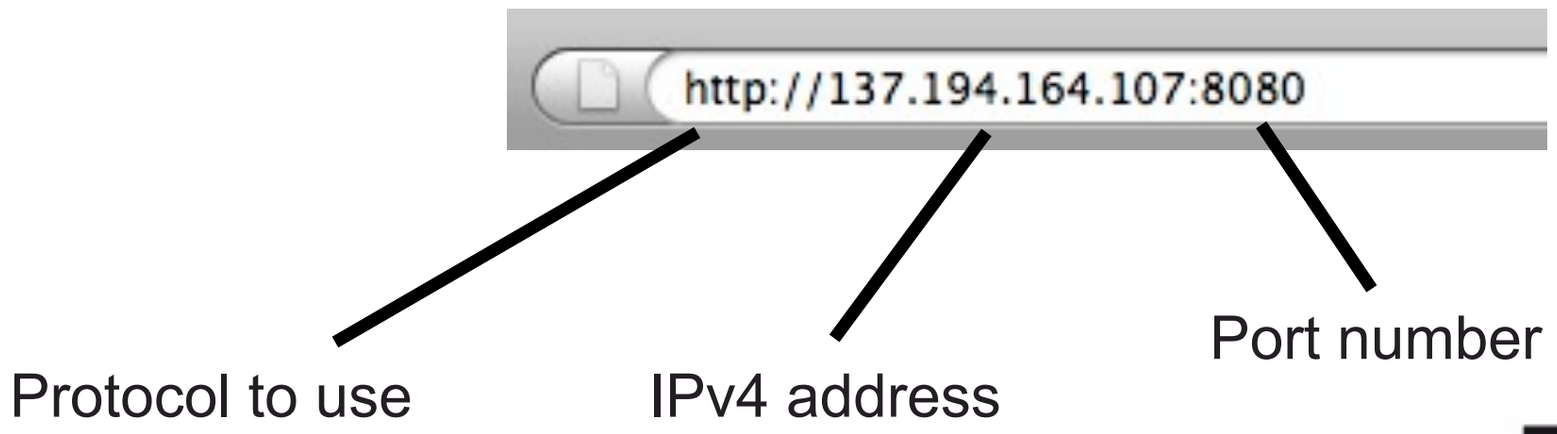
Well-known port numbers

- How does an application know which port to use?
- Some ports numbers are standardized by IANA (*well known port numbers*)
  - http://www.iana.org/assignments/port-numbers
  - cf. /etc/services file on an Unix Machine

```
ftp                 21/tcp    File Transfer [Control]
ftp                 21/udp    File Transfer [Control]
#                             Jon Postel <postel&isi.edu>
ftp                 21/sctp   FTP
#                             IETF TSVWG
#                             Randall Stewart <rrs&cisco.com>
#                             [RFC4960]
ssh                 22/tcp    SSH Remote Login Protocol
ssh                 22/udp    SSH Remote Login Protocol
```

Non-standard ports

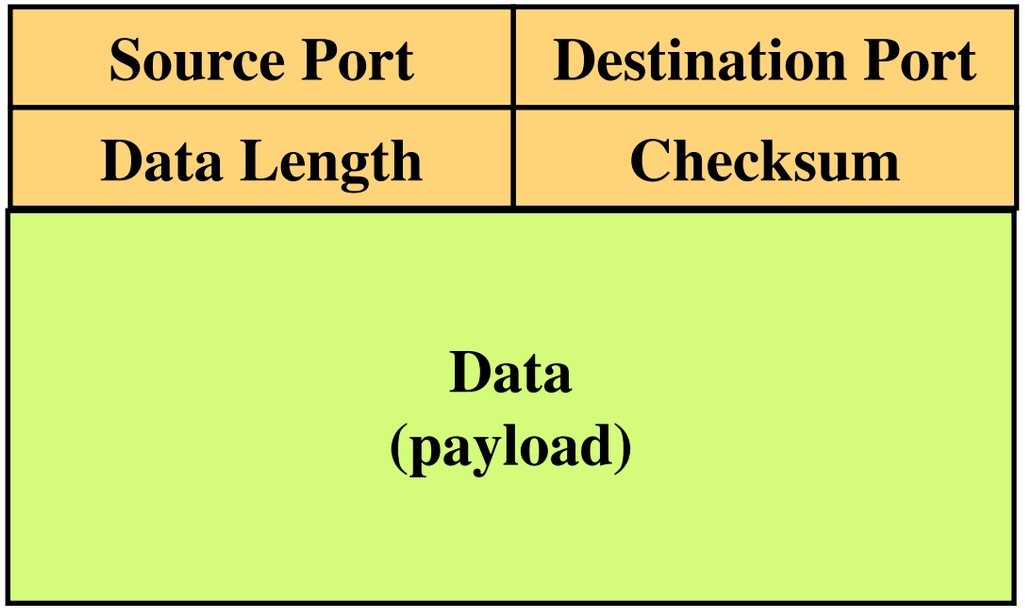
- Sometimes an alternate port can be used for a connection
  - Several applications offering the same service on the same machine (several web servers, several FTP transfers, etc.)
  - No dedicated port (new or specific application)
  - Obfuscation: hide the information of which service is on which port
- Applications need to agree on which port to use
  - Statically, in the code / a config file
  - Through explicit applicative messages (example : FTP passive mode)
  - User-specified to the application



UDP header

- An UDP datagram is an IP packet payload

- UDP has its own (minimalistic) header:
  - Source port (16 bits)
    - To address potential answers
  - Destination port (16 bits)
  - Data length (16 bits)
  - Checksum (16 bits ; RFC 768)
    - Computed over header + data
    - Necessary because no underlying layers reliability is assumed



