POLYTECH
PARIS-SUD

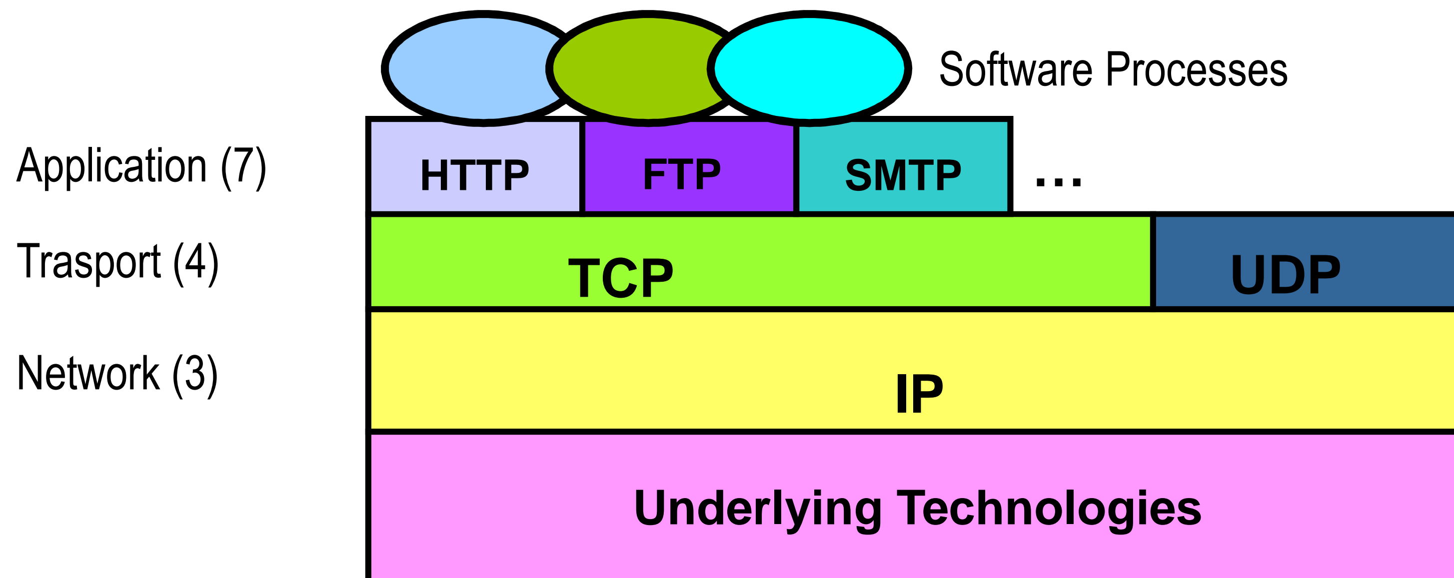
Transport Layer

-UDP (*User Datagram Protocol*)

-TCP (*Transport Control Protocol*)

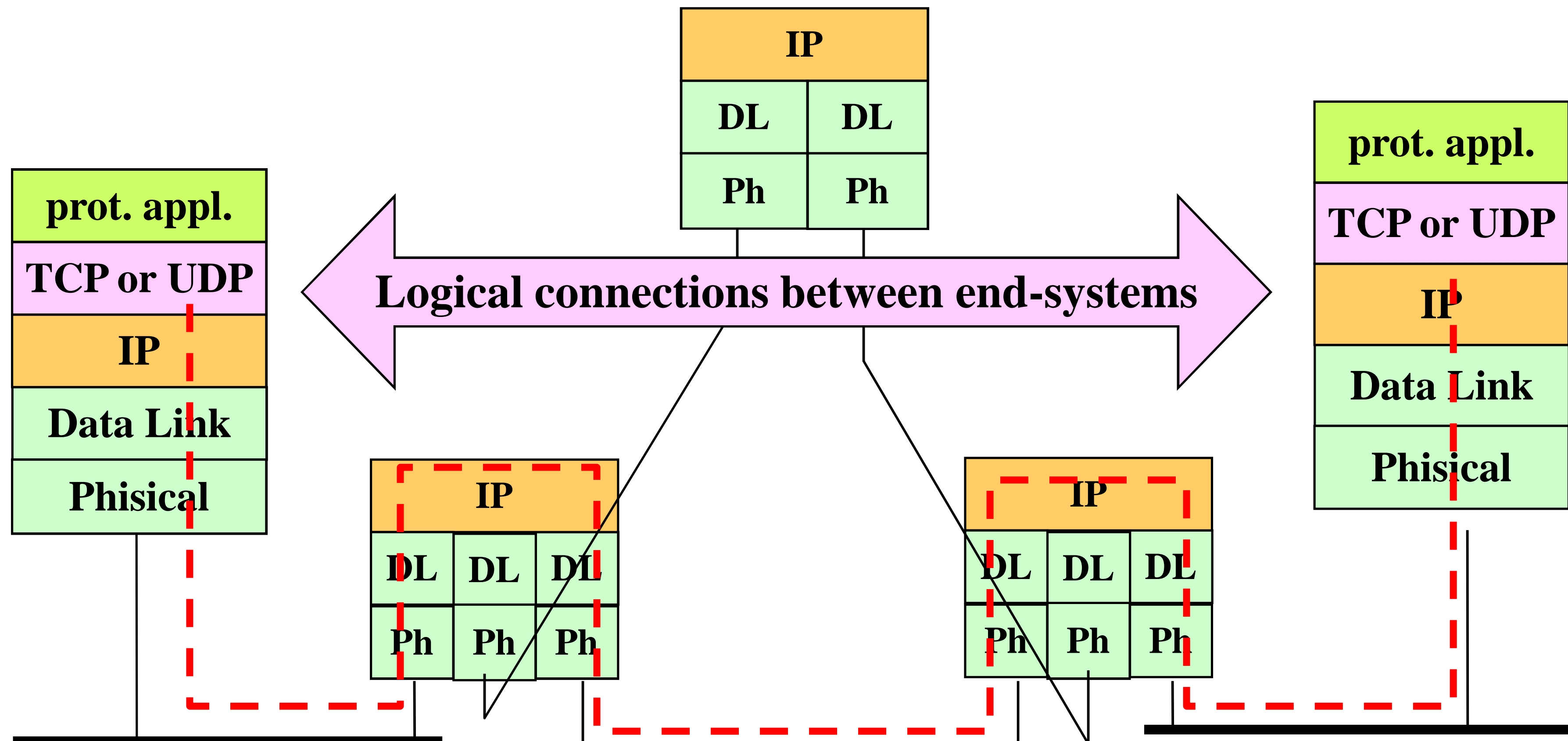
Transport Services

- The transport layer has the duty to set up *logical* connections between two applications running on remote hosts.
- The transport layer makes transparent the physical transfer of messages to the applications (software processes)



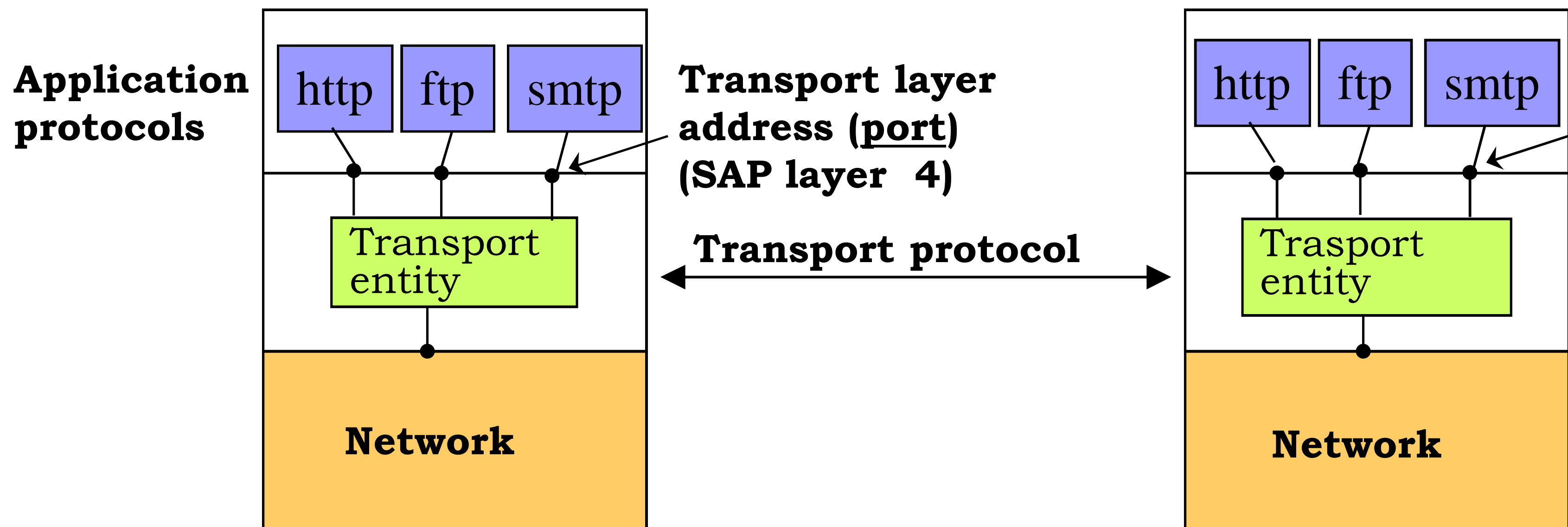
Transport Services

- The transport layer is implemented in the *end-systems (hosts) only*
- Logical connections through applications



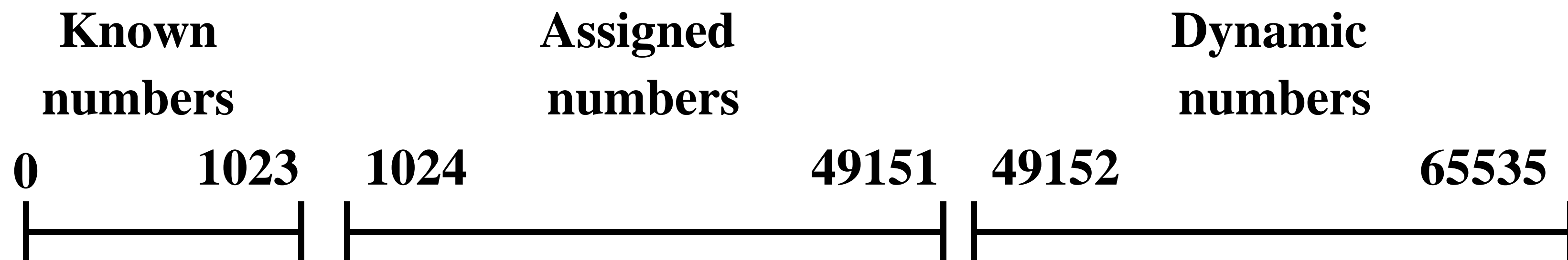
Transport Services

- Multiple applications can be contemporary active on the *end systems*
  - The transport layer acts as a *multiplexing/demultiplexing entity*
  - Each logical link between two applications is addressed at the transport layer



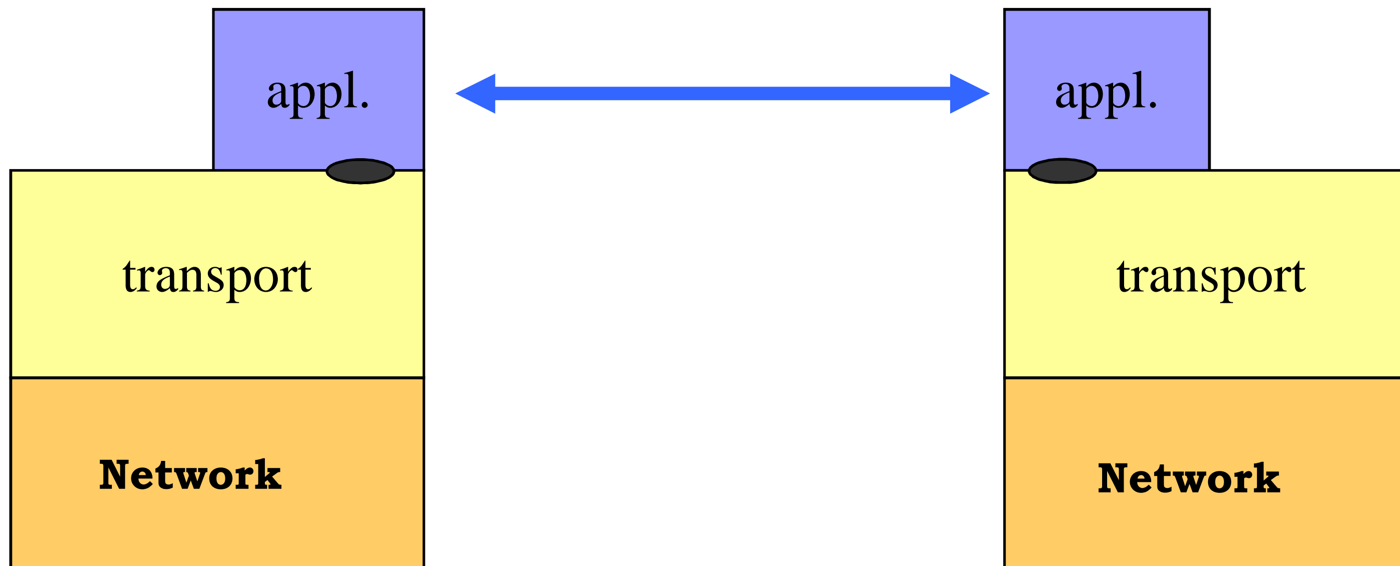
Addressing through *ports*

- *Multiplexing/demultiplexing* functions are handled through addresses contained in layer 4 PDUs (also called “segments”)
- Such addresses are 16-bits long (ports)
- Port addresses can range from 0 to 65535
- Fixed port numbers are assigned to *well known* application *servers* (HTTP, FTP, SMTP, DNS, etc.)
- Dynamic numbers are assigned to application *clients*



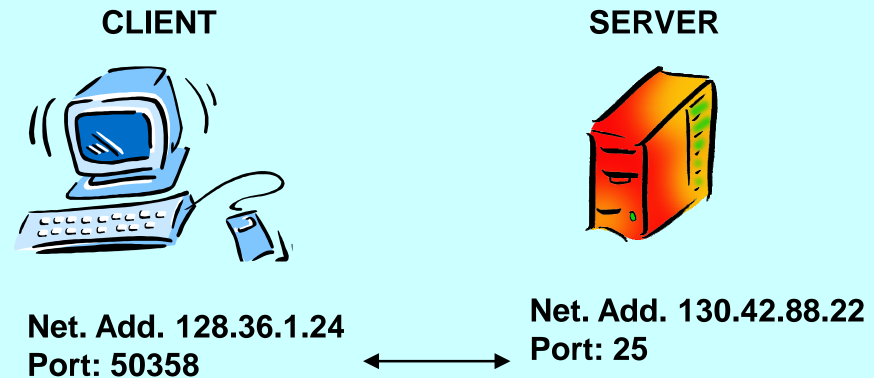
Socket

- The *port number* and the *IP address* identifies an application process running on a host
- Such *couple* is called a “*socket*”

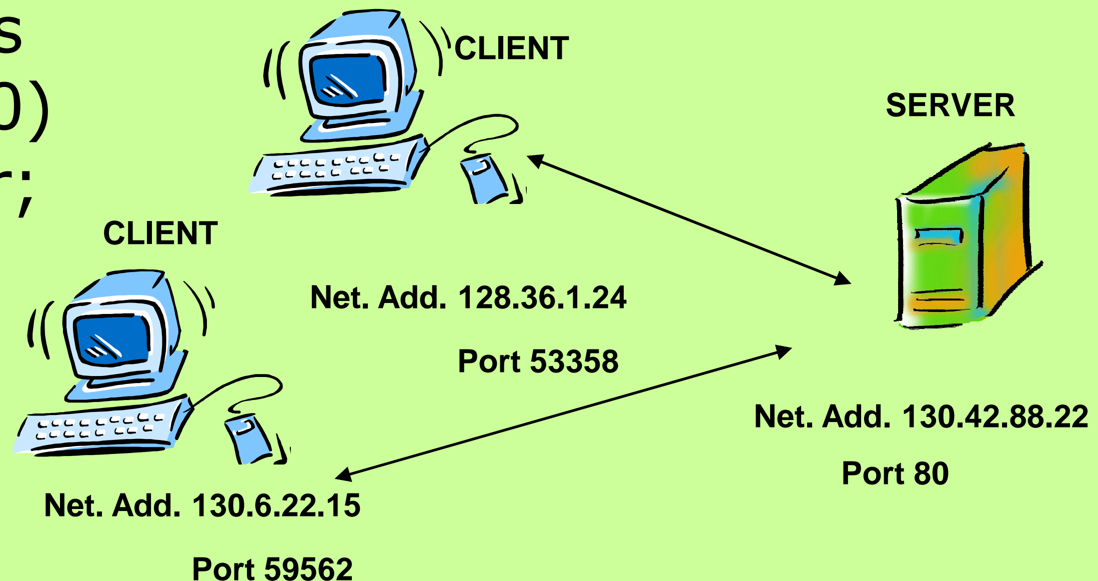


Sockets and connections

- One *client connects to the port of a SMTP server,* port 25 (email service)



- Two clients access the same port (80) of an HTTP server; different *socket* couples



