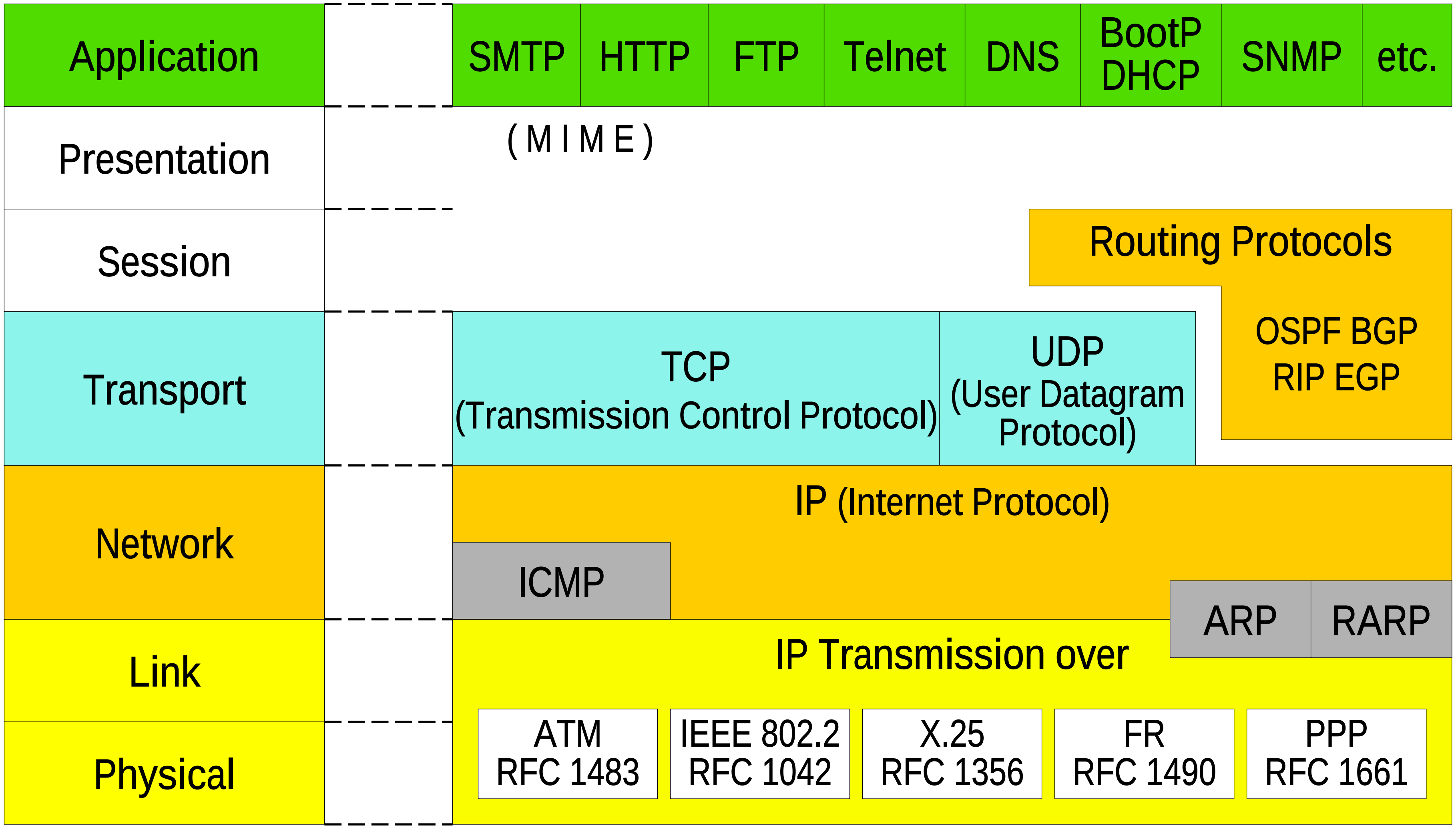
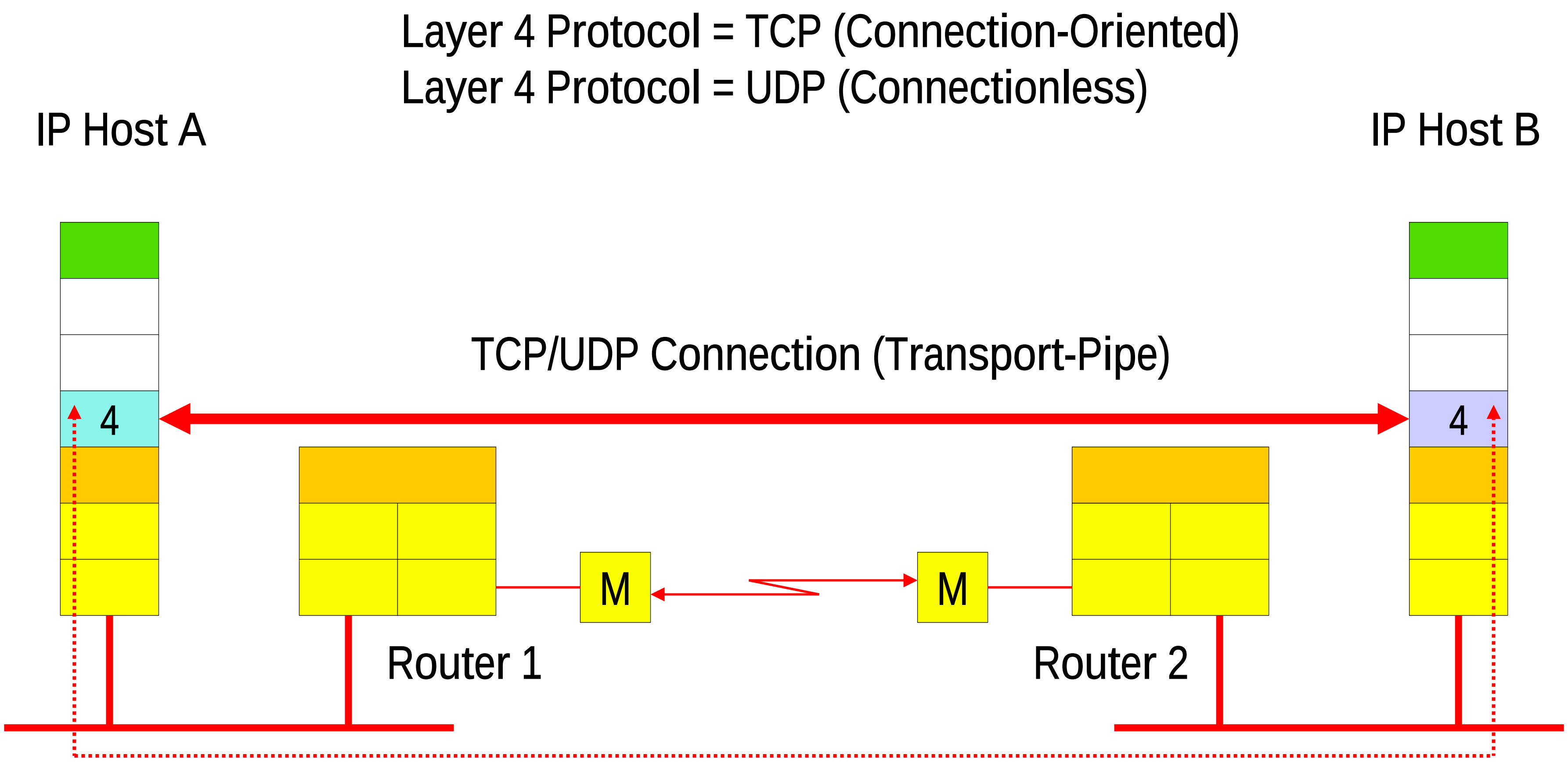
Introducing TCP & UDP