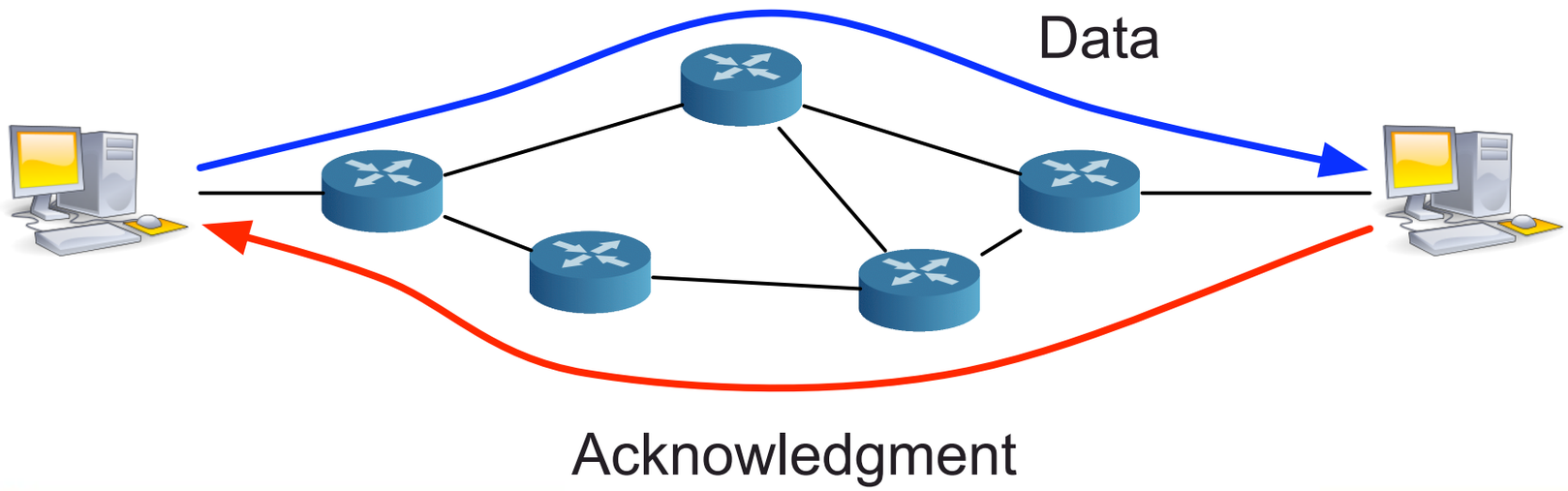
TCP: End-to-end reliability

TELECOM ParisTech

End-to-end acknowledgments

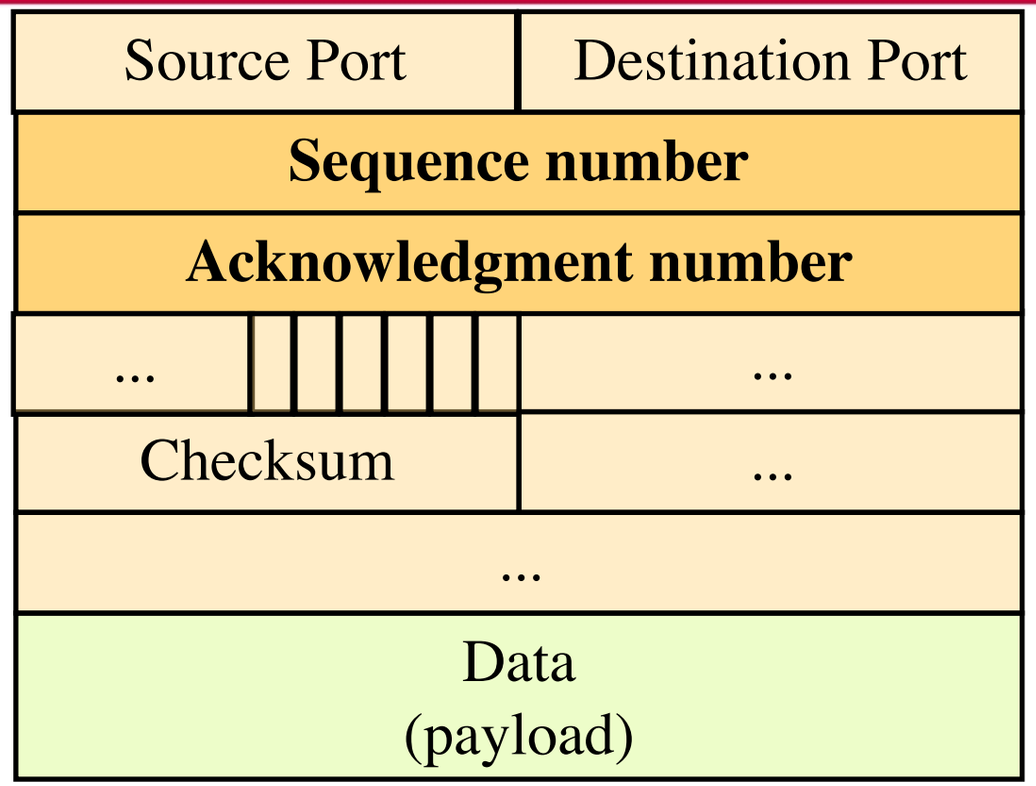
- End-to-end reliability uses the same mechanisms as link layer
  - Explicit acknowledgment of each segment
  - Incremental sequence numbers to detect missing segments
  - Usage of windows, similarly to the ARQ at link layer

- However:
  - Acknowledgments are, from IP's point of view, regular packets
  - They can follow alternate paths, be discarded, lors, late, ...



Acknowledgments implementation

- We add to the transport header a sequence number
  - 32 bits number
  - Unit: number of bytes and not segment ID (necessary when fragmentation occurs)
  - Initial value: a priori not equal to 0 or 1 ; it allows to distinguish successive short transmissions