Transport Services

- Different types of communication services
  - Reliable transfer (guaranteed, in sequence delivery)
  - Unreliable transfer (only addressing and multiplexing)
  - Connection-oriented transfer
  - Connectionless transfer
- In the TCP/IP suite, two protocols are defined
  - TCP (Transmission Control Protocol) connection oriented and reliable
  - UDP (User Datagram Protocol) connectionless and unreliable

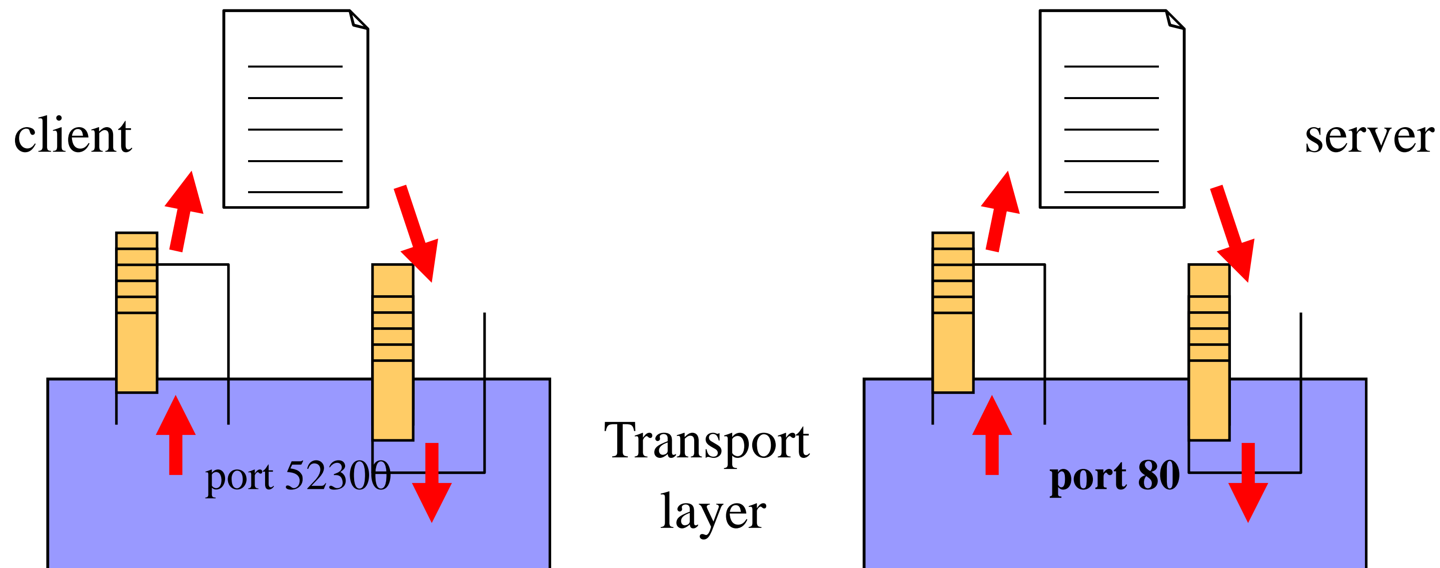
Applications and Transport

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	Typically UDP
Internet telephony	typically proprietary	Typically UDP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Source: Computer Networking, J. Kurose

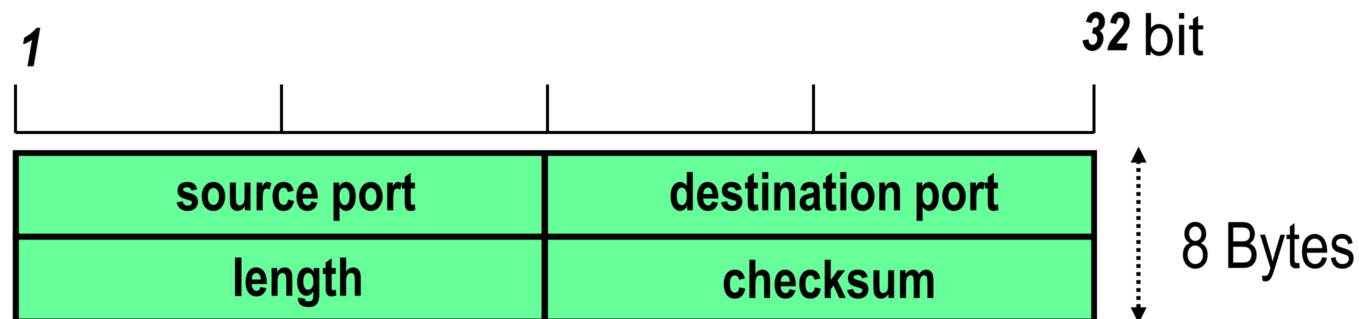
Buffering Service

- Transport protocols are implemented in the most common OSs
- Whenever a port is assigned to a process (*client* or *server*) two queues are defined and reserved by the OS
- Data *buffering* functionalities



User Datagram Protocol (UDP) RFC 768

- It does add very few things to IP:
  - Applications addressing (mux/demux)
  - Loose error checking (without correction)
- … thus:
  - It's a datagram protocol
  - It does not guarantee the delivery
  - It does not implement any control mechanism (neither on flows, nor on errors)

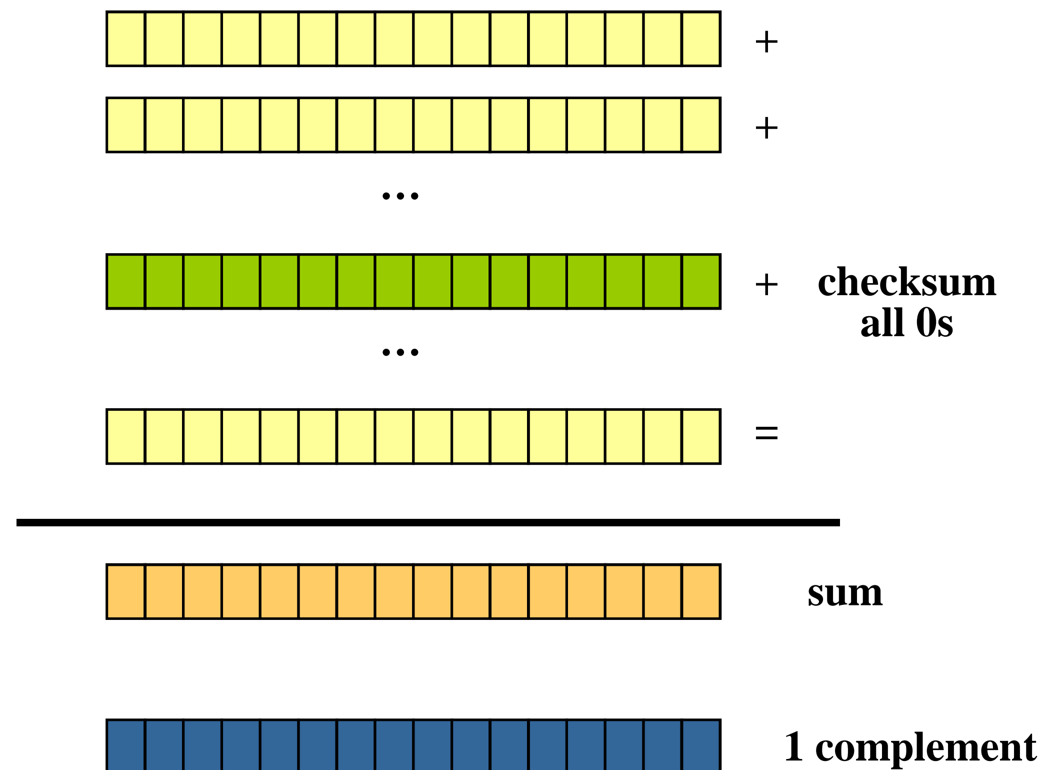


***Checksum:* integrity check**

- Redundant information in the UDP header for error control
- The *checksum* field is computed by the transmitter (16 bits) and inserted into the header
- The receiver repeats the same computation on the received segment (*checksum field included*)
- If the result is positive it processes the segment, otherwise it drops it

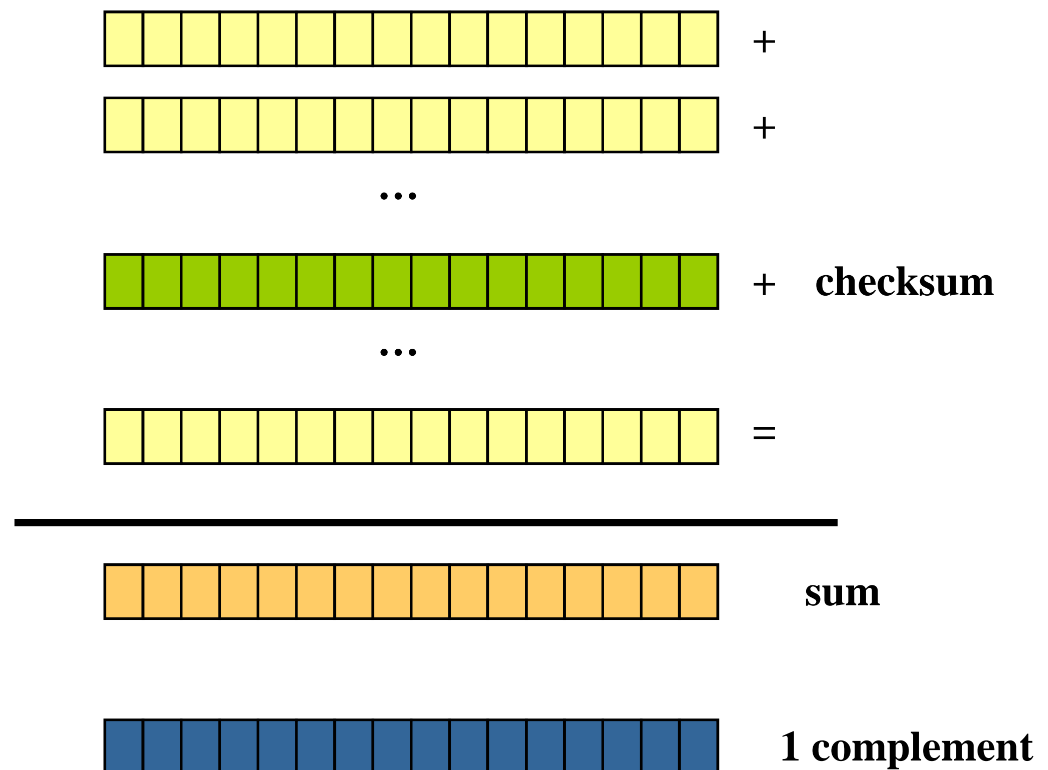
Checksum: transmitter's side

- The segment + pseudo-header are divided into 16-bits chunks
- The *Checksum field is set to 0*
- All the chunks are summed up
- The 1-complement of the result is inserted in the checksum field

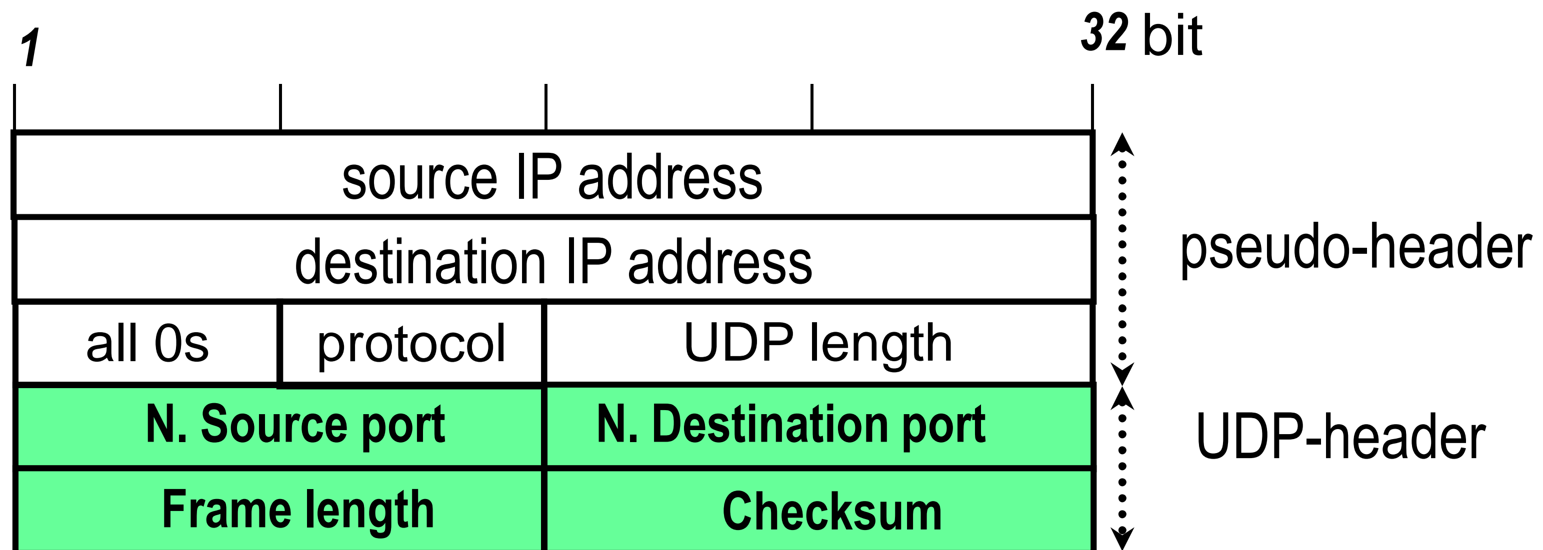


Checksum: receiver's side

- The segment + pseudo-header are divided into 16-bits chunks
- All the chunks are summed up
- The 1-complement of the result is taken
  - If all 0s the segment is processed
  - Otherwise is dropped



UDP Pseudo Header used for checksum calculation



Transmission Control Protocol (TCP) RFC 793 et al.

- TCP:
  - Ensures reliable transfer
  - Provides ordered delivery
  - Provides errorless delivery

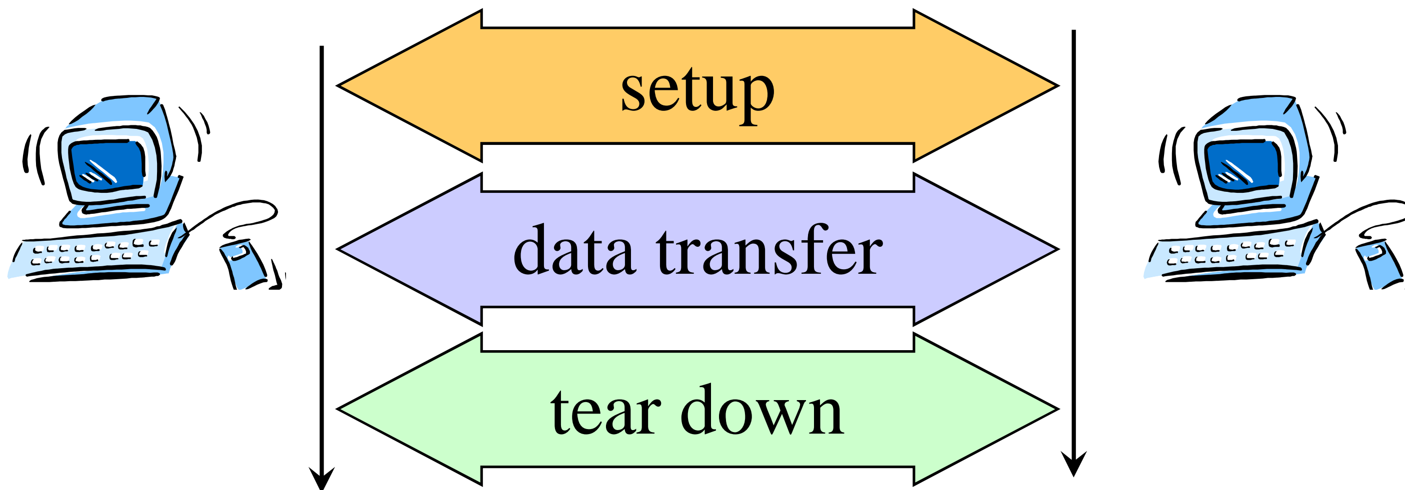
- TCP is used to build up applications relying on errorless transmissions (web, emailing, file transfers etc.)