Internet Transport Layers

TCP/IP Protocol Suite



TCP/UDP and OSI Transport Layer 4



TCP Facts (1)

- Connection-oriented layer 4 protocol
- Carried within IP payload
- Provides a reliable end-to-end transport of data between computer processes of different end systems
  - Error detection and recovery
  - Sequencing and duplication detection
  - Flow control
- RFC 793

TCP Facts (2)

- Application's data is regarded as continuous byte stream
- TCP ensures a reliable transmission of segments of this byte stream
- Handover to Layer 7 at "Ports"
  - OSI-Speak: Service Access Point

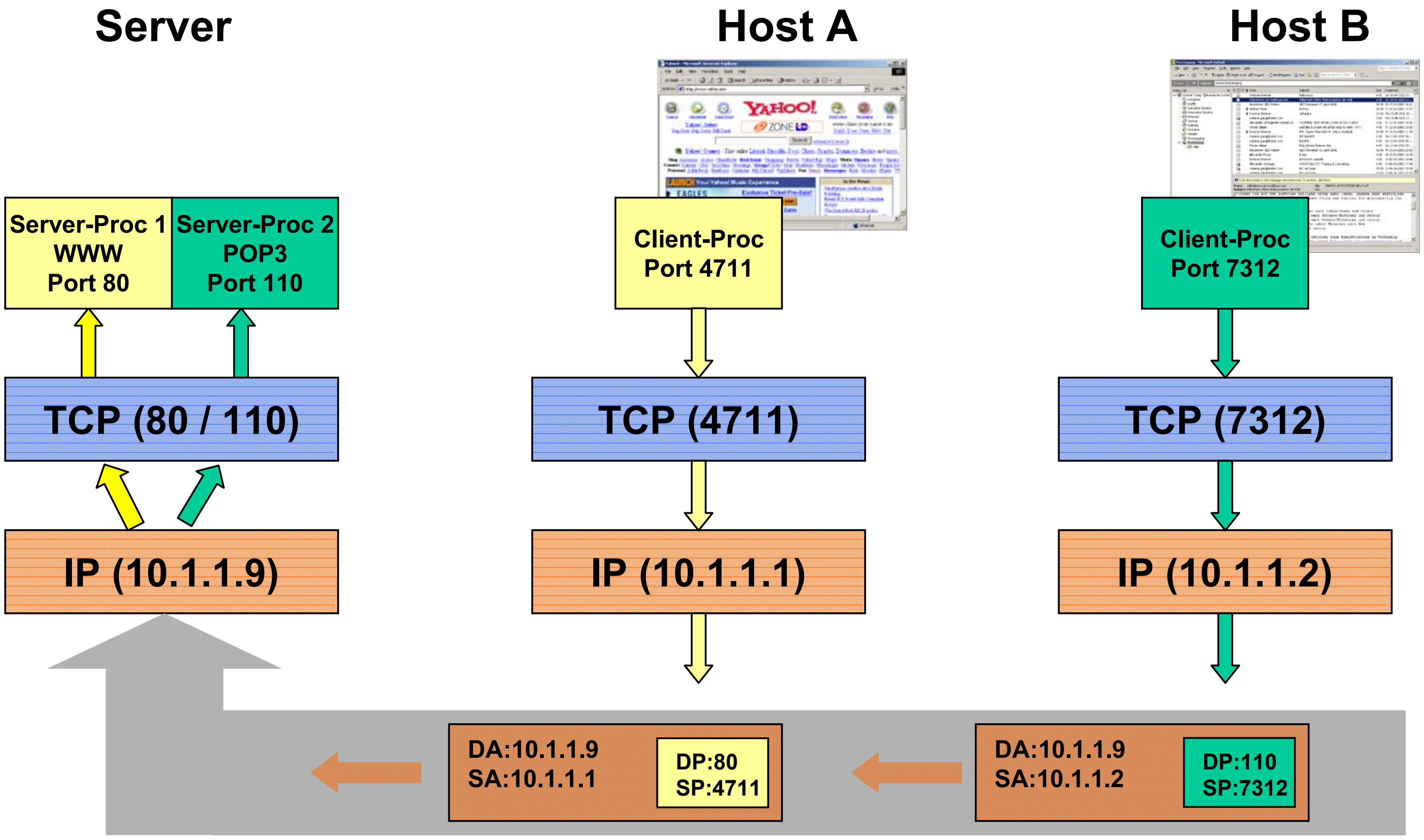
Port Numbers

- Using port numbers TCP (and UDP) can **multiplex** different layer-7 byte streams
- Server processes are identified by **Well known** port numbers : 0..1023
  - Controlled by IANA
- Client processes use arbitrary port numbers >1023
  - Better >8000 because of registered ports