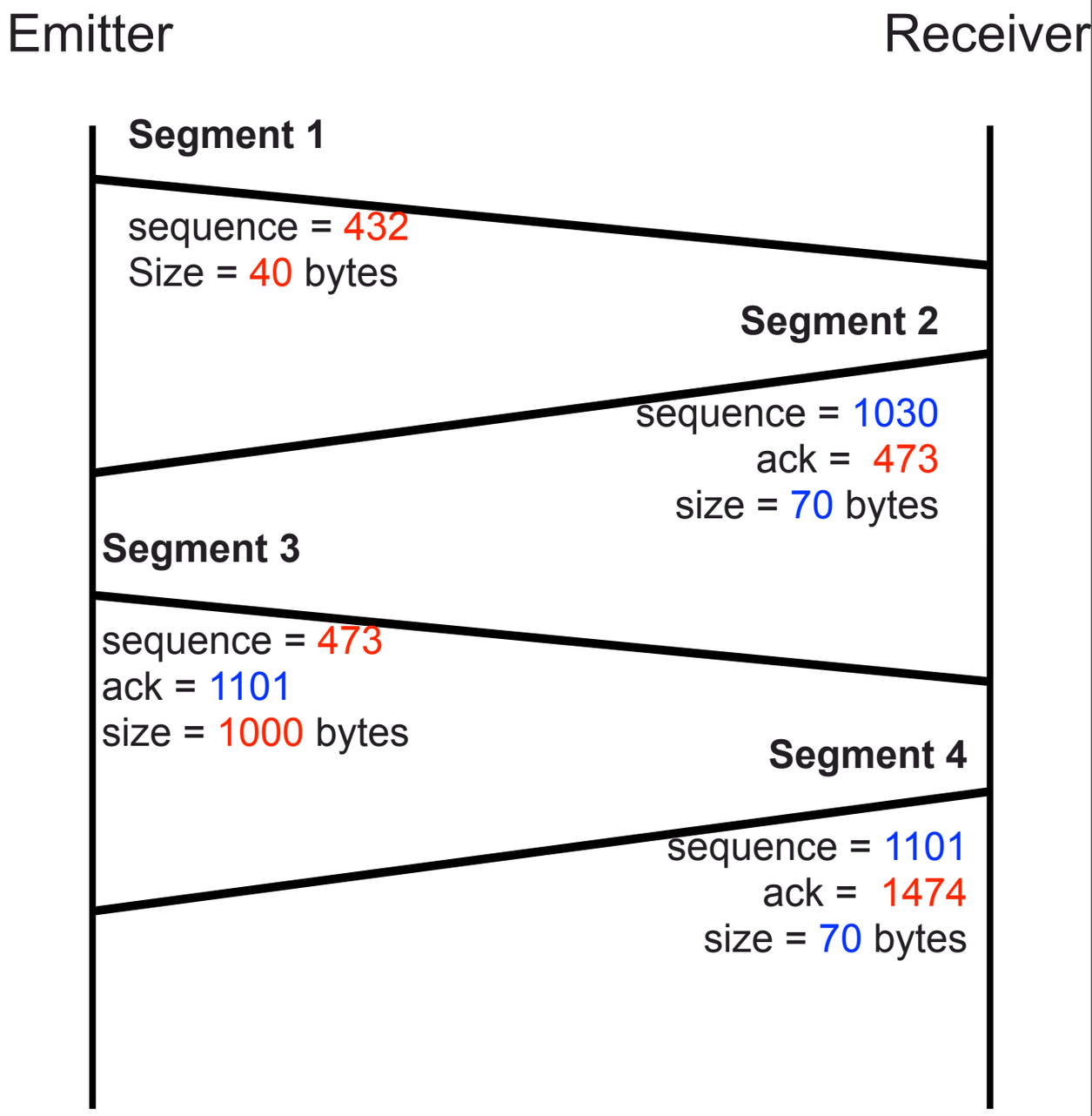


- Acknowledgments are regular segments; receiver mentions, in a dedicated field, last byte index successfully received
  - A bit (flag) indicates whether we are acknowledging something or not
  - It is indeed the next byte the receiver expects to receive
  - The format (field in a header rather than dedicated packet) allows to piggyback acks with regular data if bidirectional traffic.
  - If bidirectional traffic, sequence numbers are neither identical nor related

Acknowledgments: example

- For a bidirectional communication
  - Emitter starts with a sequence number equal to 432; receiver with 1030
  - If the counter passes over 2^{32}, loop back to 0

- If the initial sequence number may be different from 0, how to make sure that the first segments were received?



When is a segment considered as lost?

- The date at which we expect to receive an acknowledgement cannot be computed (unknown number of links, ...)
- Timeout setting requires measuring the Round Trip Time (RTT)
  - Measure, for each segment, the delay between its emission and the reception of the corresponding ACK (M)
  - Use an EWMA window to average these values:
    - RTT = α • RTT + (1 - α) • M
    - Typical coefficient value : α = 0.875
- The timeout then needs to be set according to this parameter
  - Van Jacobson Algorithm:
  - Dispersion between the expected and actual value of the RTT is measured:
    - D = α • D + (1 - α) • | RTT - M |
  - Empirically, the timeout is set to the average + 4 times the dispersion
    - Timeout = RTT + 4 • D

TCP: Sessions

TCP session

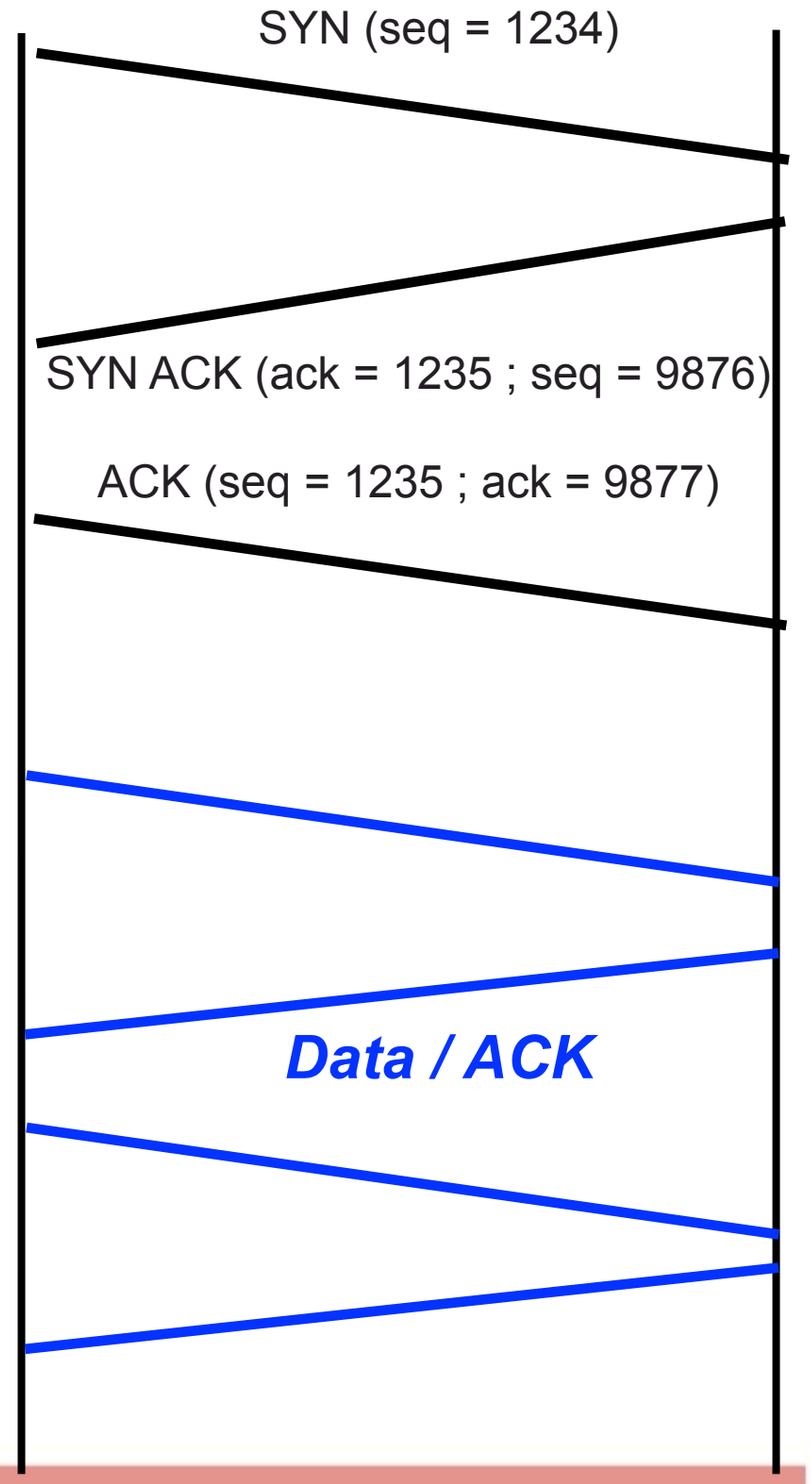
- A TCP exchange is more than a simple set of messages
  - Has a beginning and an end
  - Messages orders is preserved