- Original Philosophy of the Internet: *simple unreliable Network service (IP), reliable transport (TCP)*

- TCP implements an *end-to-end* congestion control to let the users share the common resources fairly

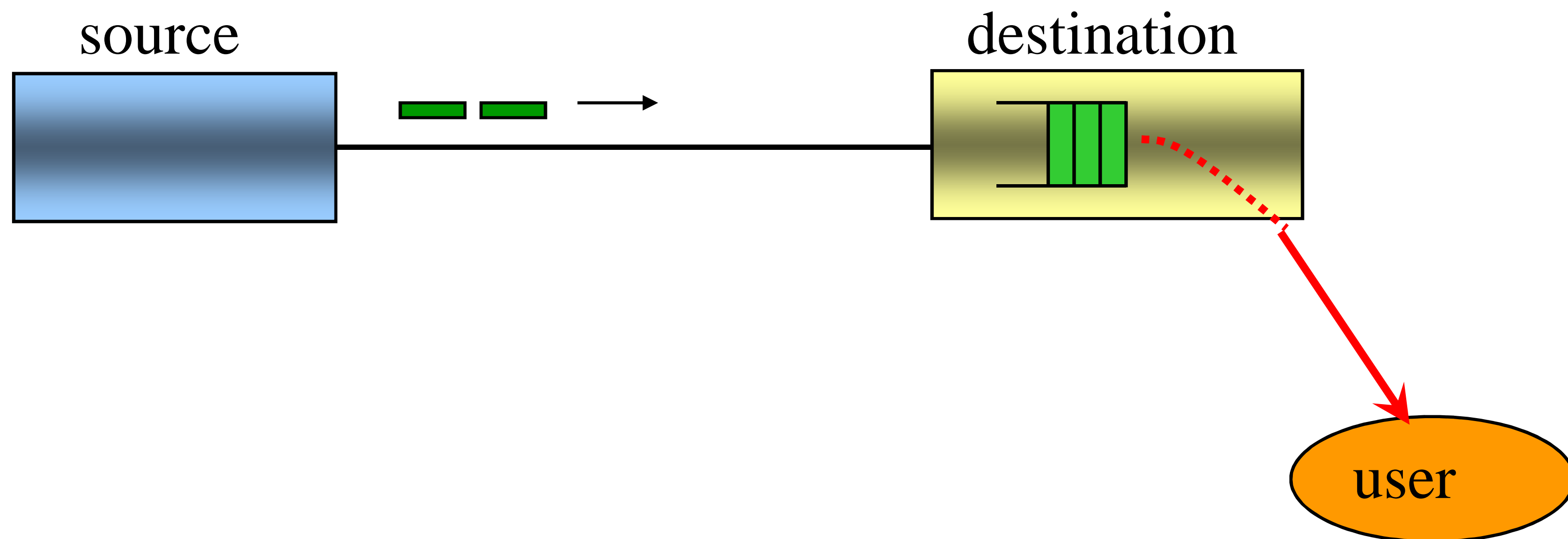
TCP: *connection oriented*

- TCP is connection oriented:
  - Before data transfer, a signaling phase to set up the communication is performed
  - TCP relies on *connectionless* services
  - TCP uses *full-duplex* connections

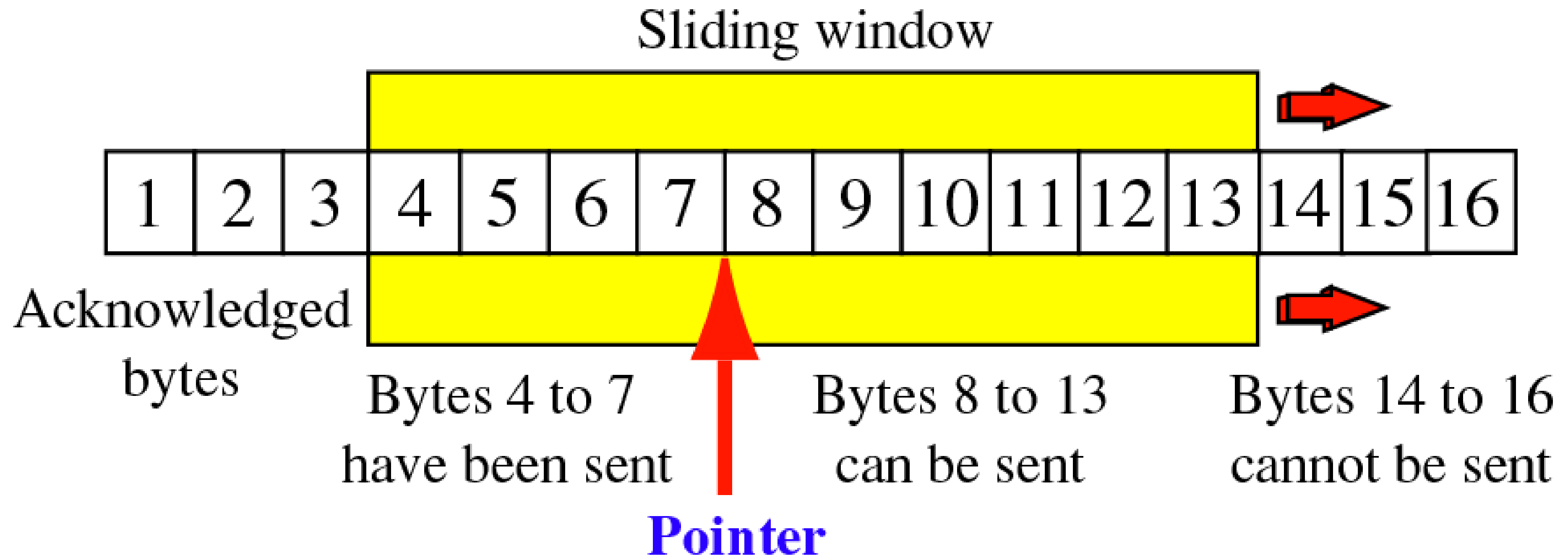


TCP: Flow control

- The incoming data flow is tailored on the capacity of the receiver
- The flow control is based on a *sliding window*

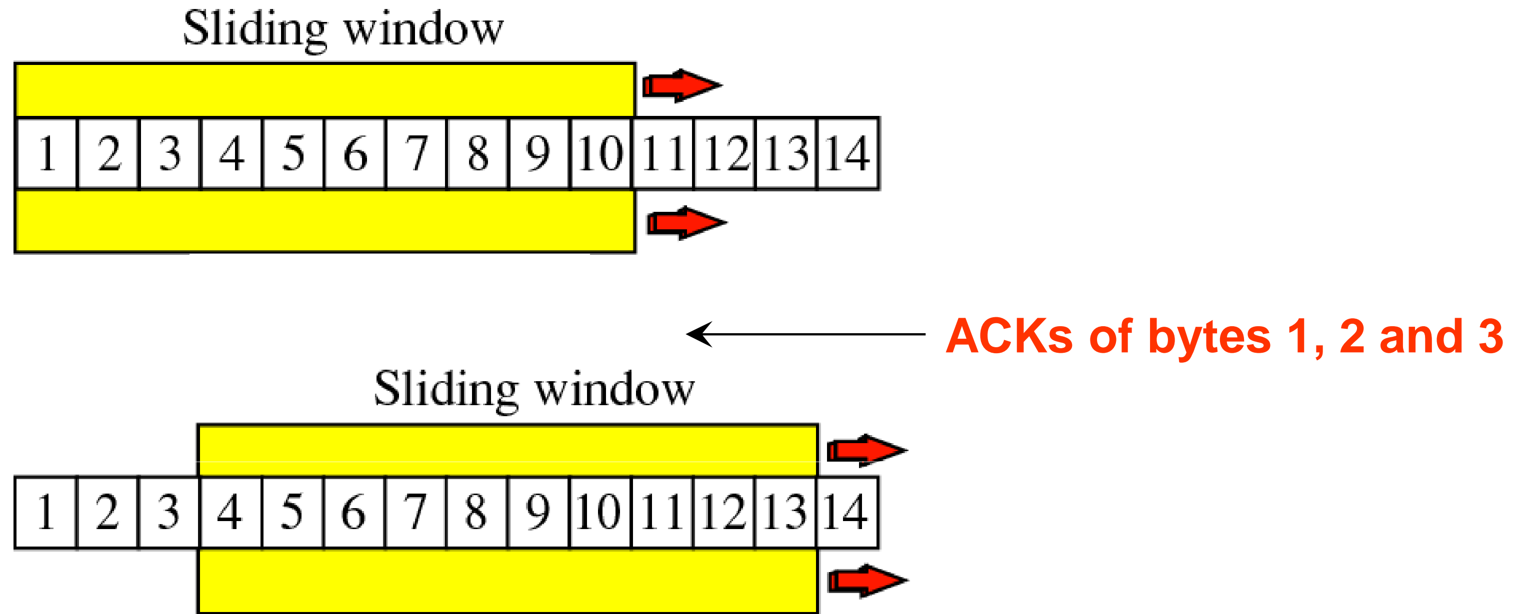


Sliding Window



- The sliding window defines the bytes which can be sent out *without waiting* for an ACK

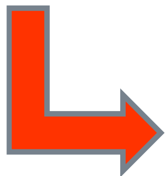
Sliding Window



- The window slides on the right of a number of positions equal to the number of acknowledged bytes (3, in this example), now including bytes 11, 12 and 13

TCP: congestion control

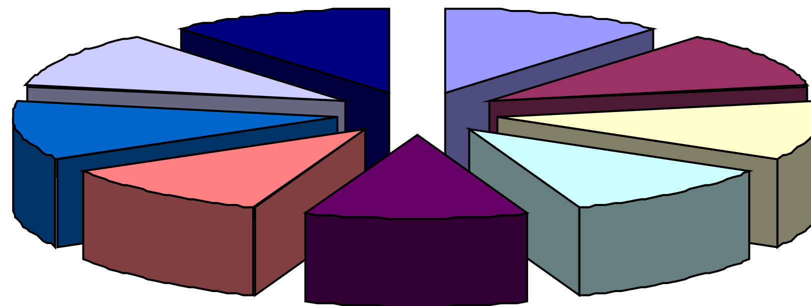
- The data flow is regulated also on the basis of the network conditions
- If the network is congested TCP reduces its transmission rate
- No mechanism to notify network congestion directly
- TCP infers network congestion on the basis of segments loss



- It is called a *black-box* approach for congestion estimation

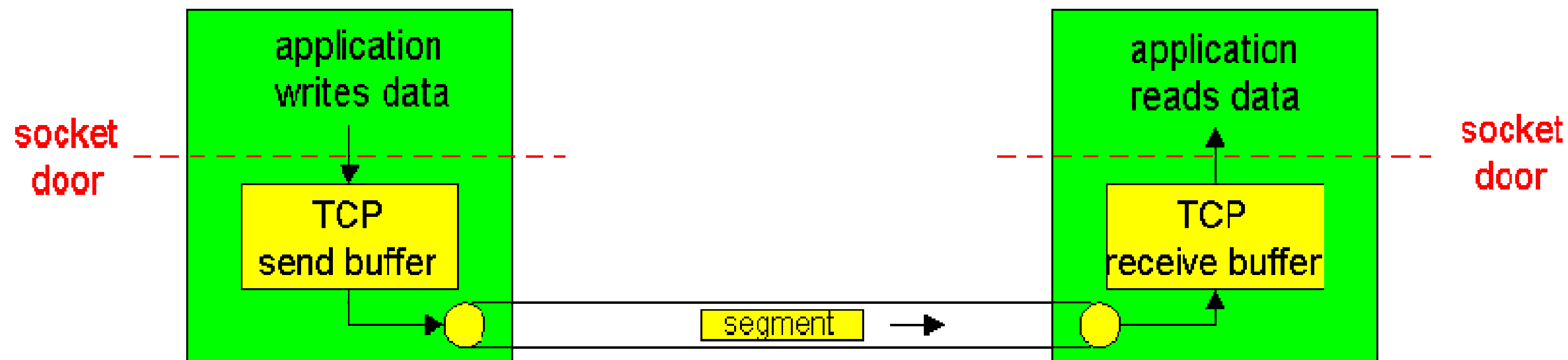
TCP: congestion control

- Still based on a *sliding window* whose dimension is regulated on the basis of the network conditions (packet loss)
- All the flows are reduced to achieve a *fair share* of the available resources



TCP: Data Flow

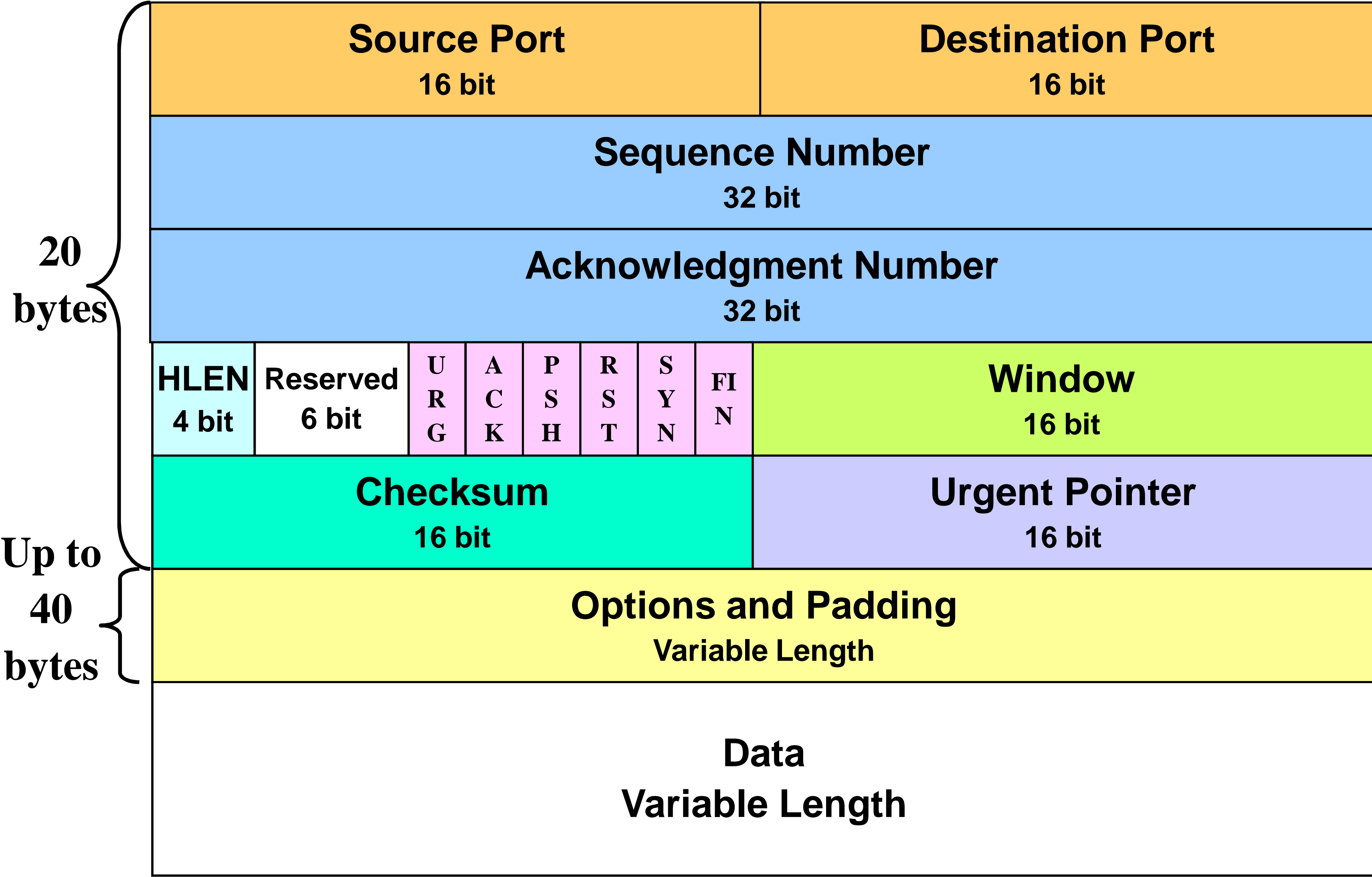
- TCP handles continuous data flows (byte stream)
- TCP converts data flows in *segments* to be passed downwards to IP
- Segment dimensions are variable
- TCP accumulates data received by the applications (buffering)
- Periodically, TCP builds up segments out of the buffered data
- The segment dimensions are critical for the TCP performance, dimension optimization techniques are needed



TCP: byte numbering and ACKs

- TCP implements a go-back-n like mechanism to cope with packet loss
- Byte numbering and ACKs
  - TCP numbers each transmitted byte with a sequence number
  - Each TCP segment reports the sequence number of the first byte carried by the segment itself
  - The receiver acknowledges the received bytes sending back the sequence number of the last received byte + 1 (It is called the *next expected byte*)
  - If the ACK does not arrive within a time limit (Retrasmission TimeOut, RTO), data is retransmitted

TCP Segment Format



TCP Segment Header(1)

- **Source port, Destination port:** 16 bits each
- **Sequence Number**: of the first byte carried by the segment
- **Acknowledgement Number:** sequence number of the *next expected byte* (valid only if the “ACK” flag field is valid, i.e. equal to 1)
- **HLEN (4 bytes words):** TCP header length, must be multiple of 32 bits (4 bytes)
- **Window:** dimension of the receiving window as communicated by the receiver
- **Checksum:** calculated on a virtual header adding IP source and destination addresses

TCP Segment Header (2)

- Flags:
  - **URG**: urgent data identifier; *urgent pointer* points to the first byte of the urgent data
  - **ACK**: set if the segment carriers a valid ACK; *acknowledgement number* contains a valid number
  - **PSH:** set if the transmitter uses the PUSH command; the receiver can ignore the command (depending on the implementations)
  - **RST:** used for connection reset, without explicit *tear down*
  - **SYN:** *synchronize*; used during the connection *setup* phase
  - **FIN:** used for the explicit tear down of the connection
- **Options and Padding:** optional fields like the MSS value (default value is 536 bytes)

Options

- 1 byte options:
  - no operation: 00000001 (used for padding)
  - end of option: 00000000 (final byte padding)
- Long options:
  - Maximum Segment Size (MSS)
  - Window Scaling factor
  - Timestamp

Options: Maximum Segment Size (MSS)

- Maximum segment dimension to be used during the connection
- Dimension is decided by the sender (and confirmed by the receiver) during the connection *setup phase*
- *Default* value is 536 byte, maximum value is 65535 byte. Typical value around 1460 bytes.