Registered Ports

- For proprietary server applications
- Not controlled by IANA only listed in RFC 1700
- Examples
  - 1433 Microsoft-SQL-Server
  - 1439 Eicon X25/SNA Gateway
  - 1527 Oracle
  - 1986 Cisco License Manager
  - 1998 Cisco X.25 service (XOT)
  - 6000-6063 X Window System

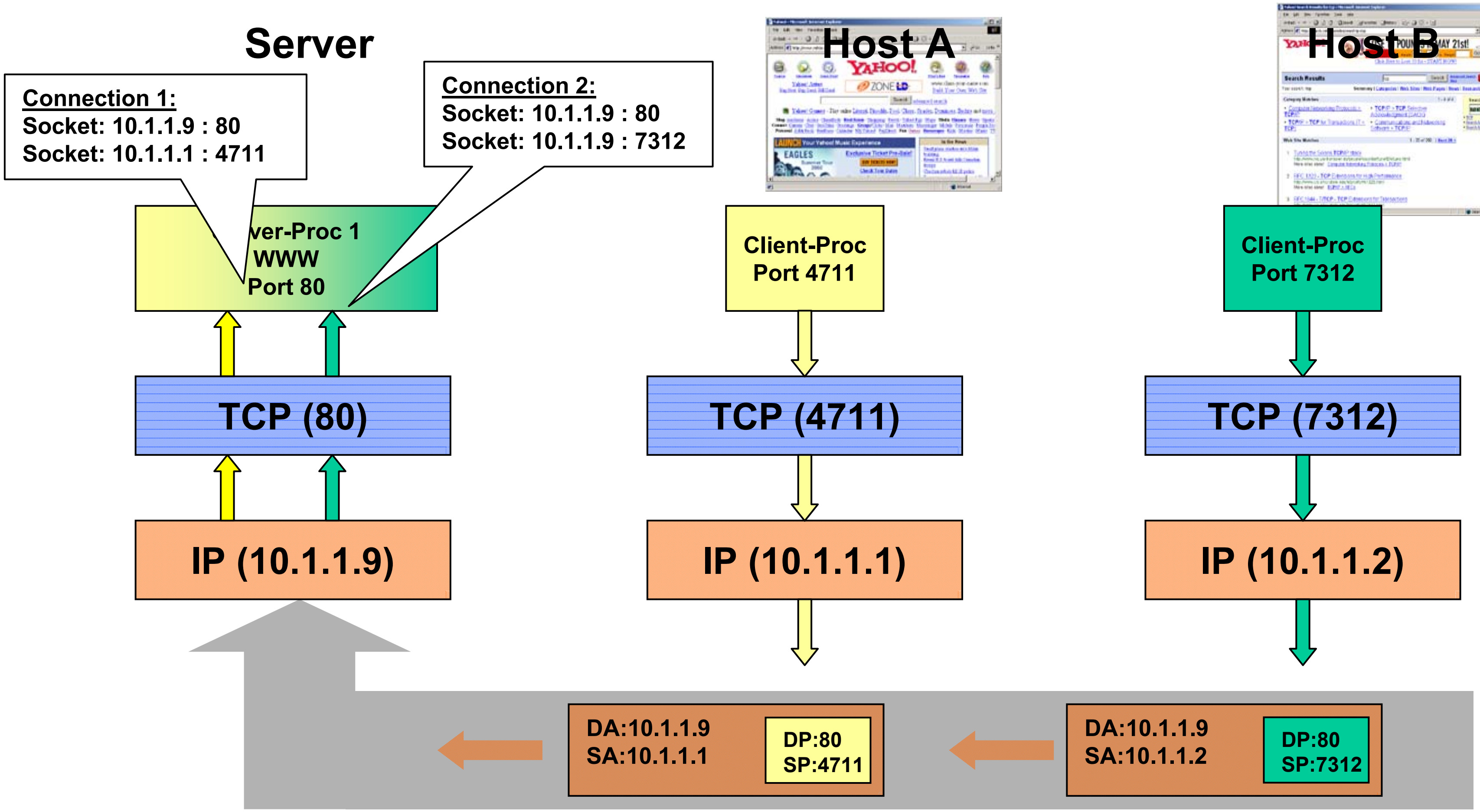
TCP Communications



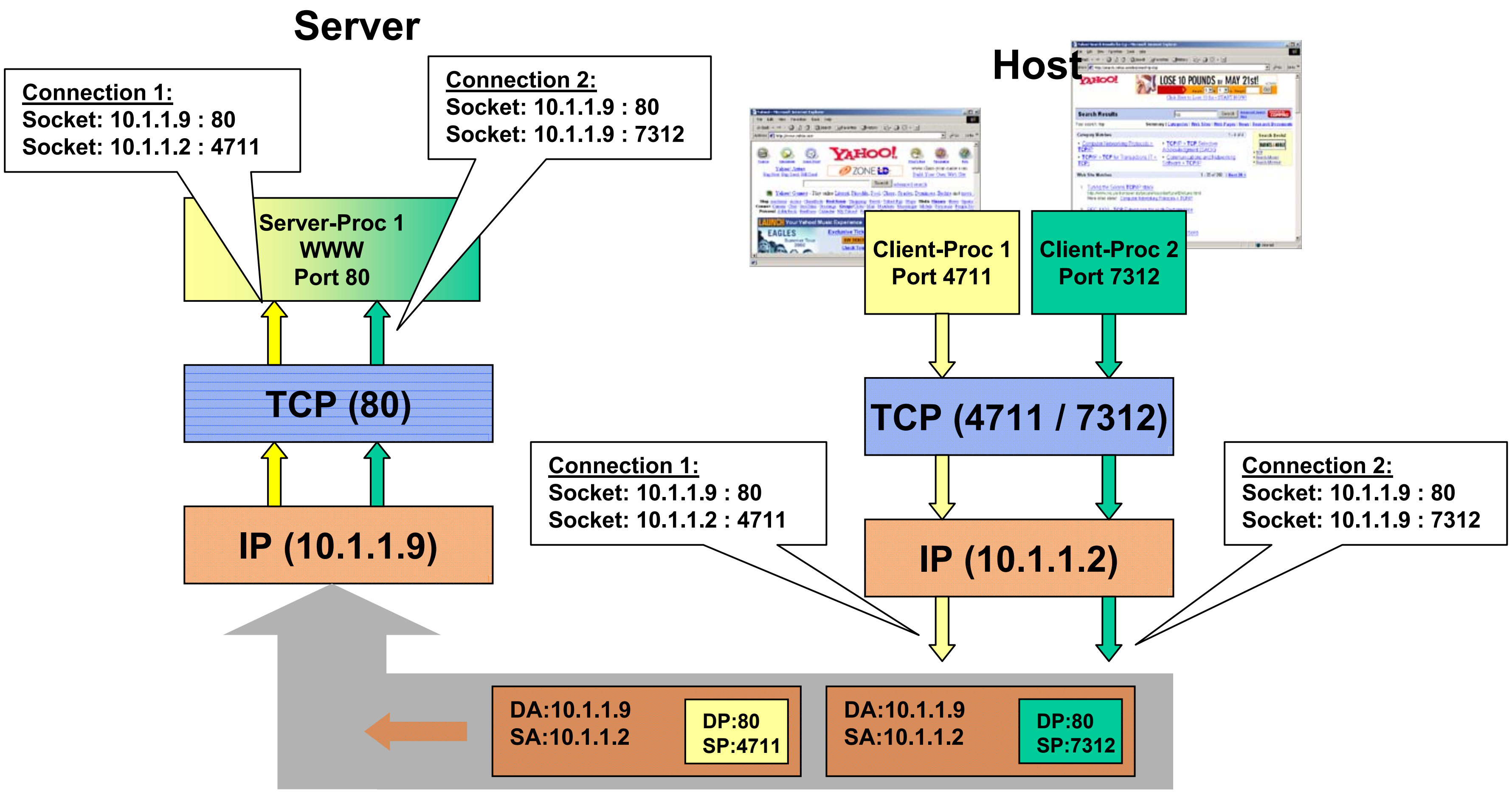
Sockets

- Server process multiplexes streams with same source port numbers according source IP address