- TCP specifies a dedicated message exchange to mark the beginning and end of a communication:
  - Start up the connection: message exchange to agree on both directions sequence numbers

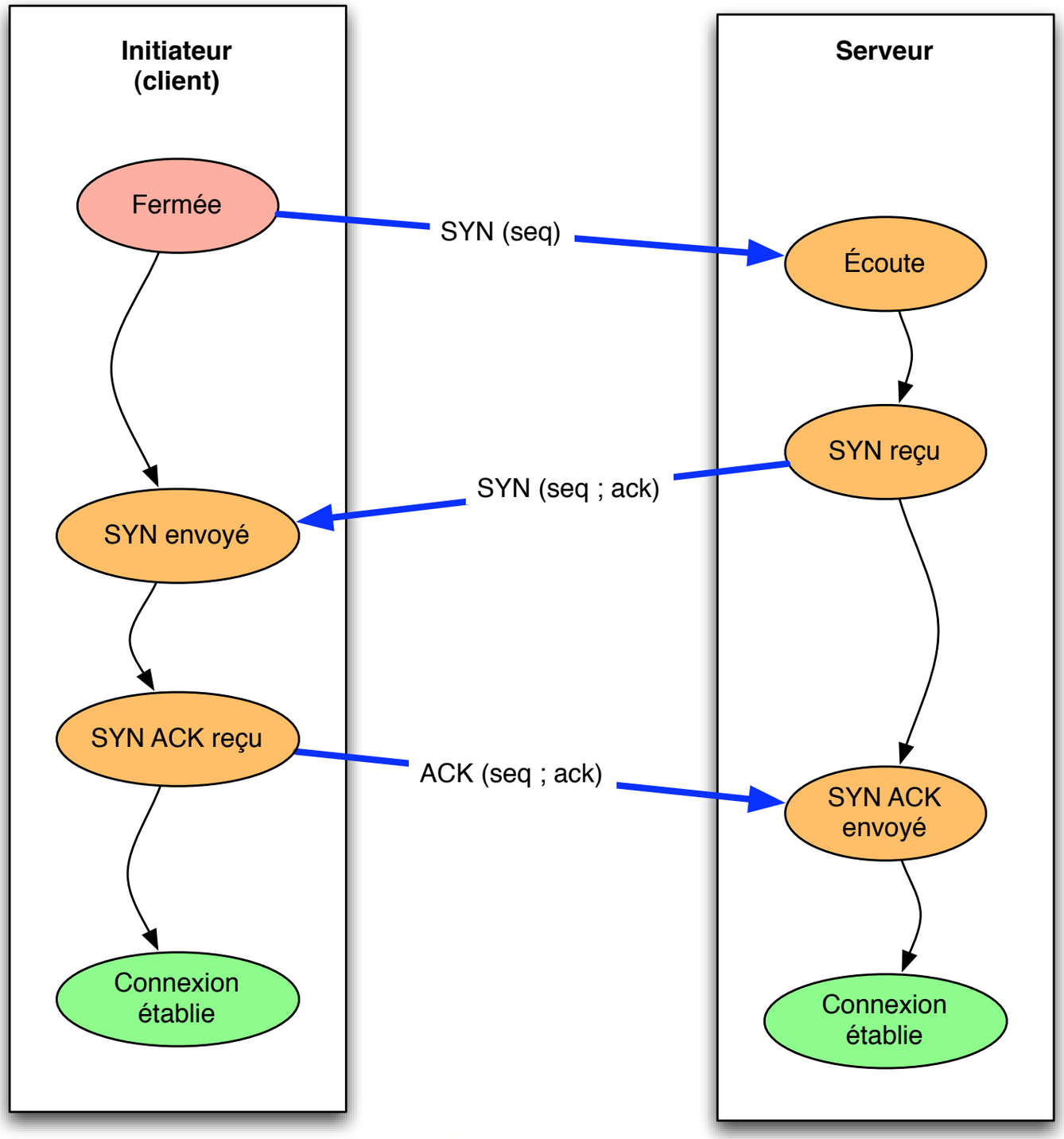
  - End of the connection: transmission of unacknowledged segments before effectively closing the connection

Connection opening

- It takes 3 messages to open a connection:
  - SYN : send the emitter's initial sequence number
  - SYN ACK : accept the connection and send the receiver's initial sequence number
  - ACK : emitter confirms it received correctly the receiver's sequence number.