Code (00000010)	Length (00000100)	MSS 16 bit

Options: Scaling Factor

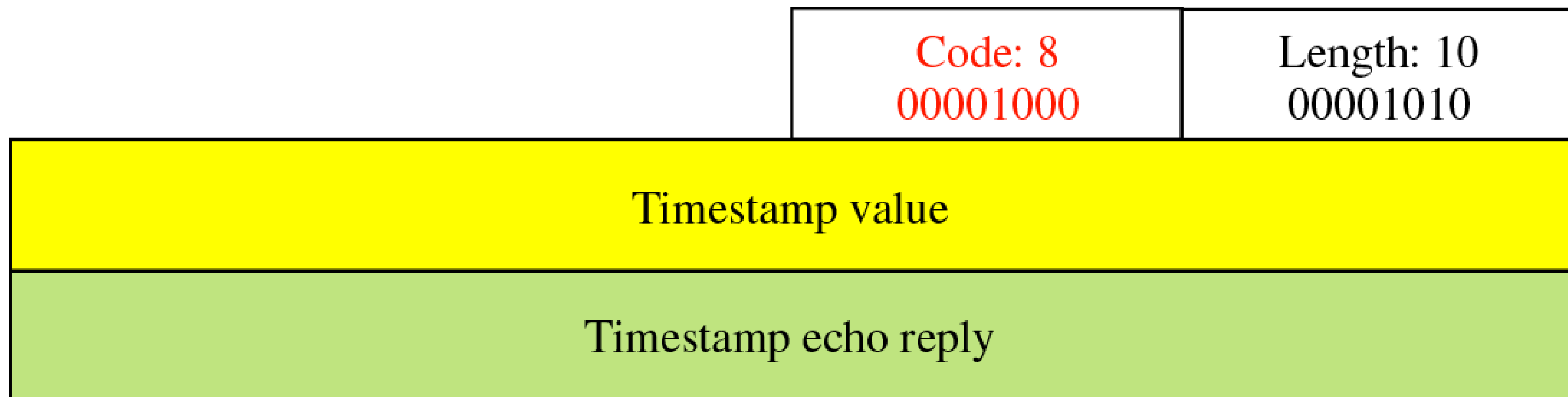
- Used for TCP connections running on high speed (optical) networks
- Scaling factor of the *window field* in the header
- *Default* is 1 byte
- The actual window value is

$$window * 2^{Scaling_Factor}$$

Code (00000010)	Length (00000011)	Scaling Factor 8 bit

Options: *Timestamp*

- Used to estimate the *Round Trip Time* (RTT)
- The TCP source prints the transmission time in *Timestamp value*
- Destination prints in *Timestamp echo reply* the received *Timestamp* value when it sends an ACK for the current segment

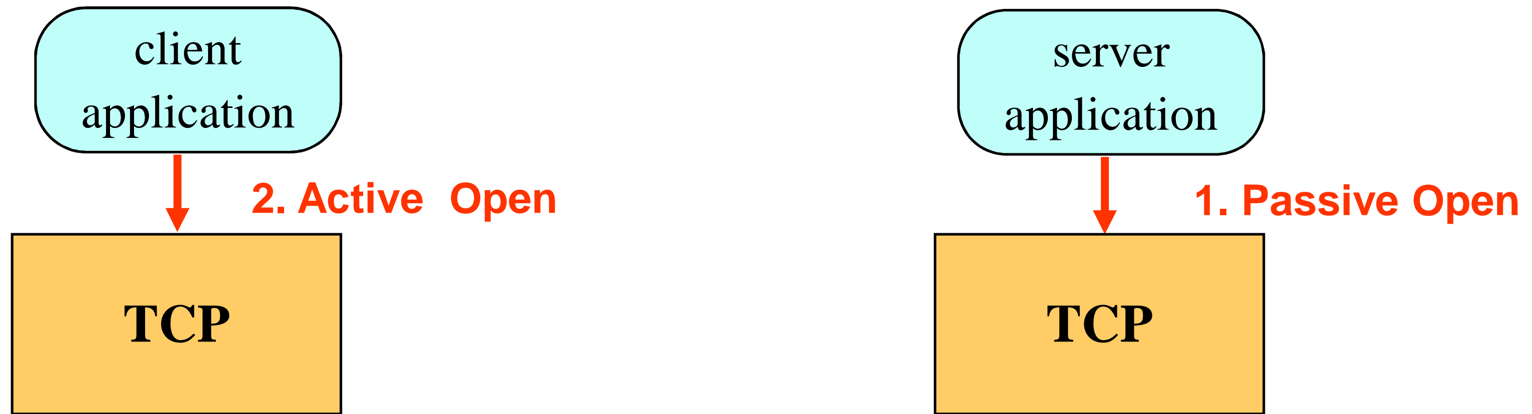


Services and Ports

- Some of the most common port-applications bindings

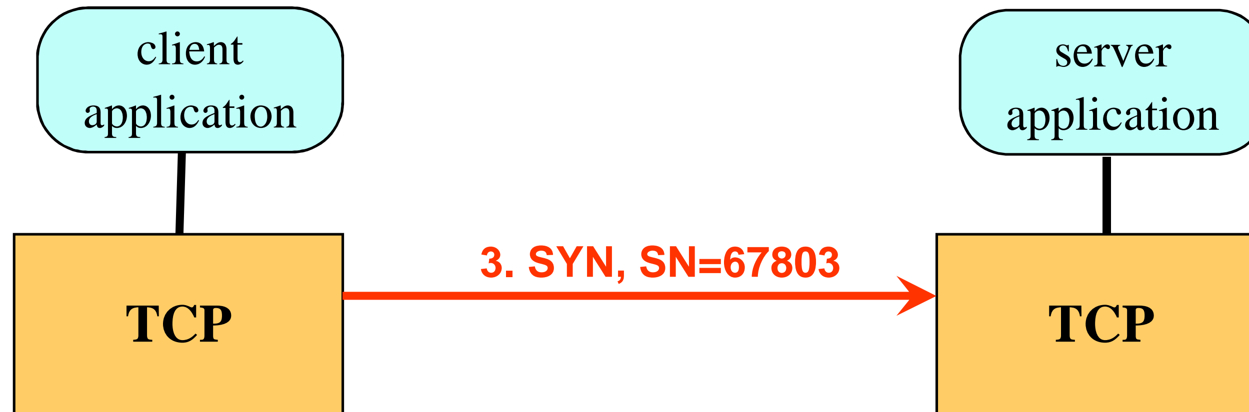
21	FTP signalling
20	FTP data
23	telnet
25	SMTP
53	DNS
80	HTTP

Connection Setup



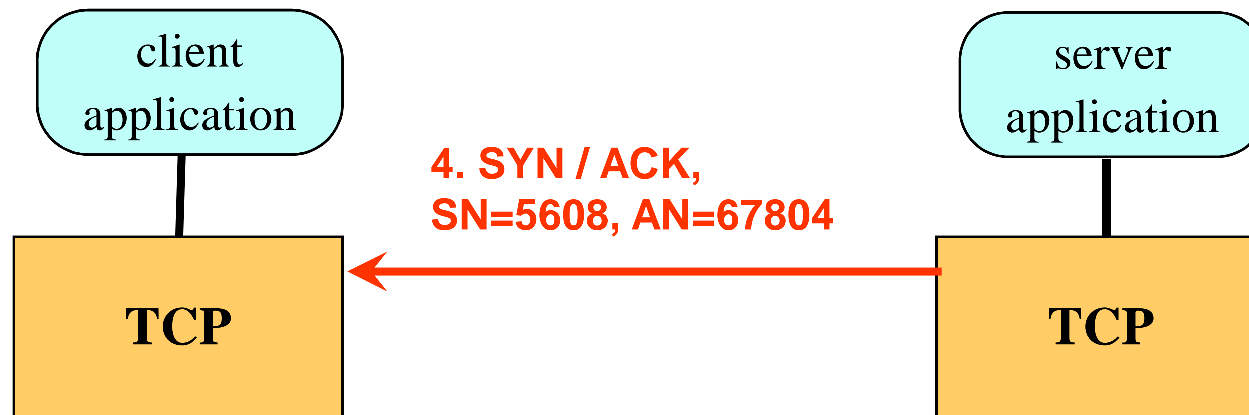
- Before *connection setup,* client-side applications and server-side must communicate with TCP software
  - 1. The *server* makes a *Passive Open*, to communicate to the TCP level that it is ready to accept new connections (on a given, known port)
  - 2. The *client* makes an *Active Open*, to communicate to the TCP that it is ready to start a new connection to a given socket

Connection Setup



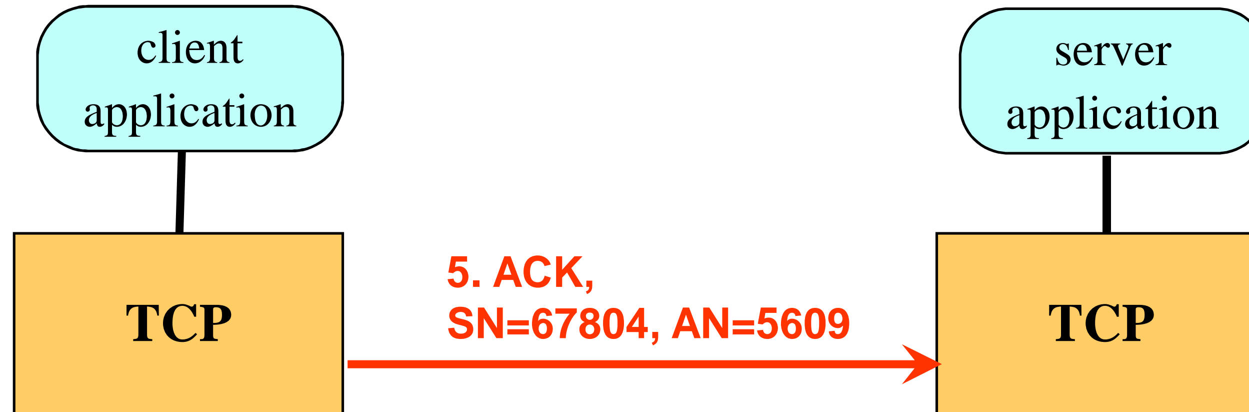
- The TCP client randomly chooses a sequence number (e.g., 67803) and sends out a SYNchronize (flag SYN=1) segment.
- Connection parameters/options can be indicated in such first segment (MSS, Windows Scaling factor…)

Connection Setup



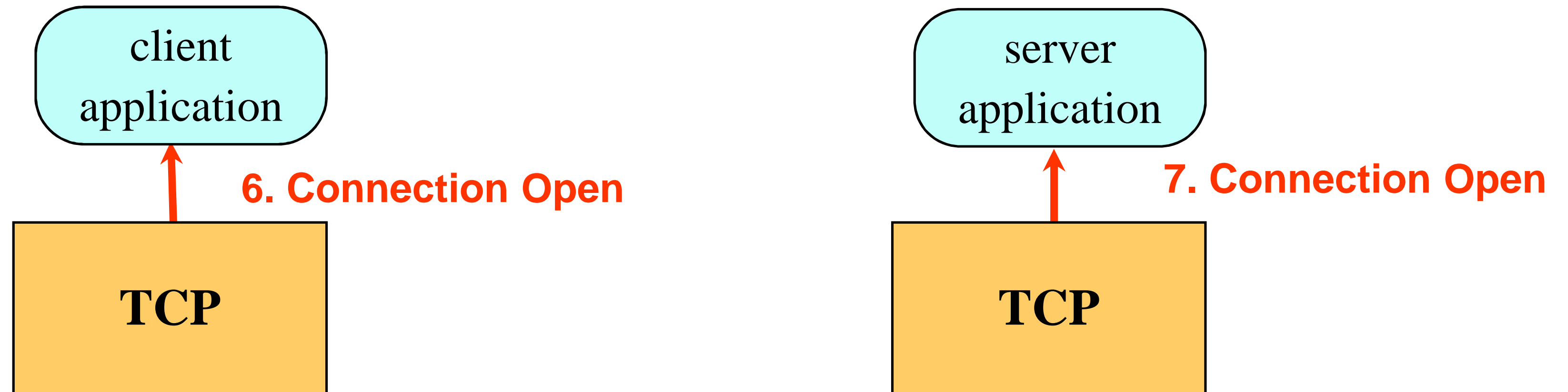
- Upon reception of the SYN, the TCP *server* chooses a sequence number (e.g., 5608) and sends out a SYN/ACK (flag SYN=1, flag ACK=1) segment containing an *acknowledgment number* to 67804 (ACK for the previous packet, 67803+1 = *next expected byte*)

Connection Setup



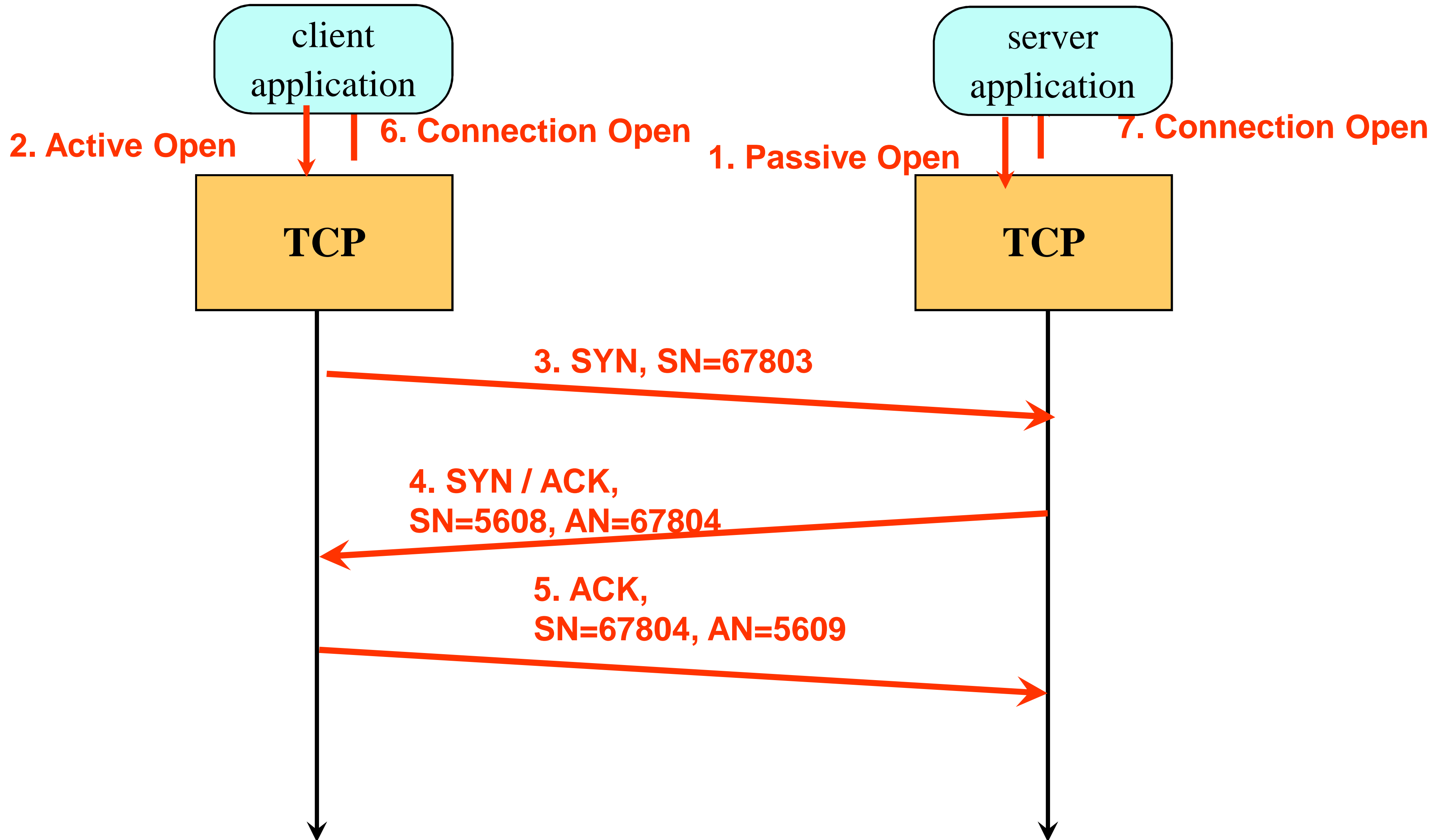
- The TCP *client* receives the SYN/ACK segment from the server, and sends out a ACK for 5609. The *payload* carries the first real data of the connection.
- The sequence number of the first byte in the payload is 67804. Also the server window is advertised.