- (PortNr, SA) = Socket
- Each stream ("flow") is uniquely identified by a socket pair

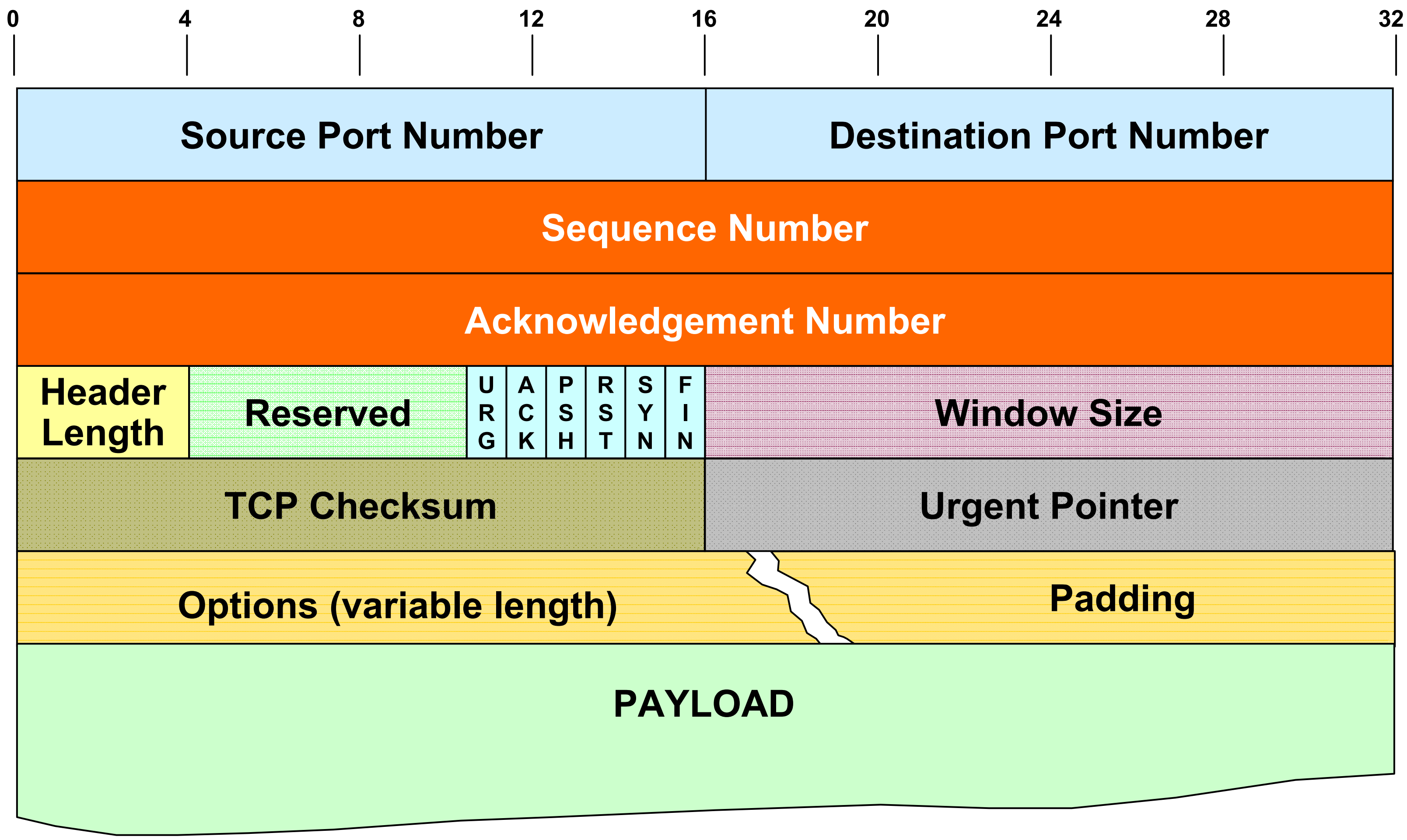
TCP Communications



TCP Communications



TCP Header



0
4
8
12
16
20
24
28
32
Source Port Number
Destination Port Number
Sequence Number
Acknowledgement Number
Header Length
Reserved
URG
ACK
PSH
RST
SYN
FIN
Window Size
TCP Checksum
Urgent Pointer
Options (variable length)
Padding
PAYLOAD