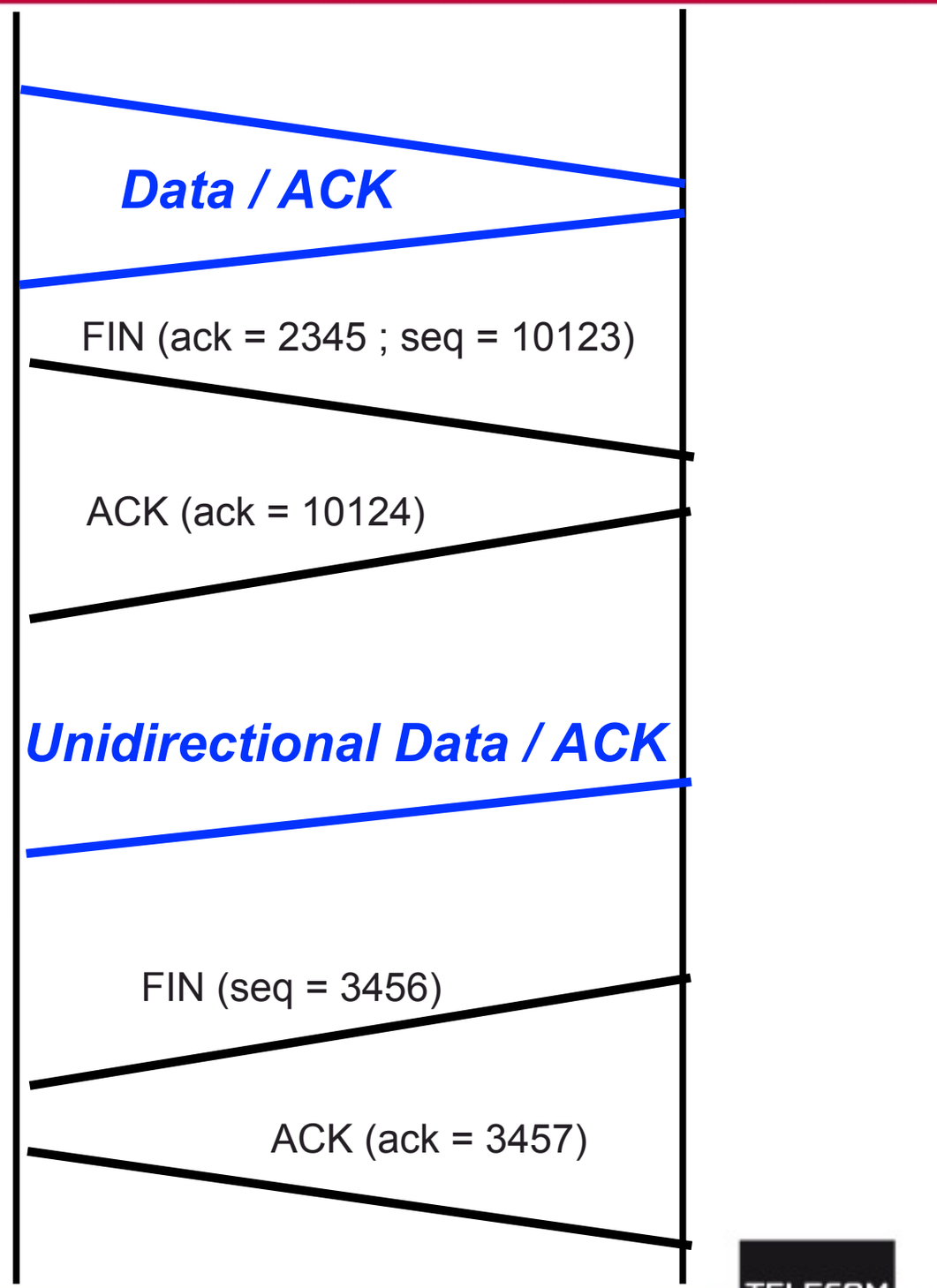
State diagram — connection opening



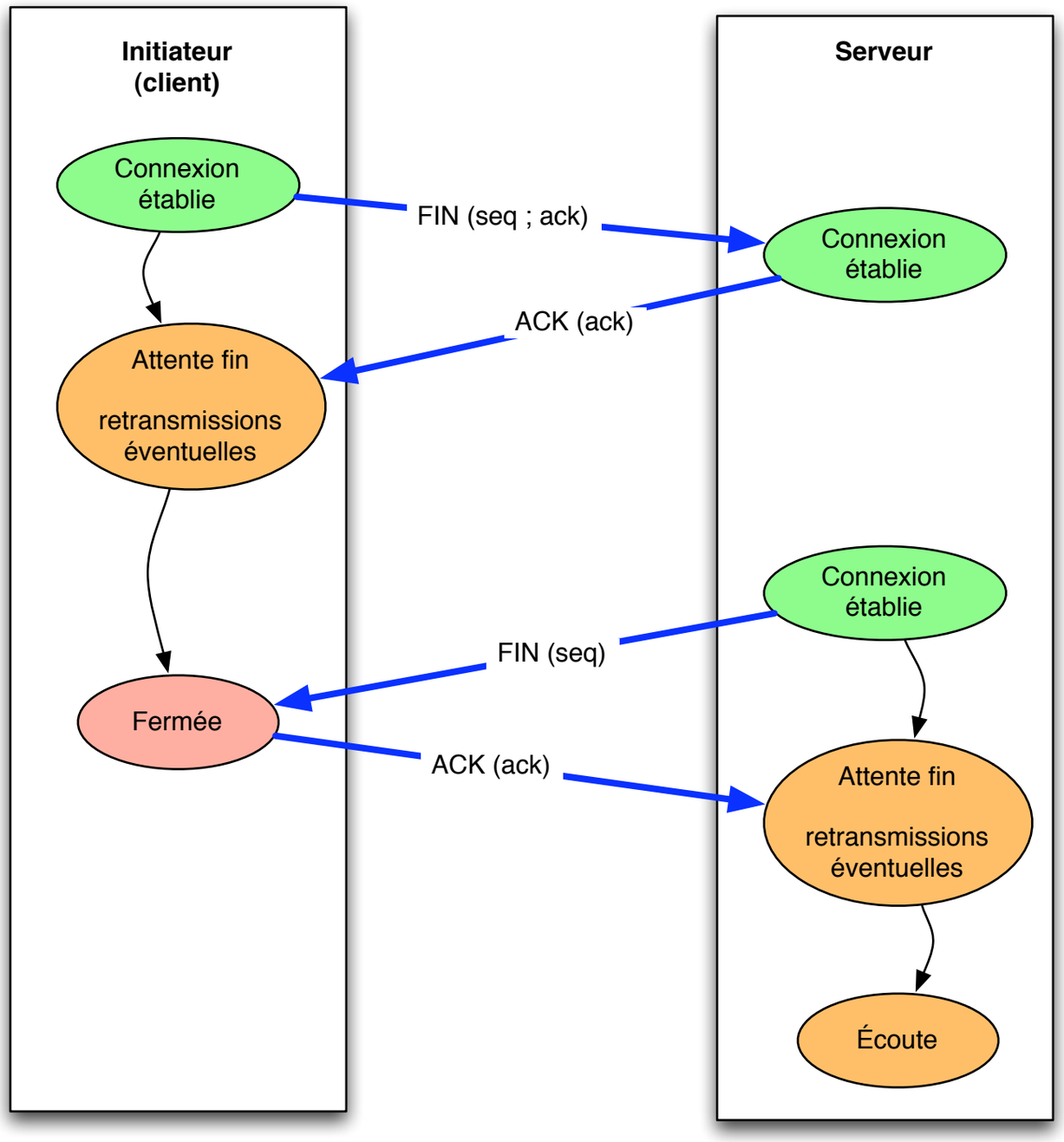
Closing a connection

- End of a connection is performed with two messages
  - FIN : indicate that we do not have any more data to transmit
  - ACK : notifies that all the expected data was received correctly

- This exchange needs to be performed in both directions
  - A connection can be kept open in a single direction



State diagram — connection closing

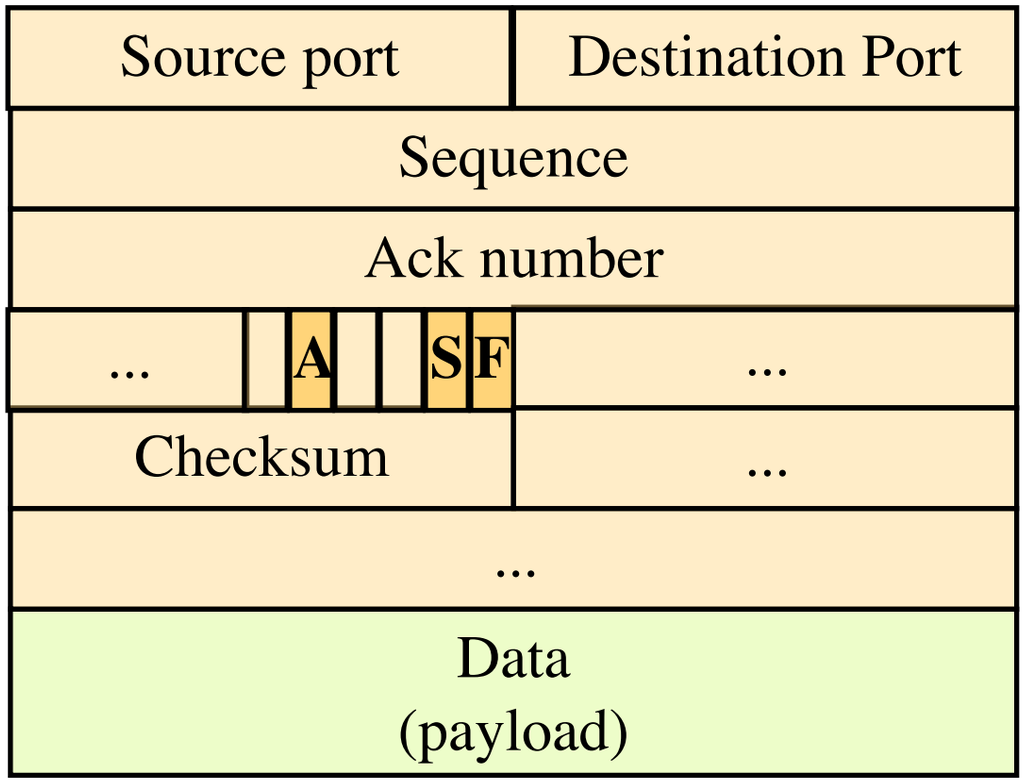


Implementation

- The control messages presented above are indeed not real dedicated messages

- Use of flags (bits) in the TCP header
  - Identical to the acknowledgments method

- Flags:
  - ACK
  - SYN
  - FIN
  - ...



TCP: Congestion control

TELECOM ParisTech

What is congestion control?

- It consists in adapting a flow throughput to the network capacity:
  - Limitation at the sender's side
  - When the receiver cannot handle the flow
  - When the network cannot handle the flow

- Why?
  - Emitting useless segments (that will be lots) imposes an unnecessary load on the network ; provokes de-ordering
  - Losses are (most of the time) due to congestion (overload of a network piece of equipment).

- How?
  - An emitter limits its sending rate according to **its perception** of the network/ receiver status

Windows

- Congestion control relies on two windows, whose size is expressed in bytes:
  - An anticipation window (cf. ARQ) that represents the amount of allowed unacknowledged data
    - Depends on the network status and in the end-to-end (bandwidth x delay) product
  - A reception window, maintained by the receiver
    - Indicates the free space in the reception buffer.
    - Transmitted in packets/acks headers

- The emitter considers both values (min) to find the correct emission throughput
  - Increase throughput when resources are available
  - Fast reaction to congestion to limit the problem quickly
  - Fairness between flows objective: everybody needs to participate to congestion resolution