Connection Setup

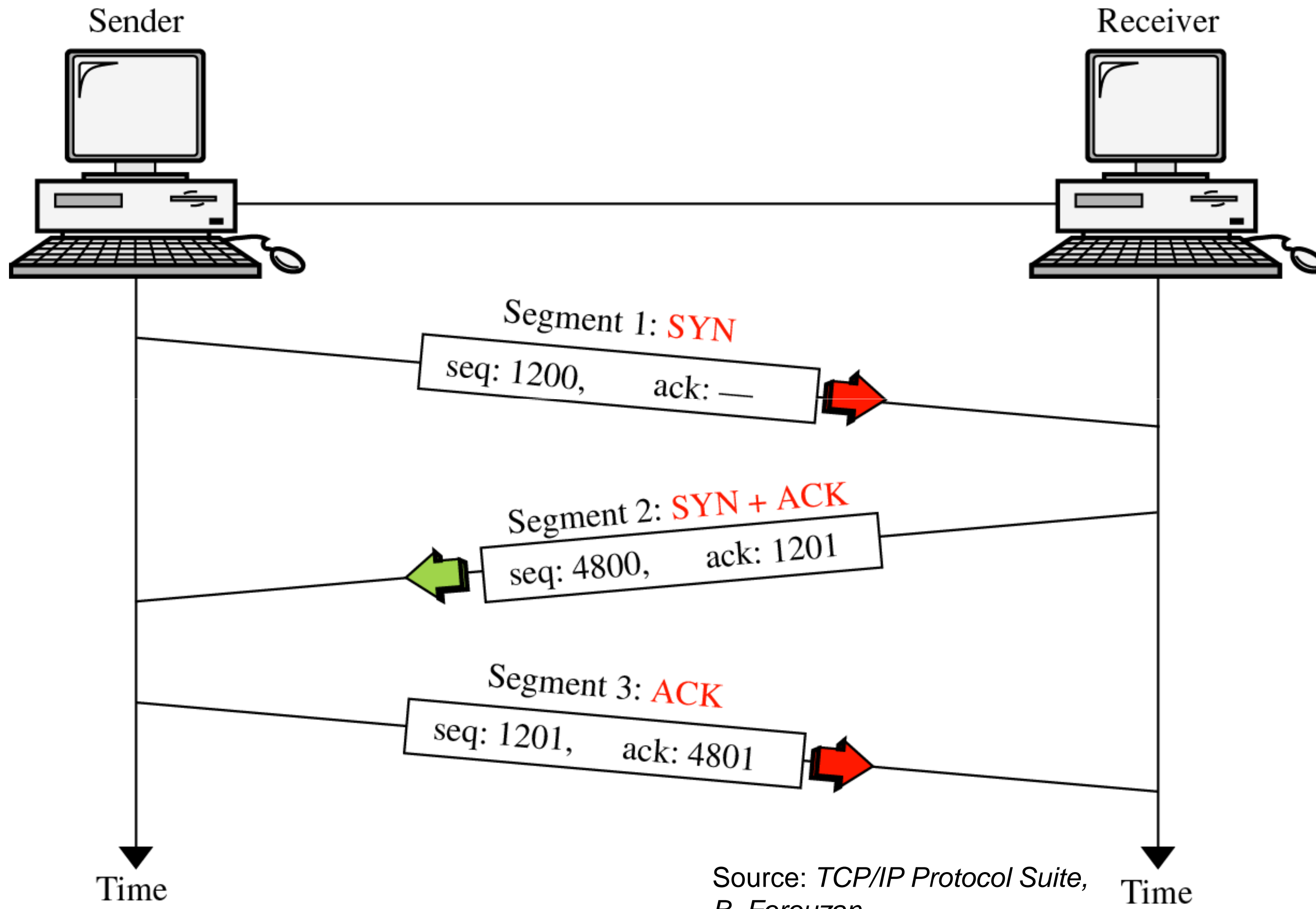


- The TCP client notifies to the application that the connection is open
- Upon reception of the ACK from the client, also the TCP server notifies to the server application that the connection is open

Connection Setup (summary)



Connection Setup

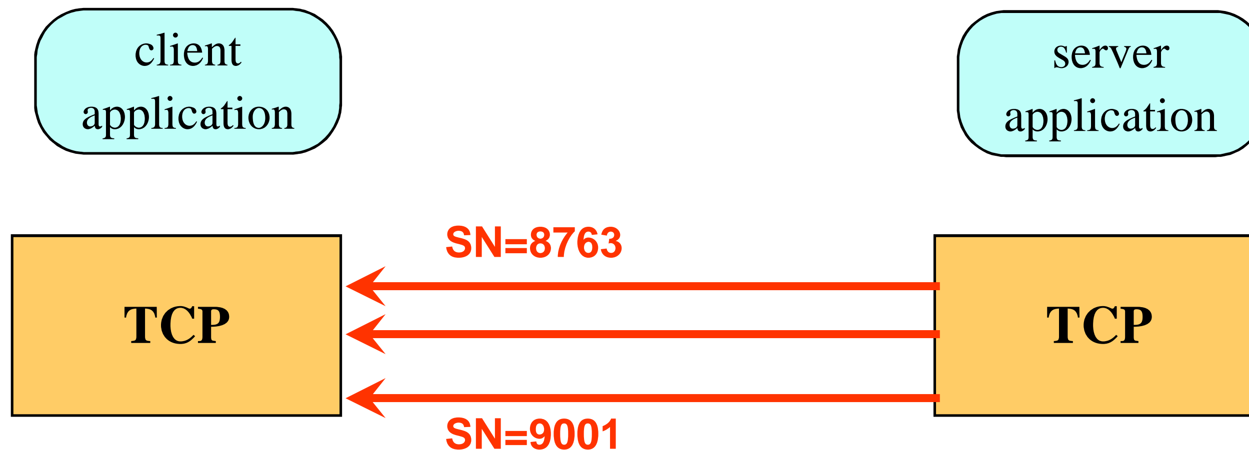


Source: *TCP/IP Protocol Suite, B. Forouzan.*

Connection Tear down

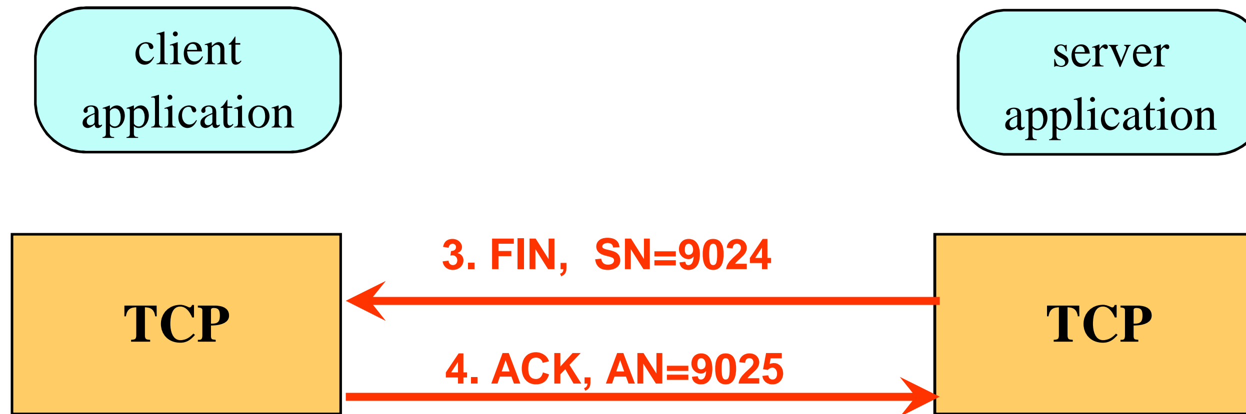
- The TCP willing to close the connection sends out a FIN (flag FIN=1) segment
- The peer TCP entity answers with a valid ACK

Connection Tear down



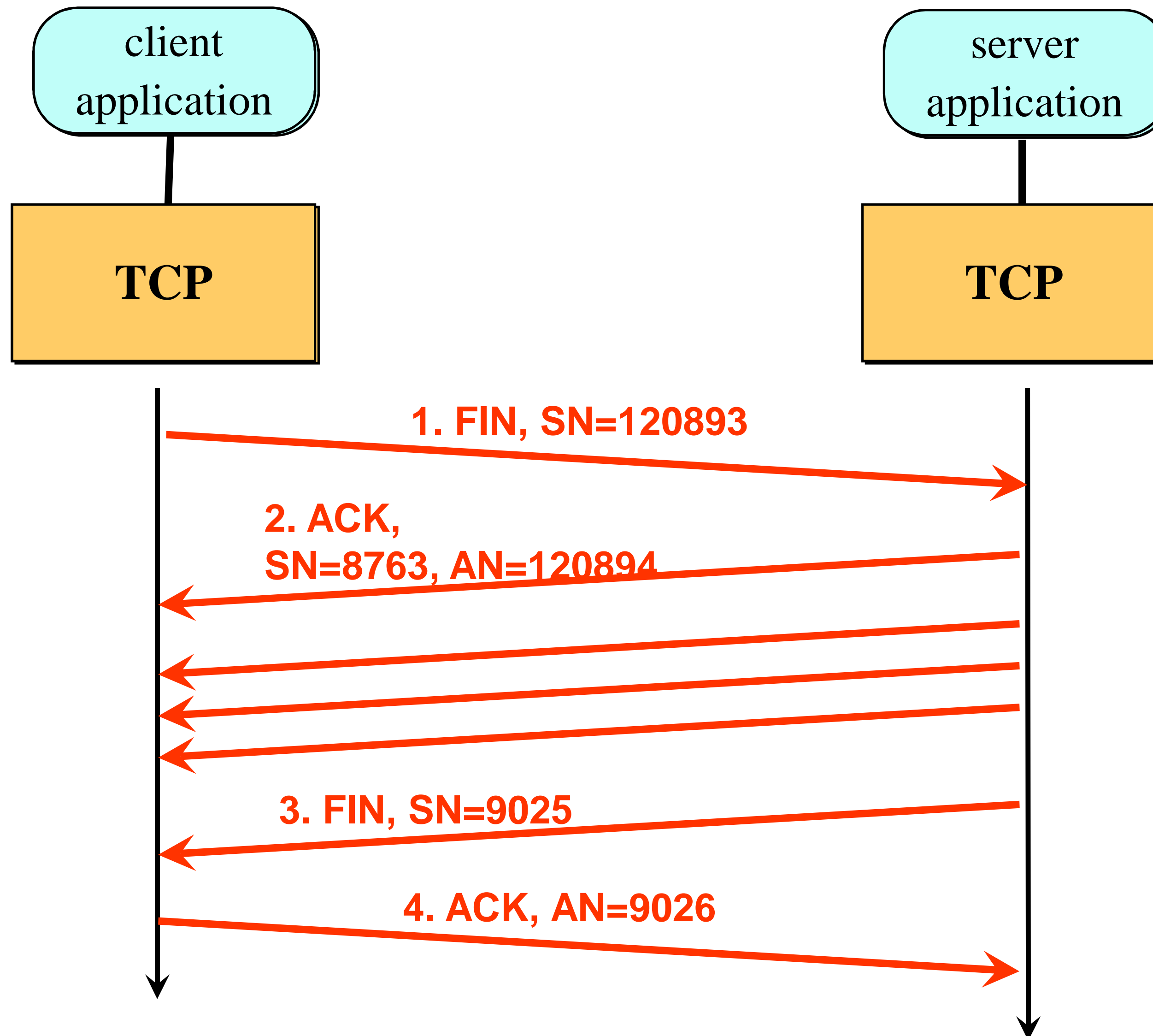
- The connection in the opposite direction remains open

Connection Tear down



- Finally, also the other TCP entity closes the connection with FIN (flag FIN=1)
- The connection is completely closed (in both directions) when the TCP entity answers with an ACK

Connection Tear down



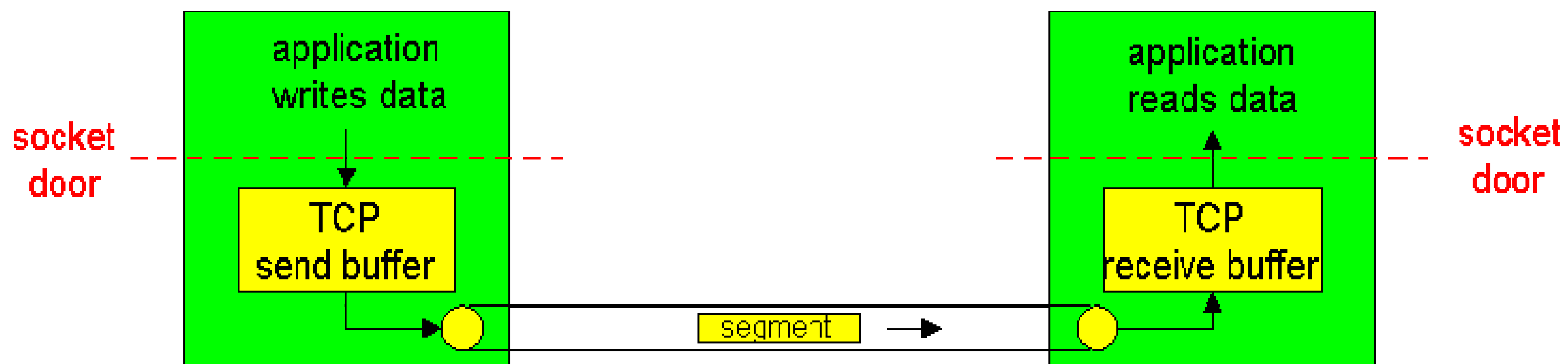
Connection Reset

- Connection can be closed without bidirectional messages exchange
- The RESET flag quits the connection in both directions
- The receiving TCP entity closes the connection

Flow Control Implementation

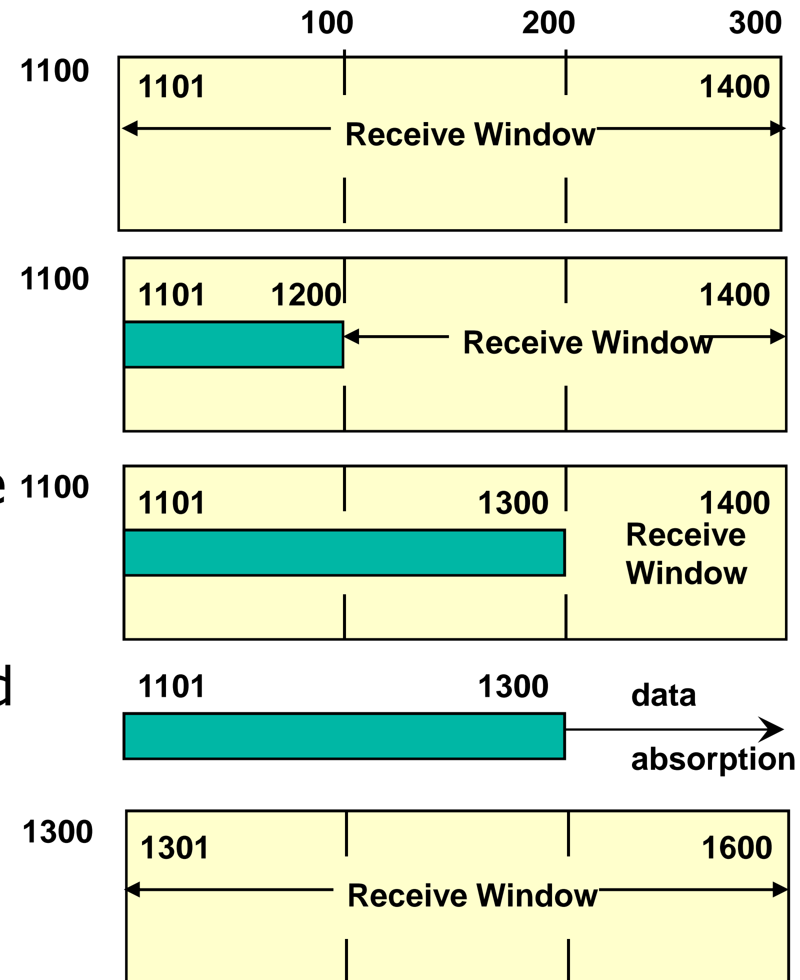
- Receiver oriented flow control

- Receiver's side:
  - Reception Buffer: stores received bytes to be absorbed by the application
- Transmitter's side:
  - Transmission Buffer: stores the bytes to be transmitted



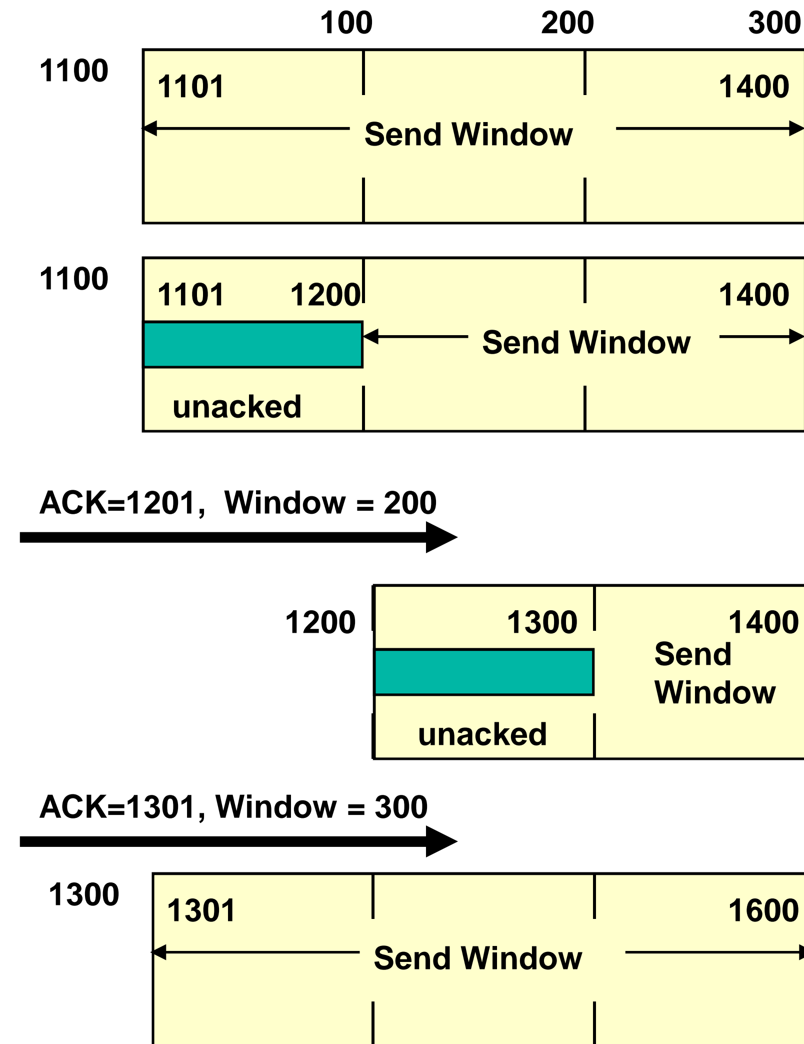
Flow Control: Receiver's Side

- *Receiver Window (RCVWND)*: portion of the reception buffer available for data storage
- Reception buffer may be filled due to congestion in the receiving application or OS
- RCVWND goes from the last byte absorbed by the application to the end of the reception buffer
- *RCVWND* dimension is advertised to the transmitter