TCP Header (1)

- Source and Destination Port
  - 16 bit port number for source and destination process
- Header Length
  - Multiple of 4 bytes
  - Variable header length because of options (optionally)

TCP Header (2)

- Sequence Number (32 Bit)
  - Number of first byte of this segment
  - Wraps around to 0 when reaching 2^{32} -1)
- Acknowledge Number (32 Bit)
  - Number of next byte expected by receiver
  - Confirms correct reception of all bytes including byte with number AckNr-1

TCP Header (3)

- URG-Flag
  - Indicates urgent data
  - If set, the 16-bit "Urgent Pointer" field is valid and points to the last octet of urgent data
  - There is no way to indicate the beginning of urgent data (!)
  - Applications switch into the "urgent mode"
  - Used for quasi-outband signaling

TCP Header (4)

- **PSH-Flag**
  - TCP should push the segment immediately to the application without buffering
  - To provide low-latency connections
  - Often ignored

TCP Header (5)

- **SYN-Flag**
  - Indicates a connection request
  - Sequence number synchronization
- **ACK-Flag**
  - Acknowledge number is valid
  - Always set, except in very first segment

TCP Header (6)

- **FIN-Flag**
  - Indicates that this segment is the last
  - Other side must also finish the conversation
- **RST-Flag**
  - Immediately kill the conversation
  - Used to refuse a connection-attempt

TCP Header (7)

- **Window (16 Bit)**
  - **Adjusts the send-window size of the other side**
  - **Used with every segment**
  - **Receiver-based flow control**
  - **SeqNr of last octet = AckNr + window**

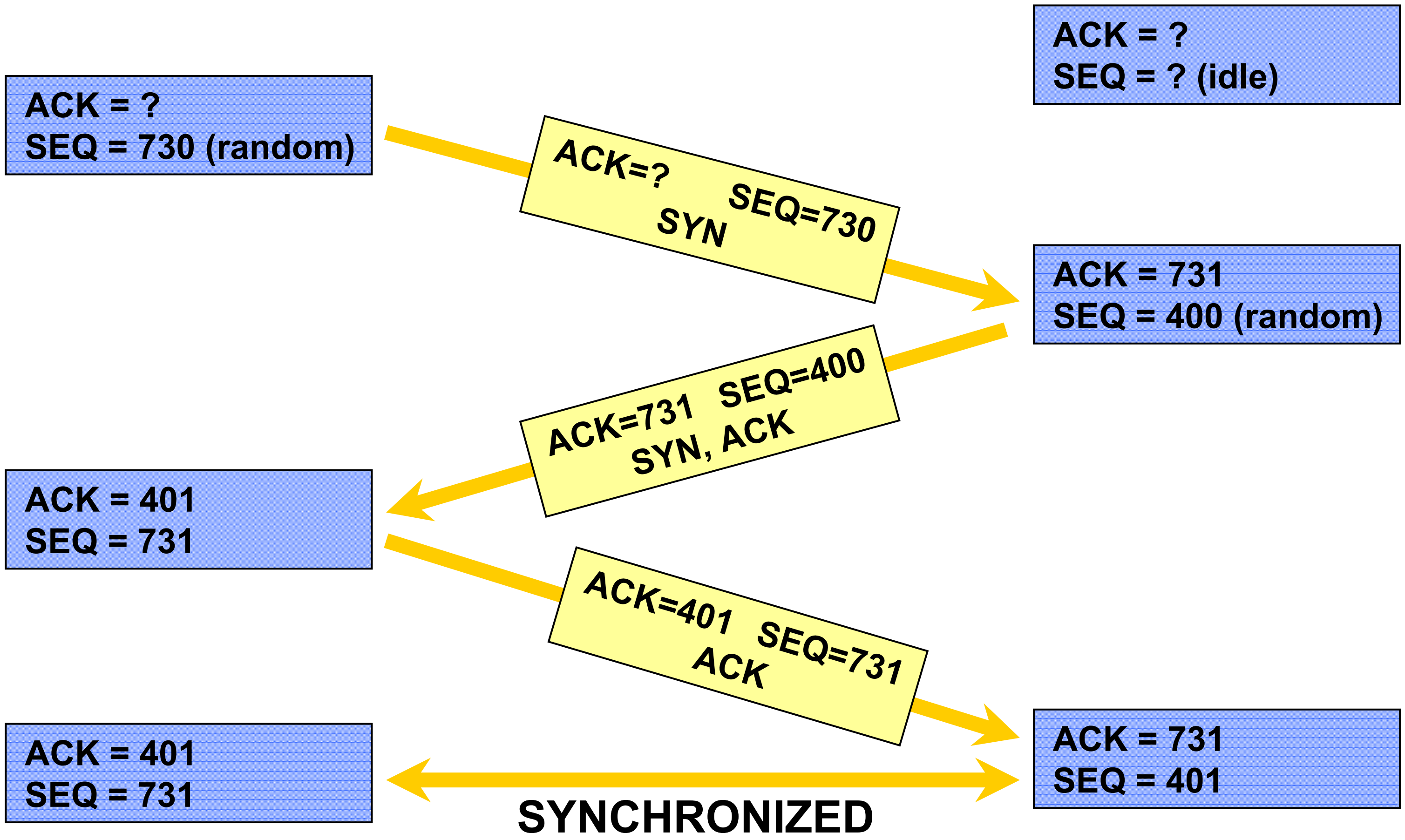
TCP Header (8)

- Checksum
  - Calculated over TCP header, payload and 12 byte pseudo IP header
  - Pseudo IP header consists of source and destination IP address, IP protocol type, and IP total length;
  - Complete socket information is protected
  - Thus TCP can also detect IP errors

TCP Header (9)