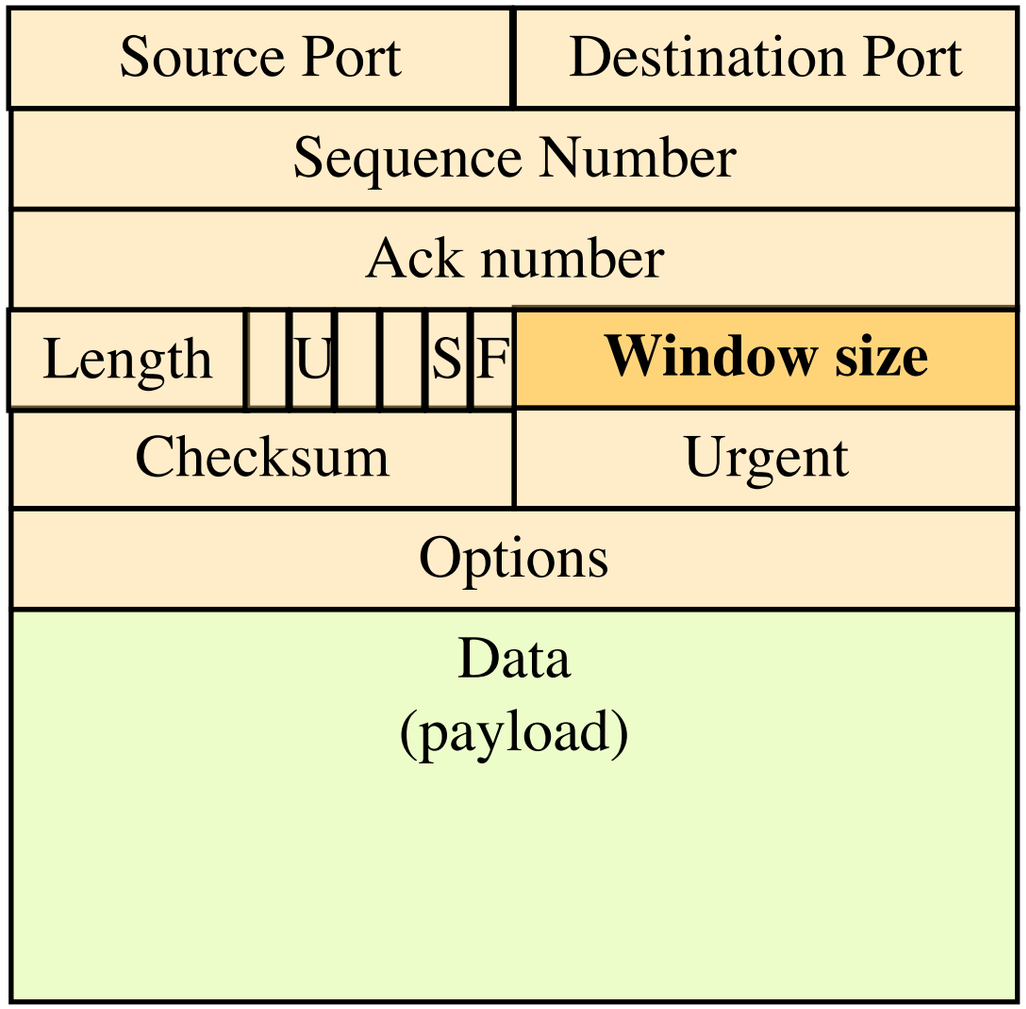
Building the emitter's perception

- The emitter can only rely on acknowledgments to know the network's status
- Lack of reception of an acknowledgment (before timeout) is interpreted as if the segment has been lost
  - Alternate possibilities: excessive delay, ack lost, ...
- Receiver window allows to distinguish between network losses and receiver losses
  - Number of bytes that the receiver can accept after this packet.

Full TCP header

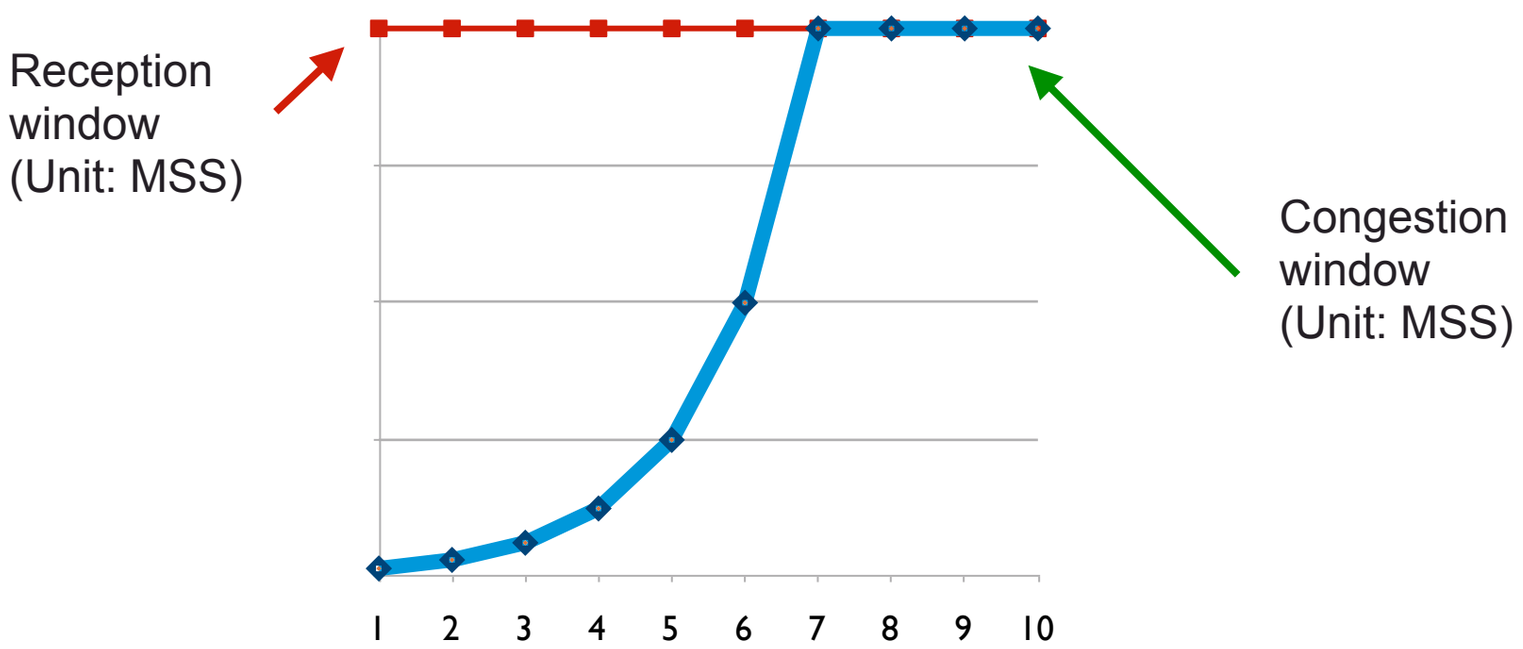
- The Window field reports the receiver's window status

- A few other fields
  - Options
  - Header length
  - Urgent data flag
  - Explicit Congestion Notification flag



Slow Start

- Cautious start
  - The emitter sends, at the beginning one MSS (max segment size) and expects to receive the ack.
- Exponential window size increase
  - Every time an ACK is received, window = window + 1
  - Doubles every RTT
  - Automatically stops when an ack is lost or when the receiver window is reached