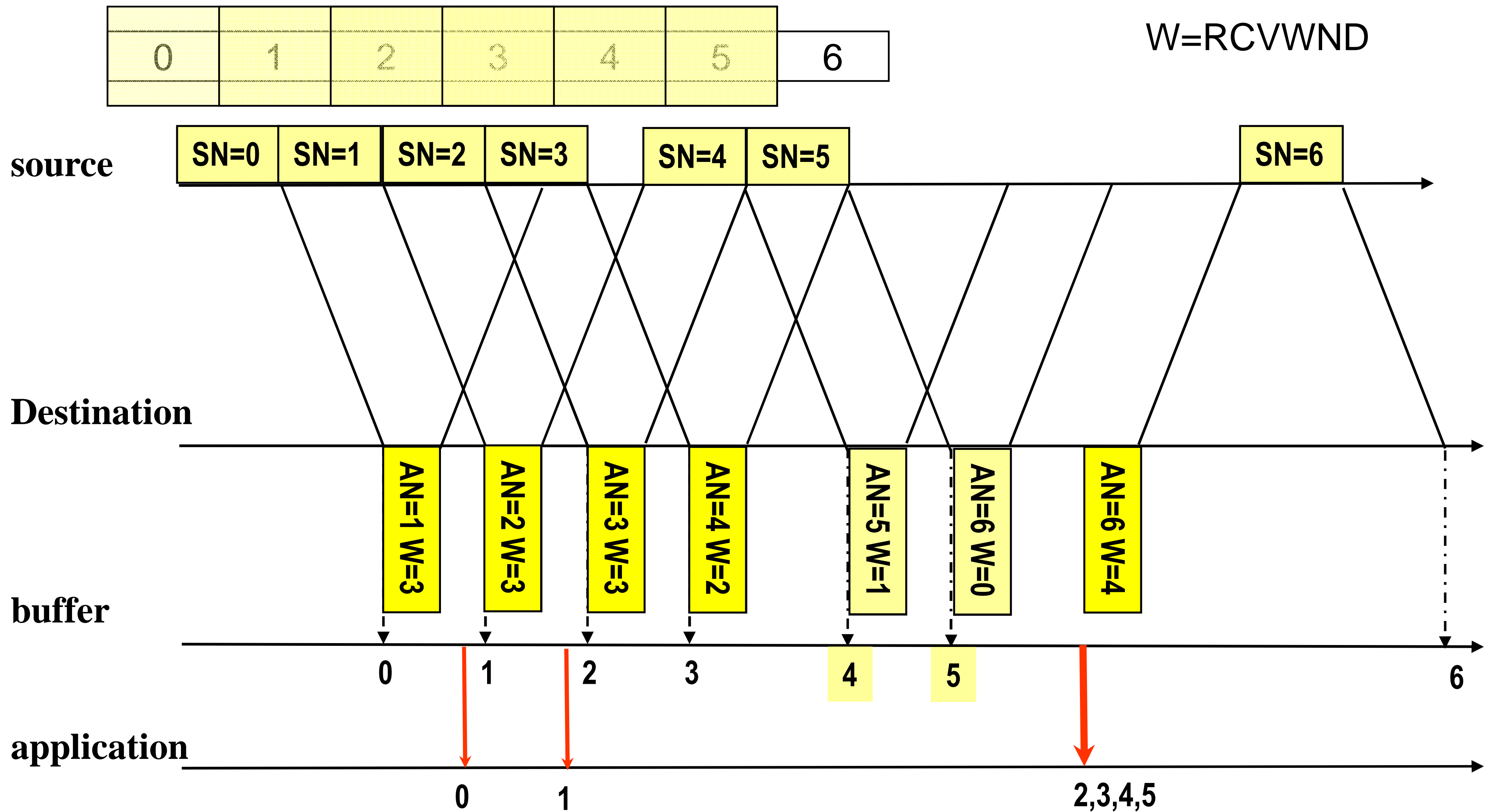


Flow Control: TX's Side

- Transmission buffer (Sliding Window)
  - Data waiting for acknowledgement
  - goes from the first un-acked byte to the dimension of RCVWND
- *Send Window (SNDWND) is the portion of the transmission buffer covering the bytes which can be transmitted*



Flow Control: An Example (W=4)



Flow Control Problems

- *Silly window syndrome* – receiver's side:
  - The receiver is slow in emptying the receiving buffer
  - Advertises very short RCVWND values
  - The transmitter sends short segments with high overhead
- *Clark's* Solution
  - The receiver advertises null RCVWND until the reception buffer is half empty or a MSS can be advertised

min (1/2 Receive_Buffer_Size, Maximum_Segment_Size).

Flow Control Problems

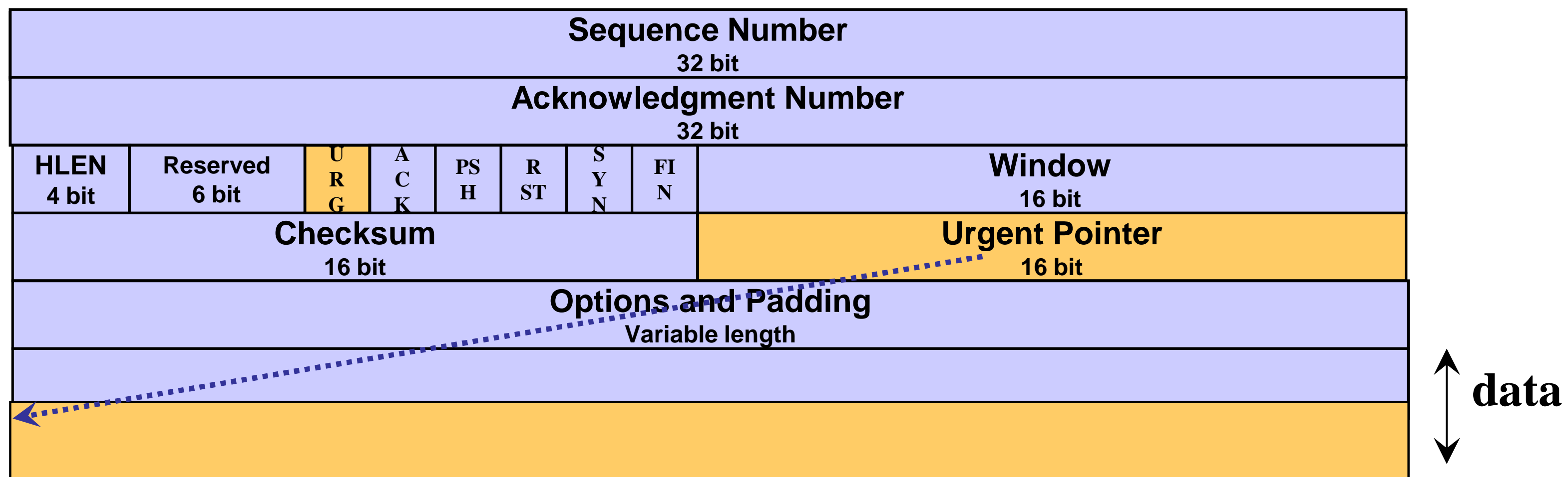
- *Silly window syndrome* – transmitter's side:
  - The application (OS) generates data slowly
  - The transmitter sends short segments (high overhead)
- *Nagle's Solution*
  - The first data are sent no matter of their dimension
  - All the following segments are generated and sent only if:
    - The transmission buffer contains enough data to fill a MSS
    - Upon reception of a valid ACK for a previously sent segment

Push

- The normal flow of bytes to be delivered to the application can be altered using the Push functionality
- An application can use a *push* command to notify the TCP about critical data (to be delivered promptly)
- The segment(s) carrying such data uses the push flag set
- E.g., Telnet

URGENT Data

- Data can be marked as URGENT
- *In-band* signaling mechanism to carry urgent data side by side with normal data



Error Control/Recovery

- Detection:
  - Corrupted Segments
  - Lost Segments
  - Duplicated Segments
  - Out-of-order Segments

  } **Checksum, Acknowledgements Timer/Time out**

- Correction
  - Retransmission
  - Discarding
  - Ordering

Error Control/Recovery

- Lost segments are mainly due to queue overflow in routers (except in Wireless networks !!!)
- TCP implements a *Go-back-N* like retransmission mechanism with *Timeout*
- The transmission window is the sliding window used for the flow control
- A timeout is set for <u>every</u> sent segment

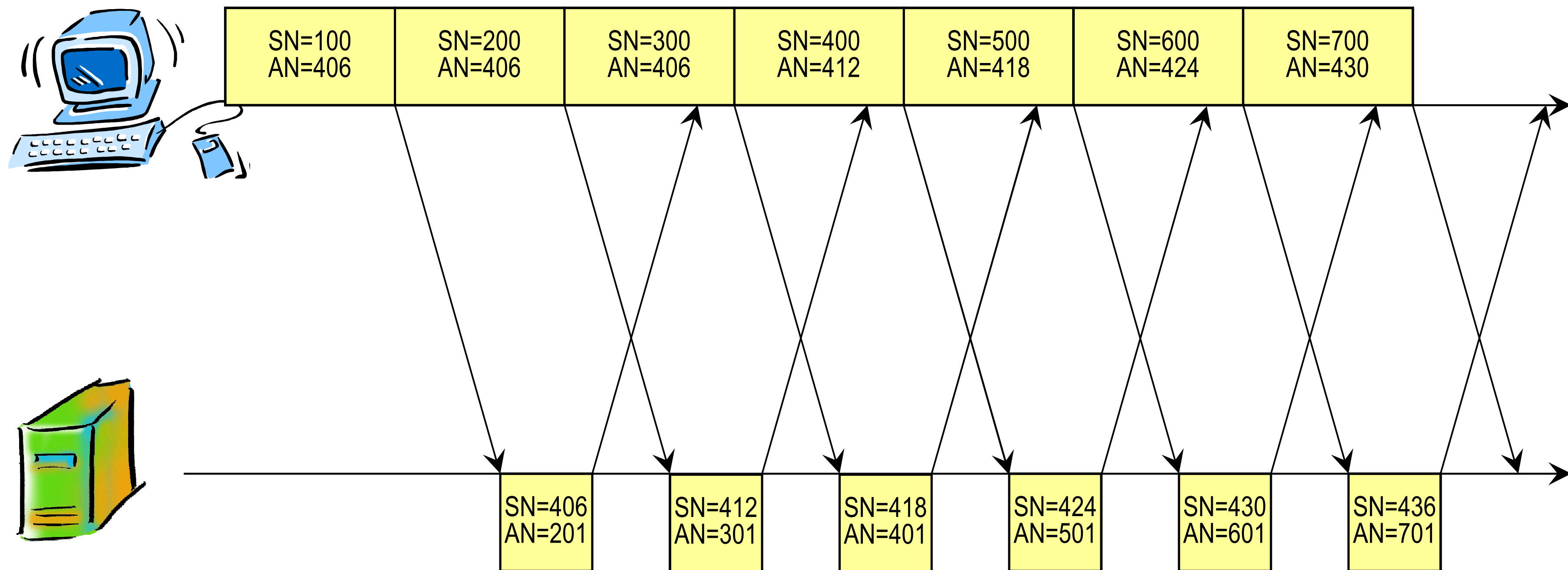
Error Control

Example 1: errorless

MSS=100 byte

Window= 4 MSS

Each packet contains 100 bytes



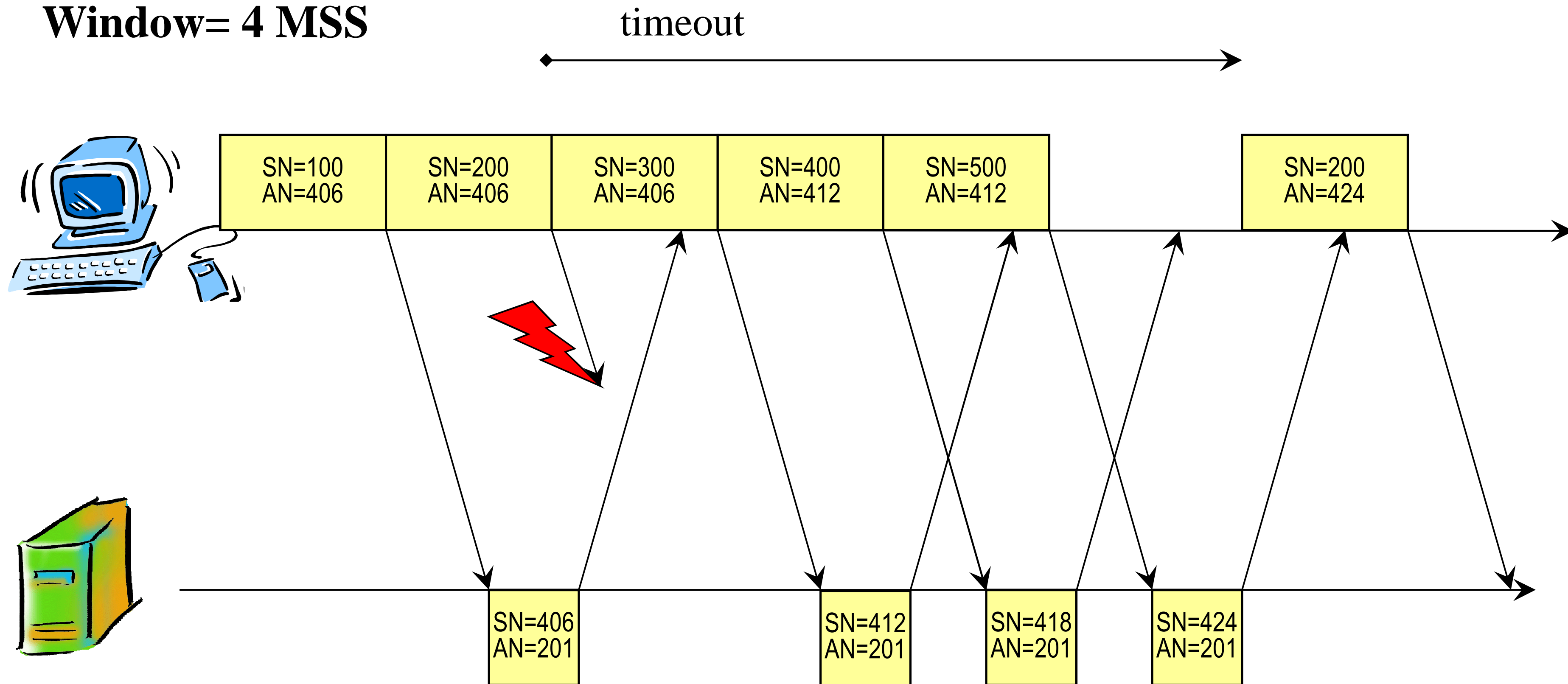
In this example, each ACKnowledgmenet contains 6 bytes