- Urgent Pointer
  - Points to the last octet of urgent data
- Options
  - Only MSS (Maximum Message Size) is used
  - Other options are defined in RFC1146, RFC1323 and RFC1693
- Pad
  - Ensures 32 bit alignment

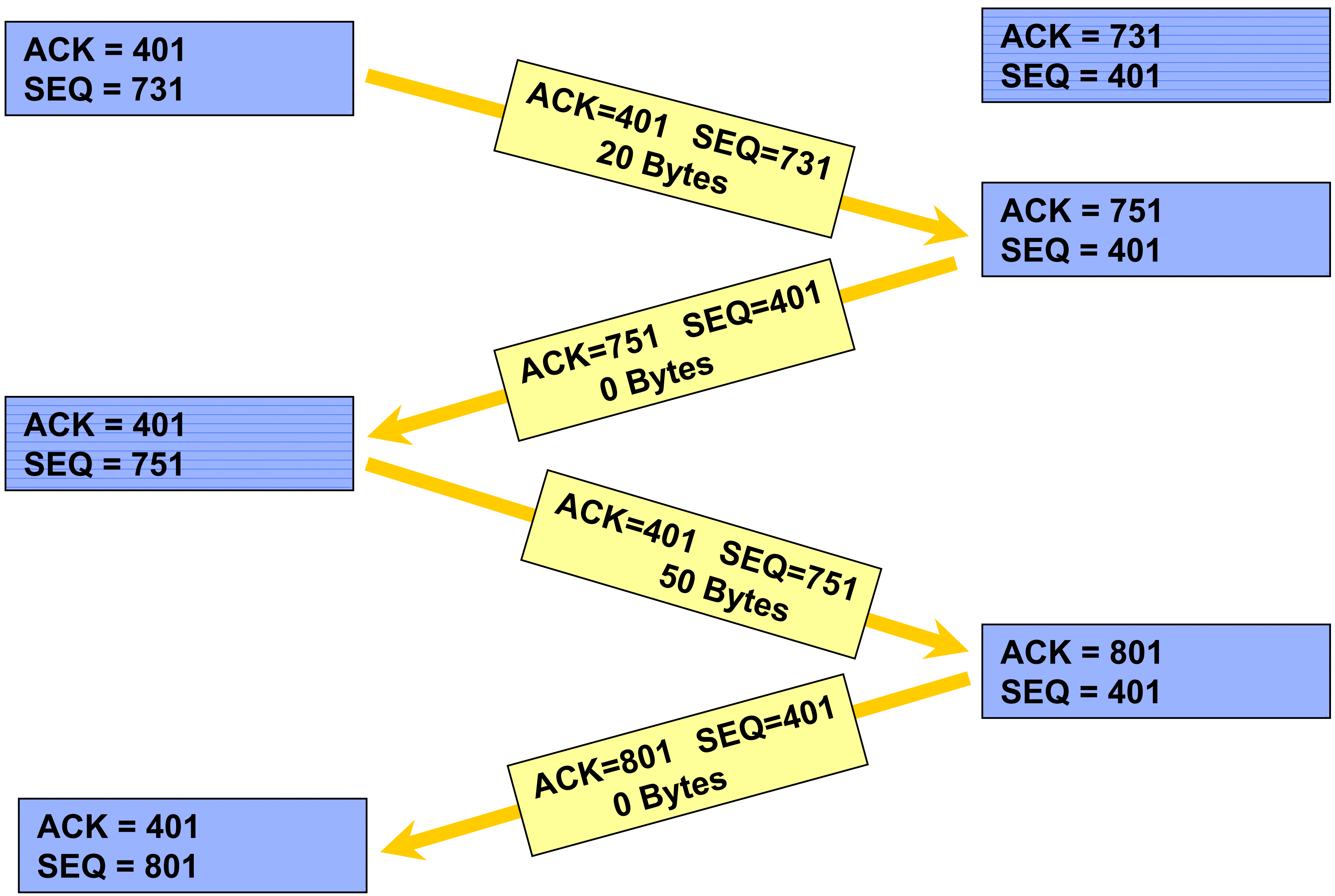
TCP 3-Way-Handshake



Sequence Number

- RFC793 suggests to pick a random number at boot time (e.g. derived from system start up time) and increment every 4 µs
- Every new connection will increments SeqNr by 1
- To avoid interference of spurious packets
- Old "half-open" connections are deleted with the RST flag