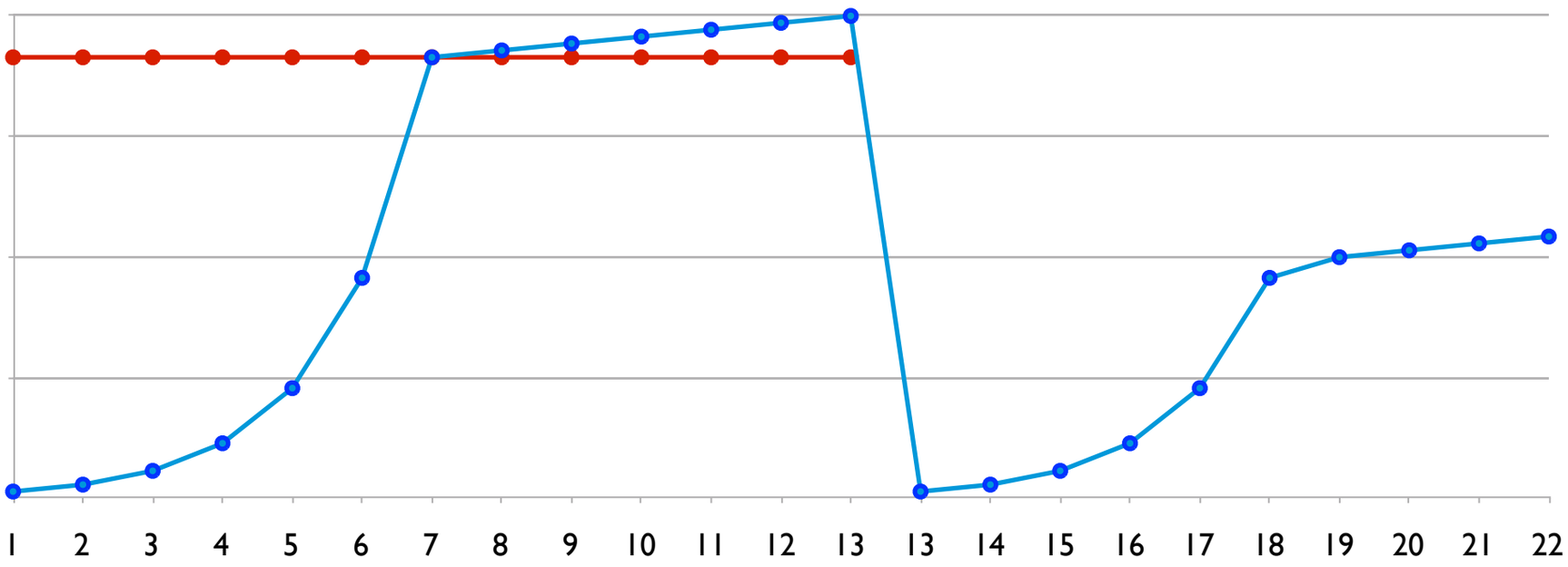
Note: The MSS is negotiated when the connection starts (SYN / SYN ACK) by both ends.

Typical value is 1460 bytes (Ethernet MTU minus 40 bytes for TCP/IP headers)

If not specified, default value = 536 bytes

Congestion Avoidance

- There is a threshold beyond which emitter becomes more cautious
  - When the congestion window reaches this threshold, window size increases by one MSS at most every RTT
  - Initial threshold value: 64 kB
- When a segment is lost (ack not received)
  - Re-initialize window to 1 MSS
  - Congestion avoidance threshold is divided by 2.



Fast retransmit & fast recovery

- Fast Retransmit
  - Timeout to conclude that a segment is lost can, from time to time, be shortened.
  - A receiver always acks only the last received byte. If there is a missing packet, there will be duplicate acks.
  - When the emitter receivers 3 duplicate acks (4 acks for the same byte), it concludes it has to retransmit the next segment.

- Fast recovery
  - After a fast retransmit phase, the collision avoidance mode is used (linear increase), without going through the slow start phase. We do not need to reset to 0, the reception of the subsequent segments indicate a moderate congestion.
  - When a timer expires, we reset the window and start slow start again.
  - Allows a more important throughput when congestion is moderate

TCP Flavors (versions)

- Numerous TCP versions have been proposed (IETF drafts / RFC)

Version	**Year**	***Slow Start***	***Congestion Avoidance***	***Fast Retr /***	***Other***
Tahoe	1988	yes	yes	partial	
Reno	1990	yes	yes	yes	
New Reno		yes	yes	yes	Enhanced *Fast recovery*
Vegas	1994	yes	yes	yes	Use an history to reduce window size *before* losses
SACK	1996	yes	yes	yes	Selective repeat — Implemented as an option

Conclusion

- Two main transport protocols exist
  - UDP : minimalistic - port numbers & error check
  - TCP : reliability (ack + retransmissions), connection (open & close the session), congestion control

- When a congestion happens, TCP adapts its throughput, UDP suffers losses
  - Unfair coexistence of both protocols

- Today, most flows are TCP flows in the Internet (even though P2P tends to use UDP and implement retransmission at application layer)

- Other enhancements (ECN, ...) and other protocols (DCCP, SCTP, ...) exist.

Programming Interface : *sockets*

TELECOM ParisTech

Programming TCP/IP applications

- For a programmer, a distributed application is on the top of the transport layer
  - Session (SSL/TLS) and presentation (XML, JSON, ...) are invoked through additional libraries
- The classical API is called “sockets”
  - Close to Inter-process communications (usually in operating systems courses)
- Classical sockets invoke UDP or TCP over IP
  - There are also other types of sockets, depending on the transport (e.g. DCCP) or network (e.g. IPv6) layers.

Socket API: usage

- **Client Side**
- **Server Side**
  - Reserve **port X** with the operating system
    `Sockets : bind()`
  - Start monitoring the port for incoming connections
    `Sockets : listen()`
- Send a connection request on **port X**
  `Sockets : connect()` →
  - OS examines the port number and transmits the datagram to the registered application
  - Application answers — accept the connection
    `Sockets : accept()` ←
- Application-level exchanges
  `Sockets : send() ; recv()` ↔ `Sockets : send() ; recv()`
- Close the connection
  `Sockets : close() ; shutdown()` ↔ `Sockets : close() ; shutdown()`