Error Control

Ex 2: error on segments

MSS=100 byte

Window= 4 MSS

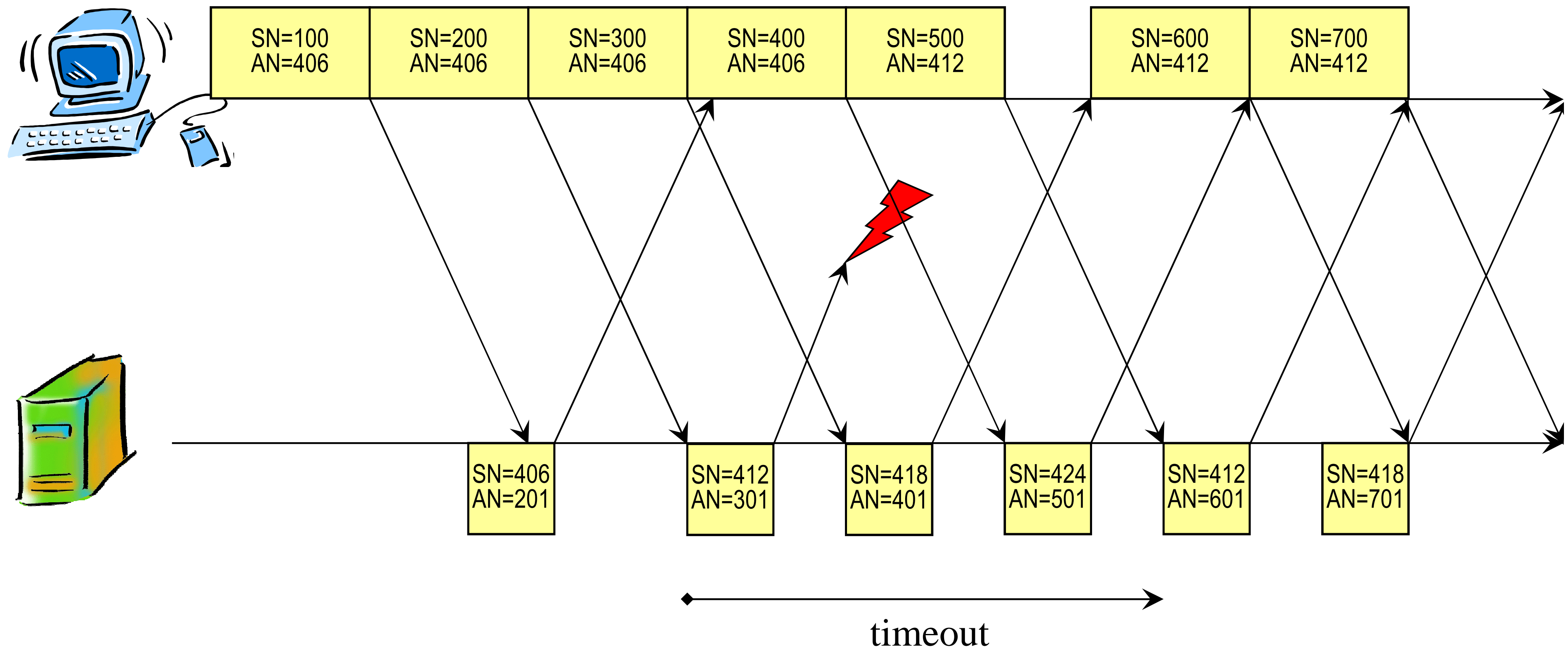


Error Control

Ex 3: error on ACKs

MSS=100 byte

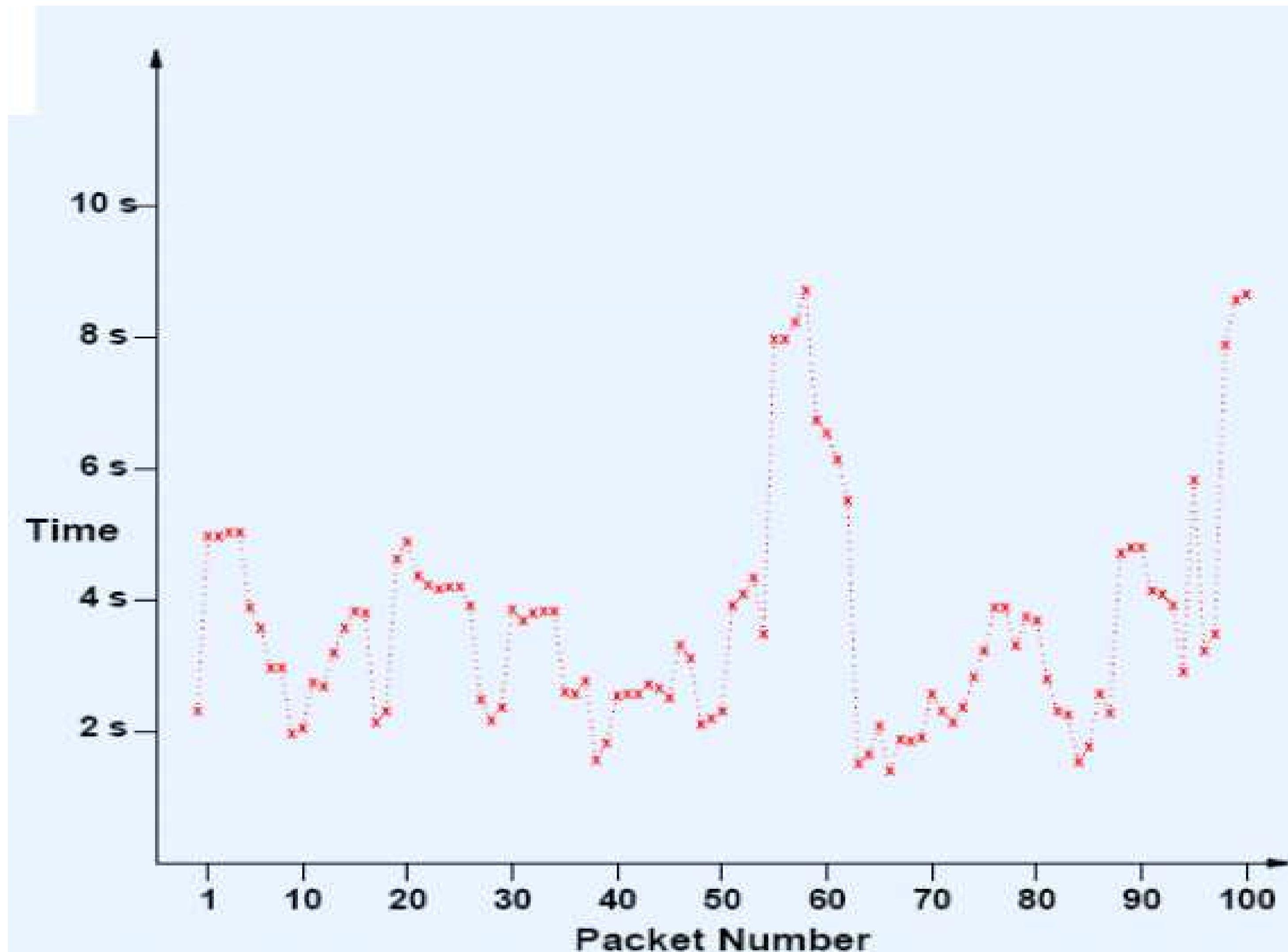
Window= 4 MSS



Time-Out Management

- How to set the timeout:
  - Too short: the transmitter will overflow the communication with (useless) retransmitted segments
  - Too long: slow recovery from errors
- The optimum depends on the delay of the connection (local networks, satellite networks)
- TCP sets the timeout by estimating the RTT *(Round Trip Time)*

RTT Variability



RTT Estimation

- Karn and Jacobson algorithms are used to estimate the RTT
- RTT samples $\{RTT^{(i)}\}$ are collected for every segment (ACK reception - Transmission time)

Mean Value Estimation

- TCP calculates the *Smoothed Round Trip Time* (SRTT) using Jacobson's formula

$$SRTT^{(i)} = (1-\alpha)\, SRTT^{(i-1)} + \alpha\, RTT^{(i)}.$$

- where α ranges from 0 to 1 (practical value is 1/8)

RTT Estimation

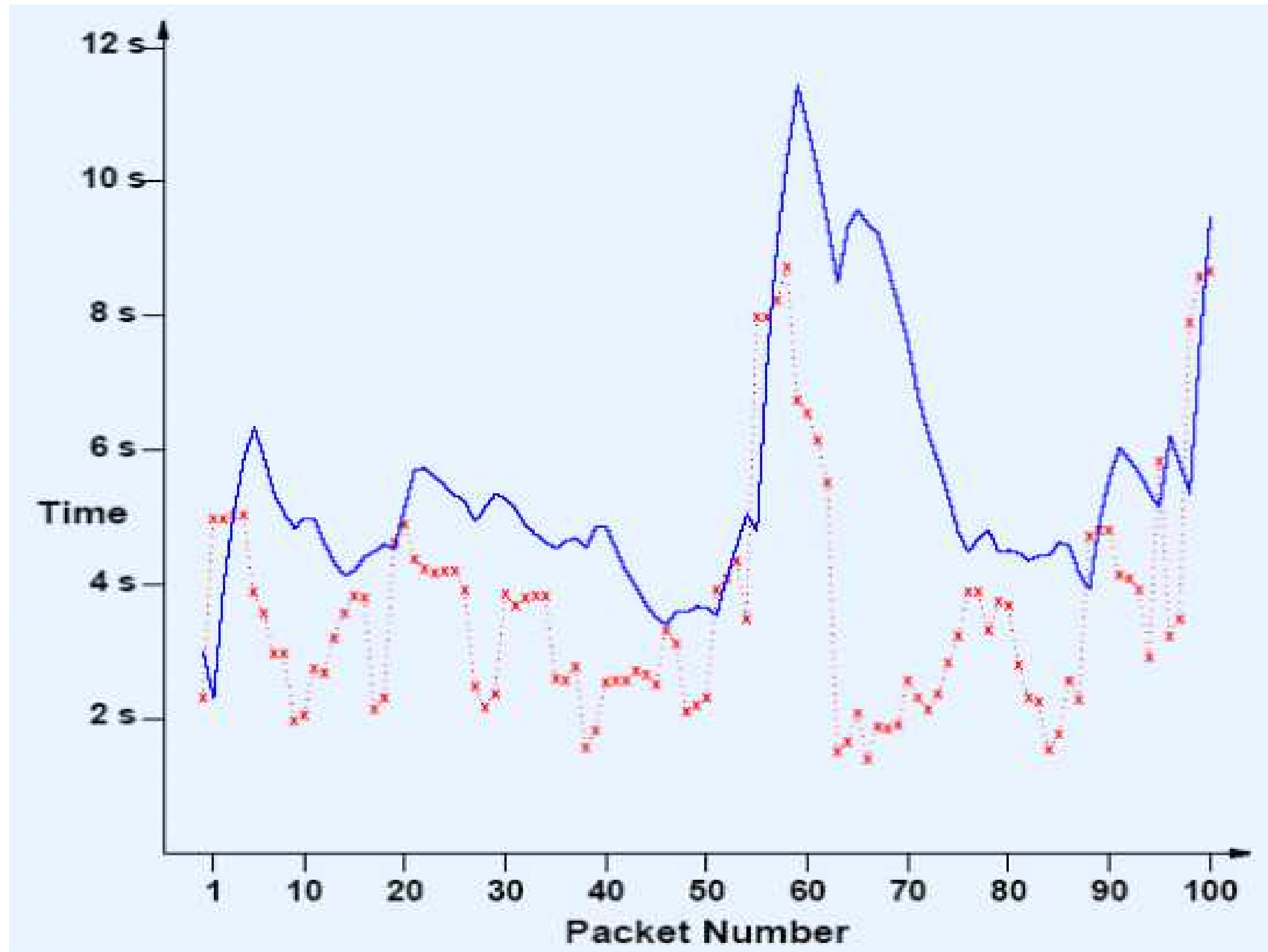
Standard Deviation estimation

$$DEV = |RTT^{(i)} - SRTT^{(i-1)}|$$

- A smoothed value of the standard deviation is calculated:

$$SDEV^{(i)} = 3/4\ SDEV^{(i-1)} + 1/4\ DEV$$

RTT Estimate Quality



Time Out Calculation

- Timeout is given by:

$$TIMEOUT = SRTT + 2\ SDEV$$

- Initial conditions
  - SRTT(0) = 0
  - SDEV (0) = 1.5 s
  - Timeout =3 s

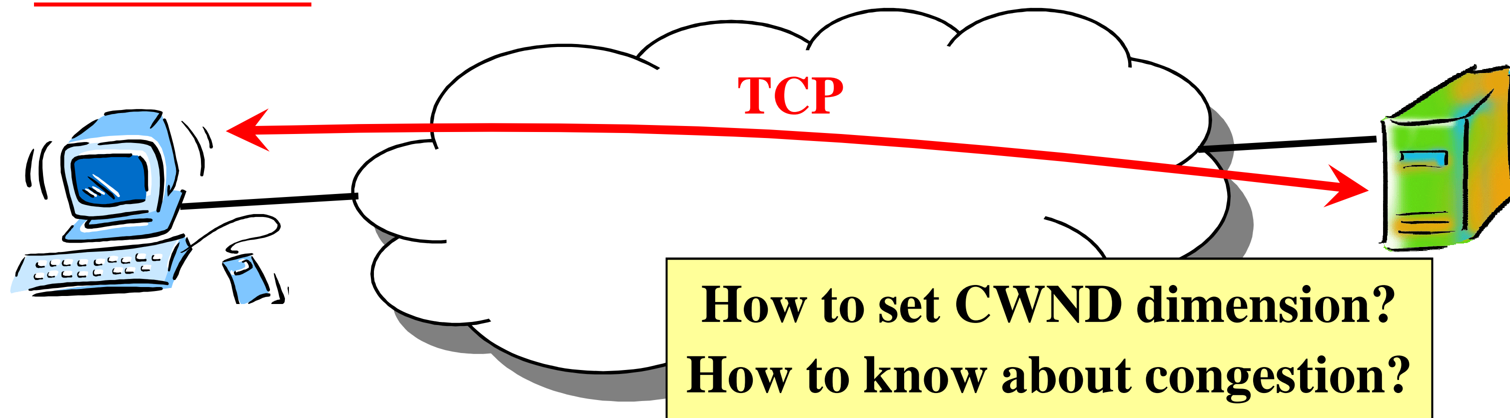
- If a segment is retransmitted, the *Karn's* algorithm is adopted to set the timeout:
  - RTT estimate does not get updated
  - The timeout is multiplied by 2
  - The same rule is applied for consecutive retransmissions
  - A maximum number of retransmissions is defined

Congestion Control

- Flow Control
  - Depends on the receiver's "capacity" only
  - Is not effective in avoiding *network* congestion
- The INTERNET does not implement congestion control mechanisms at the network layer (e.g. traffic admission control)
- Congestion control is delegated to the TCP!!!
- Since TCP runs end-to-end, the congestion control in the Internet is also *end-to-end*

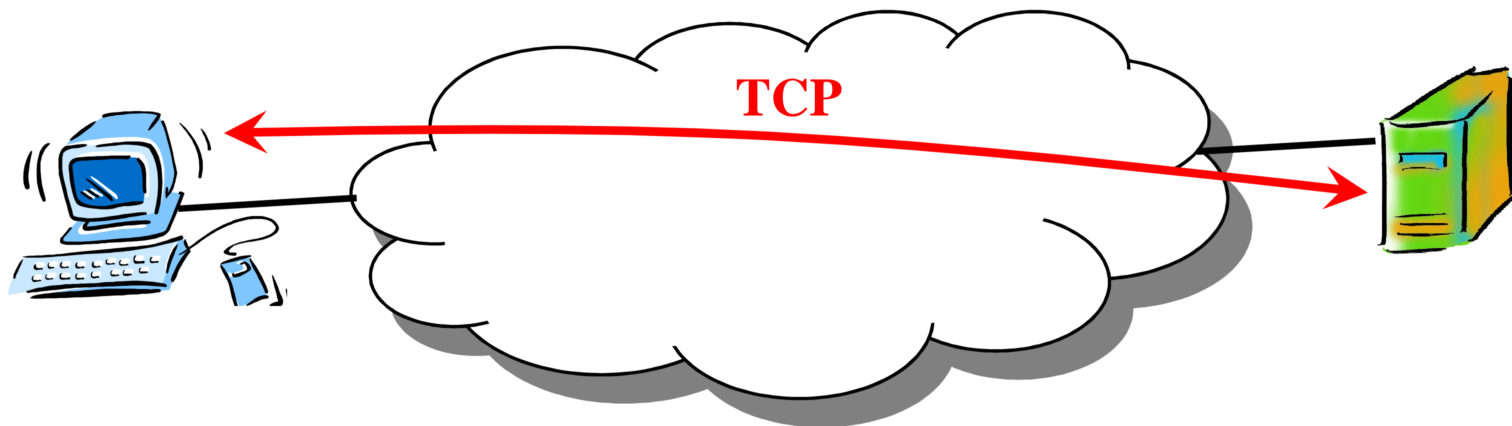
Congestion Control

- The congestion control is implemented through a sliding window (again)
- A *Congestion Window (CWND)* is kept by the transmitter
- CWND dimension depends on the status of the network as perceived by the transmitter (ACKs, expired timeout)
- The transmitter cannot transmit more than the *minimum* between RCVWND and CWND



Congestion Control

- A segment loss is interpreted by the TCP as a congestion event
- TCP reacts to such event by ***reducing*** the CWND dimension

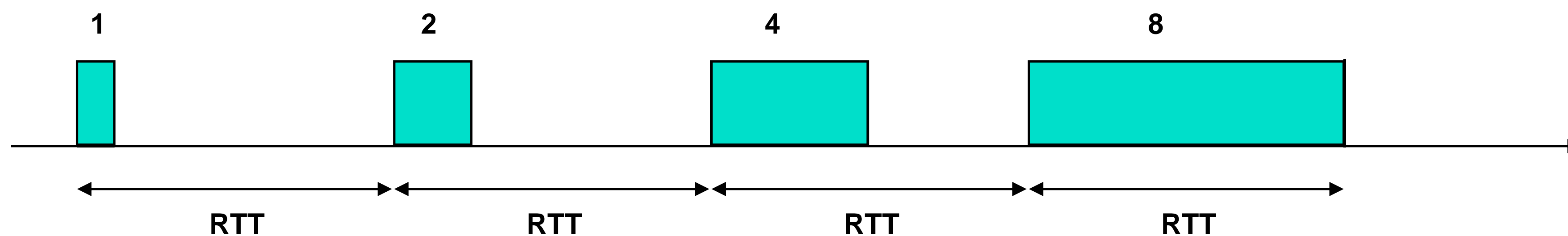


Slow Start & Congestion Avoidance

- The value of CWND dimension is regulated through an algorithm
- The update rule depends on the communication phase
- Two phases are defined:
  - *Slow Start*
  - *Congestion Avoidance*
- The transmitter keeps the variable SSTHRESH to distinguish between the two phases:
  - → if CWND < SSTHRESH: *Slow Start*
  - → if CWND >= SSTHRESH: *Congestion Avoidance*