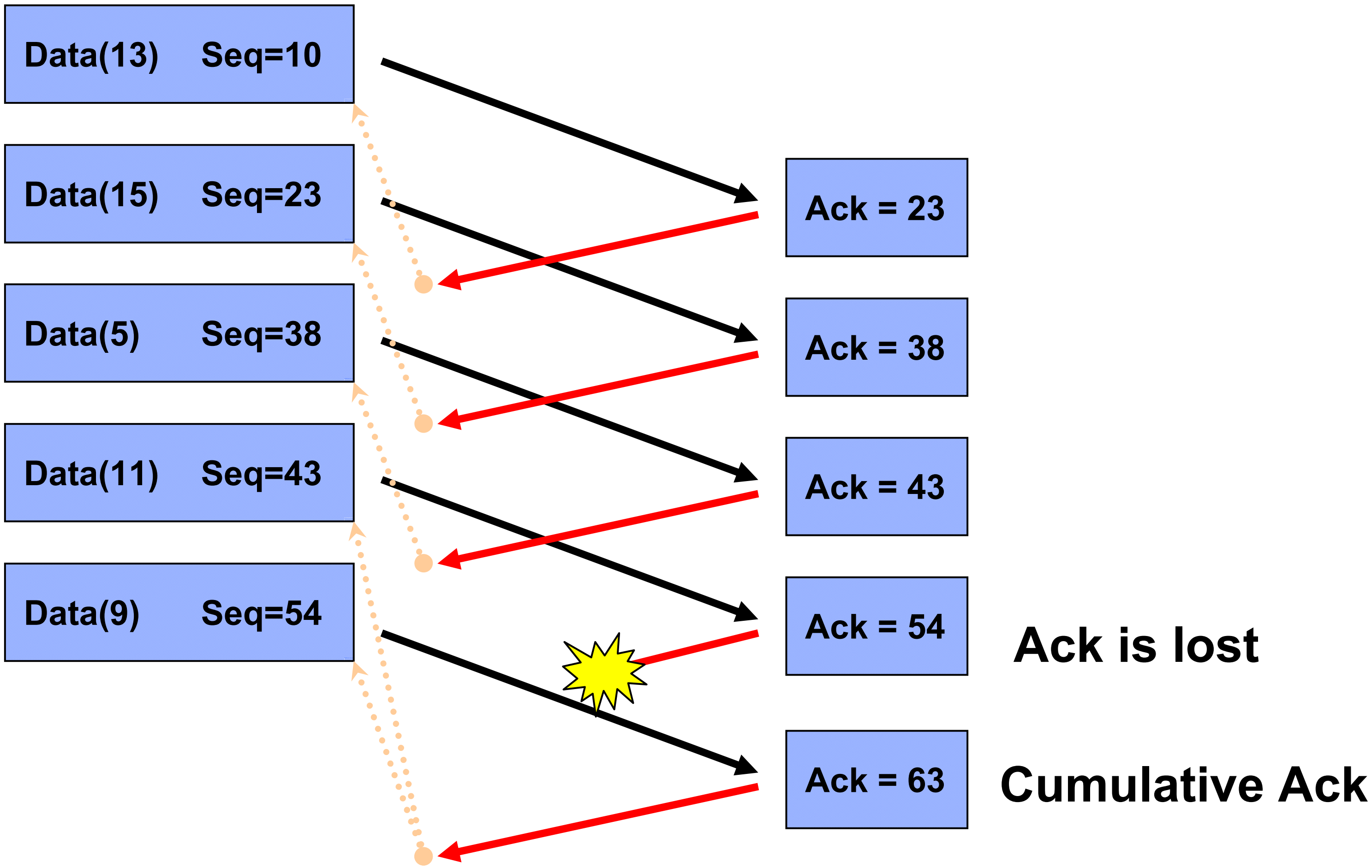
TCP Data Transfer



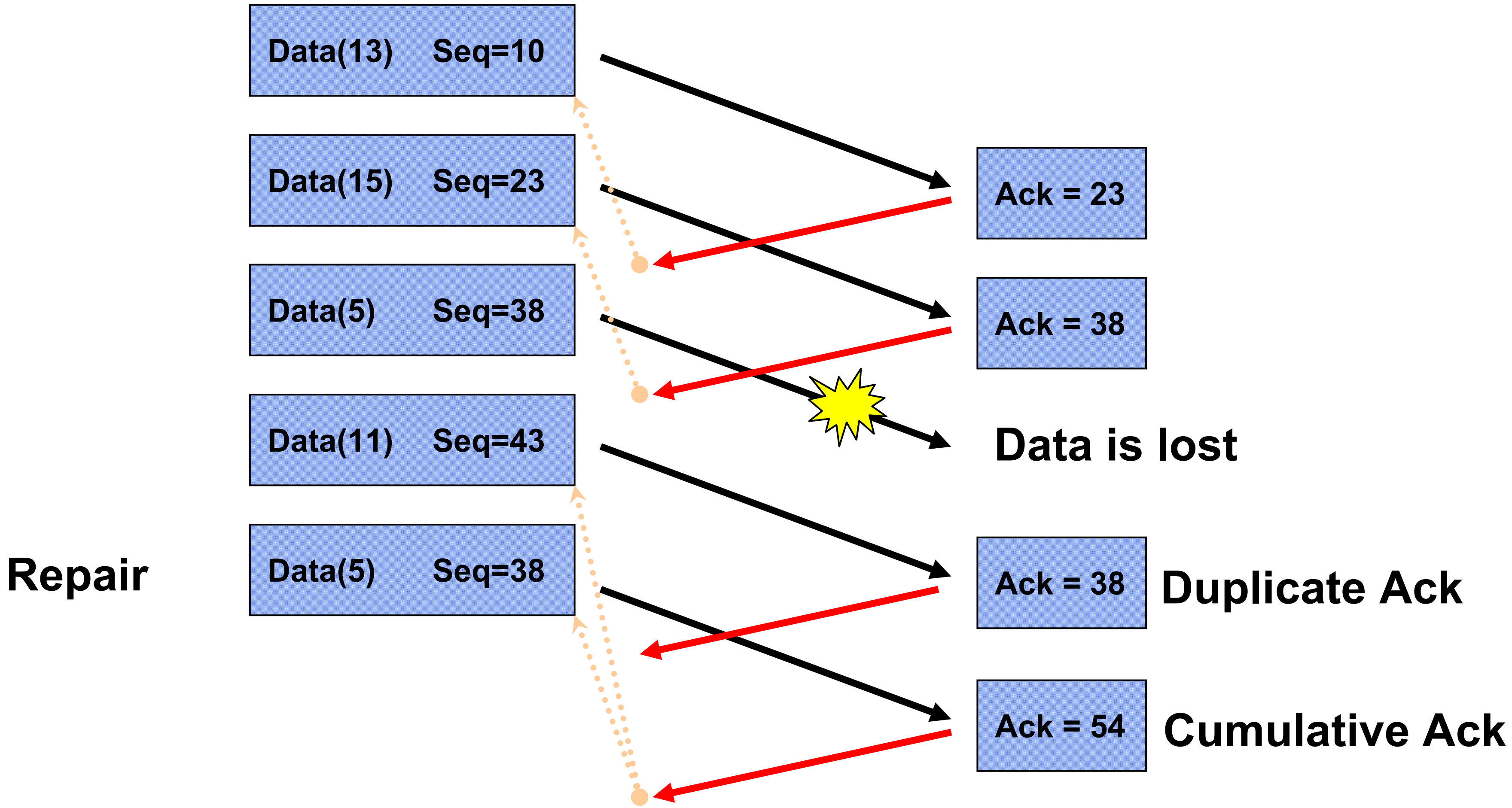
TCP Data Transfer

- Acknowledgements are generated for all octets which arrived in sequence without errors (positive acknowledgement)
- Duplicates are also acknowledged (!)
  - Receiver cannot know why duplicate has been sent; maybe because of a lost acknowledgement
- The acknowledge number indicates the sequence number of the next byte to be received
- Acknowledgements are cumulative: Ack(N) confirms all bytes with sequence numbers up to N-1
  - Therefore lost acknowledgements are no problem