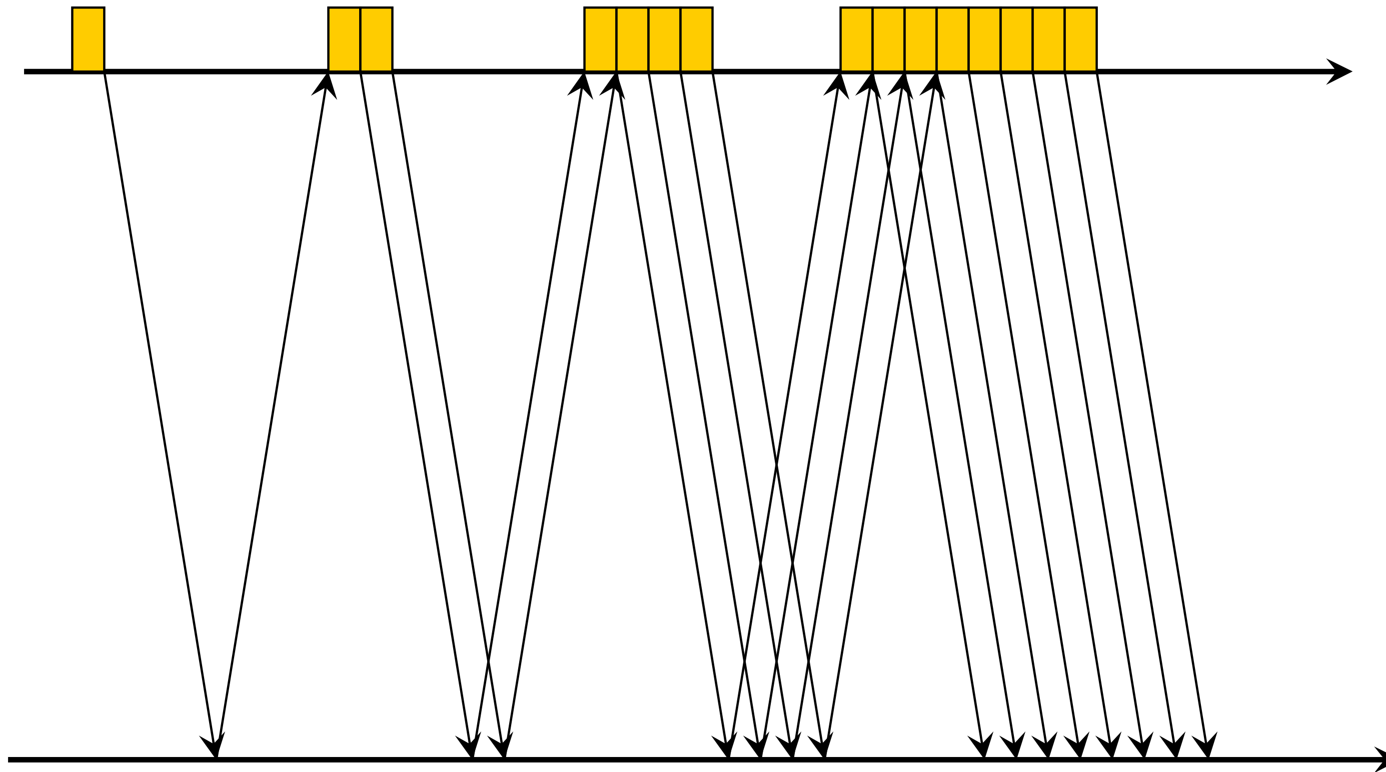
Slow Start

- In the beginning, CWND is set to 1 segment (MSS), and SSTHRESH to a (very) higher *default* value *("infinite")*
- Since CWND < SSTHRESH, connection is in *Slow Start*
- *In Slow Start:*
  - *CWND is incremented by 1 for each received ACK (exponential increase)*



Slow Start

- Exponential increase of the transmission rate (not so *slow*, indeed!)



Slow Start

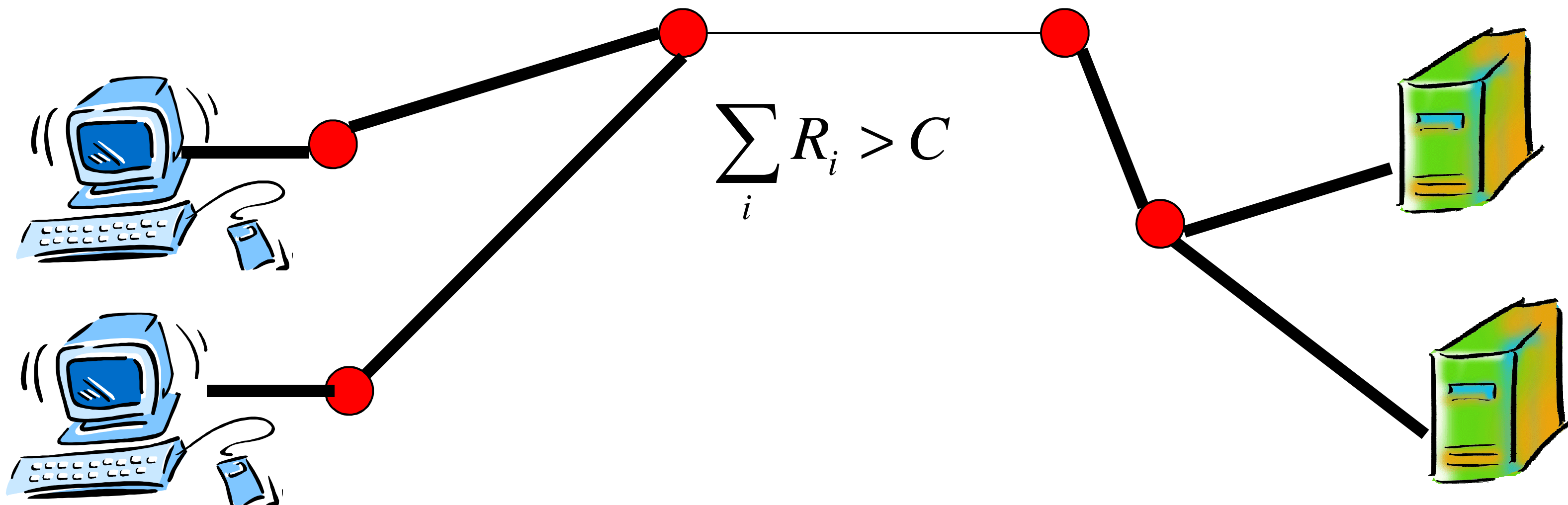
- Such phase goes on until
  - The first congestion event (segment loss)
  - CWND < SSTHRESH
  - CWND < RCWND

- The average transmission rate can be estimated as:

$$R = \frac{CWND}{RTT} \quad \text{[bit/s]}$$

Congestion Event

- One link on the path to destination gets congested
- The sum of the traffic flow using that link is higher than the link capacity
- Overflow and loss in some queue



Congestion Event

- Effect: a timeout expires
  - TCP *first* updates the SSTHRESH value according to the following equation

$$SSTHRESH = \max\left(2MSS, \frac{\mathrm{CWND}}{2}\right)$$

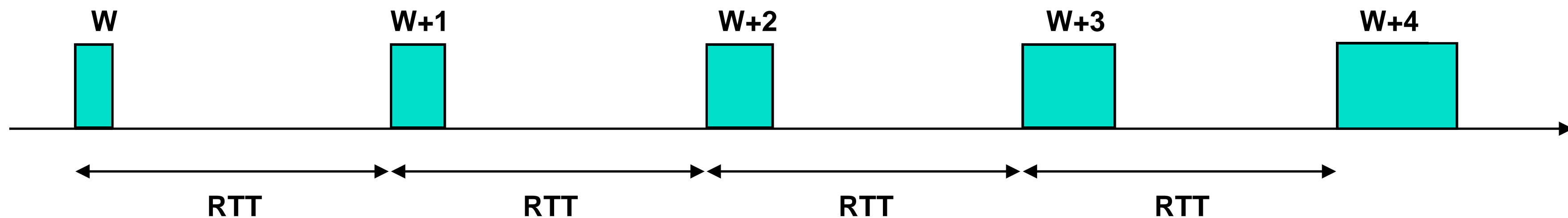
  - And *then* sets CWND = 1

Congestion Event

- As a result:
  - CWND (equal to 1) is lower than SSTHRESH (which is now >=2) and the connection falls back to the Slow Start phase
- Obviously, all lost packets are retransmitted (go-back-N) starting from the packet from which the timeout has expired (the lost one)
- The SSTHRESH value is (ideally) an indicator of the optimum CWND to avoid future congestion events

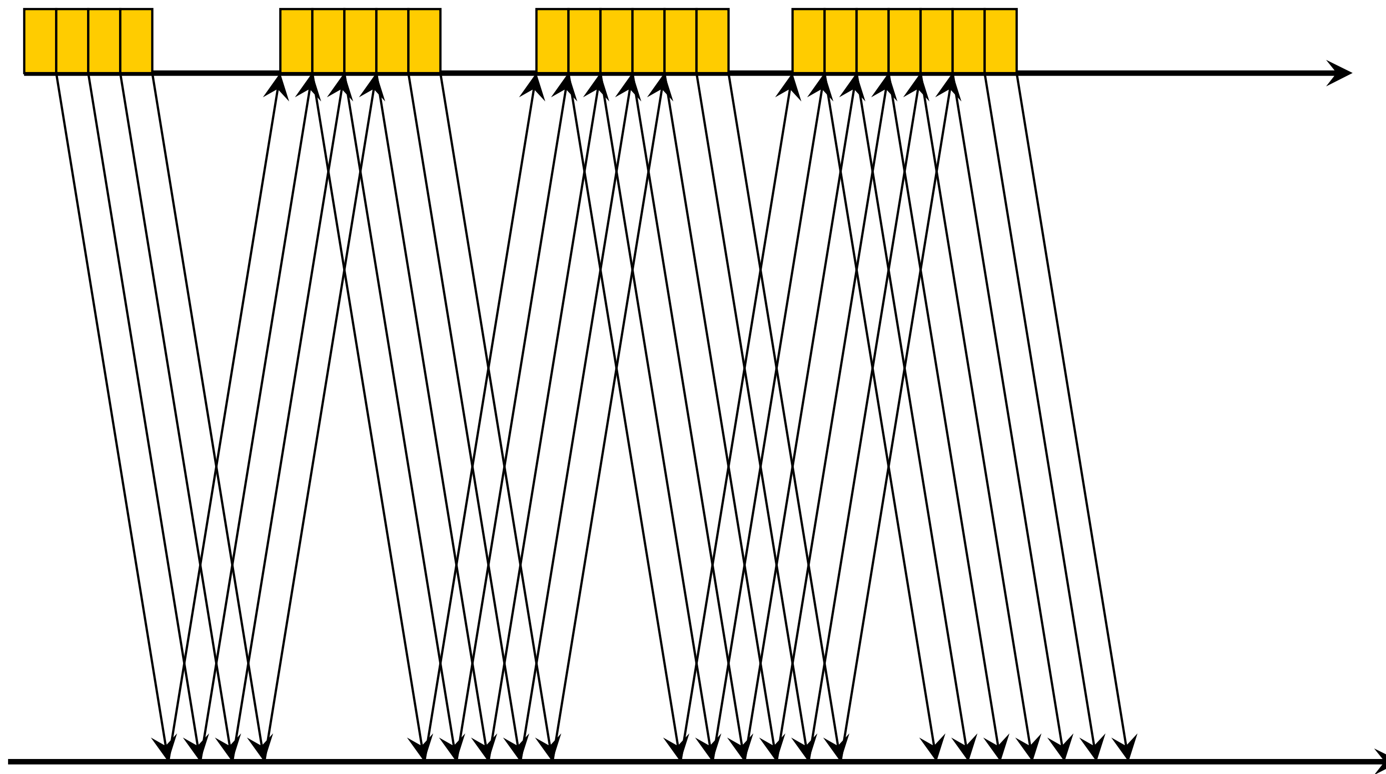
Congestion Avoidance

- Slow start goes on until CWND = SSTHRESH, after that the *Congestion Avoidance* phase starts
- In *Congestion Avoidance*:
  - CWND is incremented by 1/CWND for each received ACK (linear increase)
- In other words, if all the segments of the transmission window are acknowledged, the next transmission window size is augmented by 1

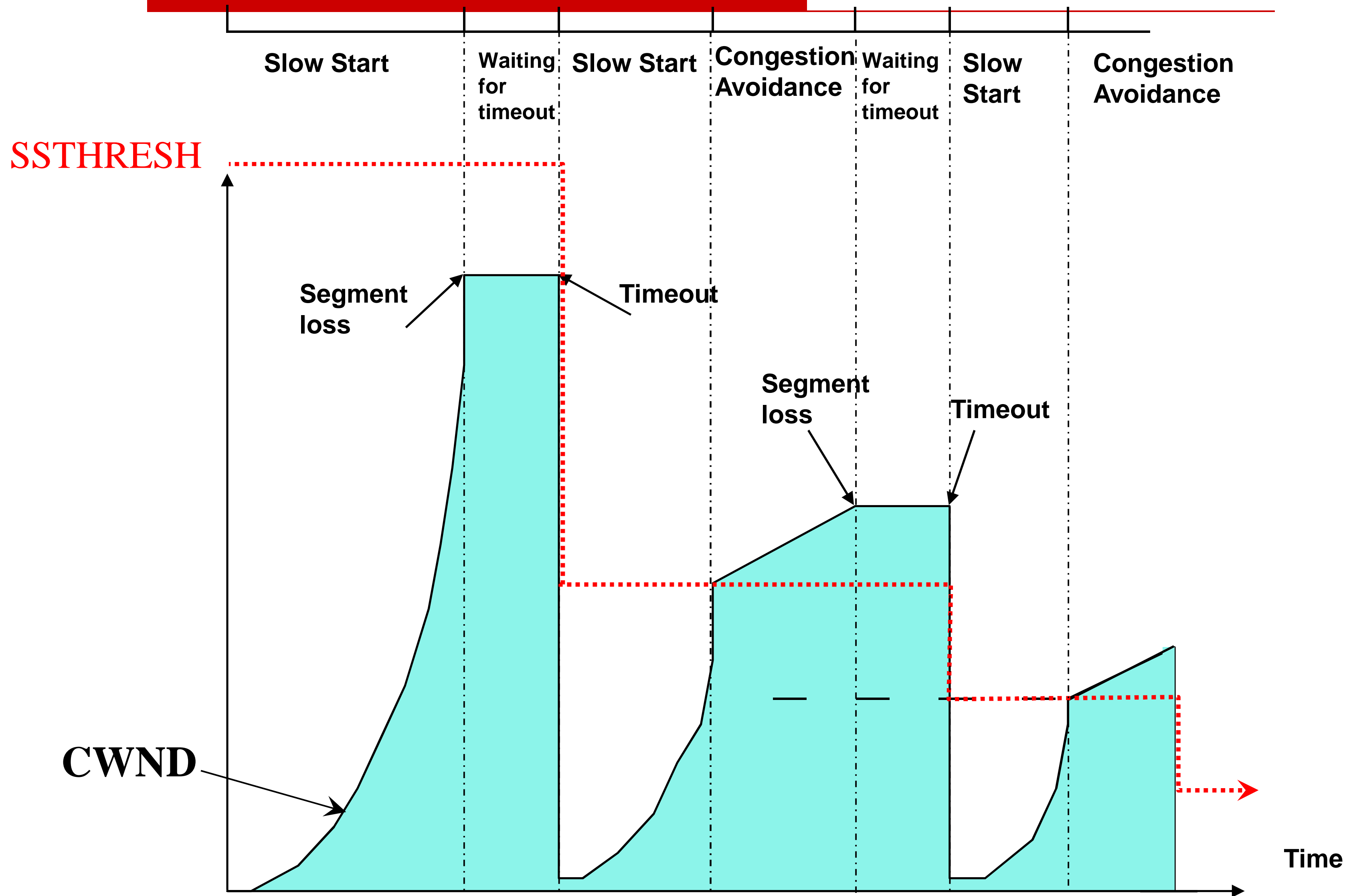


Congestion Avoidance

- Linear window increase



TCP Connection Lifetime: an example



Fast Retransmit and Fast Recovery

- Modification to the TCP operation implemented in TCP Reno
- Duplicated ACKs:
  - If the TCP receives *out of order* segments, it advertises the numbers of the missing one(s)
- Dup-ACKs may be due to the loss of single segments
- *Fast retransmit* and *Fast recovery* try to cope rapidly with such losses

Fast Retransmit and Fast Recovery

- Concept:
  - If dup-ACKs are received, only one segment has been lost
  - All the following segments have been received correctly (low congestion probability)
  - CWND can be incremented by the number of segments arrived at destination (dup-ACKs)
- Problems:
  - Multiple losses *cannot* be recovered

Fast Retransmit and Fast Recovery

1. Upon reception of the 3rd dup-ACK:

$$SSTHRESH = \max\left(\frac{\text{FlightSize}}{2}, 2MSS\right)$$

2. The missing segment is fast retransmitted
3. $$CWND = SSTHRESH + 3 \cdot MSS$$
4. For each dup ACK received afterwards, CWND is incremented by 1
5. New segments are transmitted if allowed by the CWND and RWND values
6. Upon reception of a ACK for new data the fast recovery quits and:

$$CWND = SSTHRESH = \max\left(\frac{\text{FlightSize}}{2}, 2MSS\right)$$