Cumulative Acknowledgement



Duplicate Acknowledgement



TCP Retransmission Timeout

- Retransmission timeout (RTO) will initiate a retransmission of unacknowledged data
  - High timeout results in long idle times if an error occurs
  - Low timeout results in unnecessary retransmissions
- TCP continuously measures RTT to adapt RTO

Retransmission ambiguity problem

- If a packet has been retransmitted and an ACK follows: Does this ACK belong to the retransmission or to the original packet?
  - Could distort RTT measurement dramatically
- Solution: Phil Karn's algorithm
  - Ignore ACKs of a retransmission for the RTT measurement
  - And use an exponential backoff method

RTT Estimation (1/2)

- For TCP's performance a precise estimation of the current RTT is crucial
  - RTT may change because of varying network conditions (e. g. re-routing)
- Originally a smooth RTT estimator was used (a low pass filter)
  - M denotes the observed RTT (which is typically inprecise because there is no one-to-one mapping between data and ACKs)
  - $R = \alpha R + (1 - \alpha) M$ with smoothing factor $\alpha = 0.9$
  - Finally $RTO = \beta \cdot R$ with variance factor $\beta = 2$

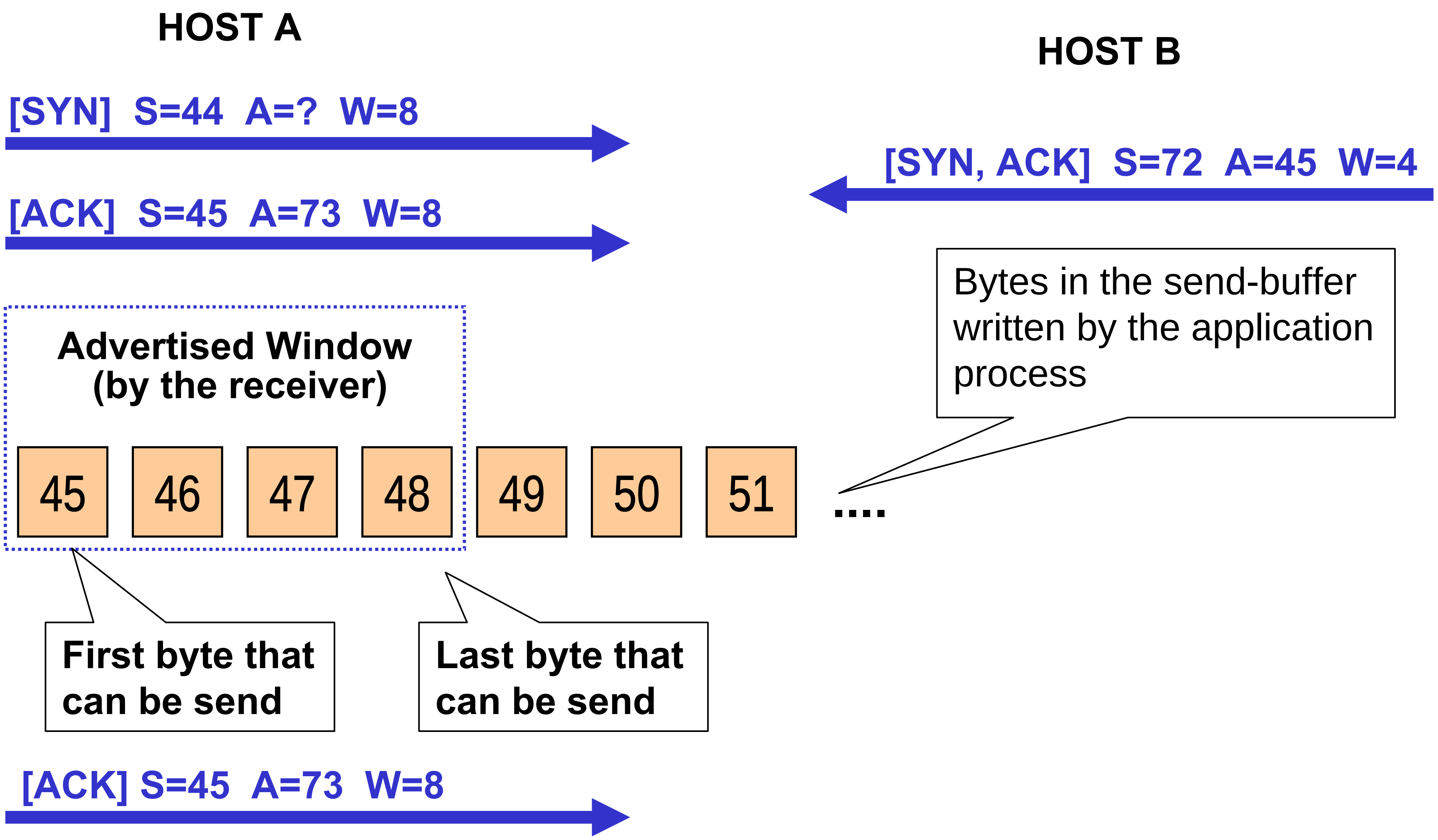
RTT Estimation (2/2)

- **Initial smooth RTT estimator could not keep up with wide fluctuations of the RTT**
  - **Led to too many retransmissions**
- **Jacobson's suggested to take the RTT variance also into account**
  - **Err = M − A**
    - **The deviation from the measured RTT (M) and the RTT estimation (A)**
  - **A = A + g · Err**
    - **with gain g = 0.125**
  - **D = D + h (|Err| − D)**
    - **with h = 0.25**
  - **RTO = A + 4D**

TCP Sliding Window

- TCP flow control is done with dynamic windowing using the sliding window protocol
- The receiver advertises the current amount of octets it is able to receive
  - Using the window field of the TCP header
  - Values 0 through 65535
- Sequence number of the last octet a sender may send = received ack-number -1 + window size
  - The starting size of the window is negotiated during the connect phase
  - The receiving process can influence the advertised window, hereby affecting the TCP performance

TCP Sliding Window

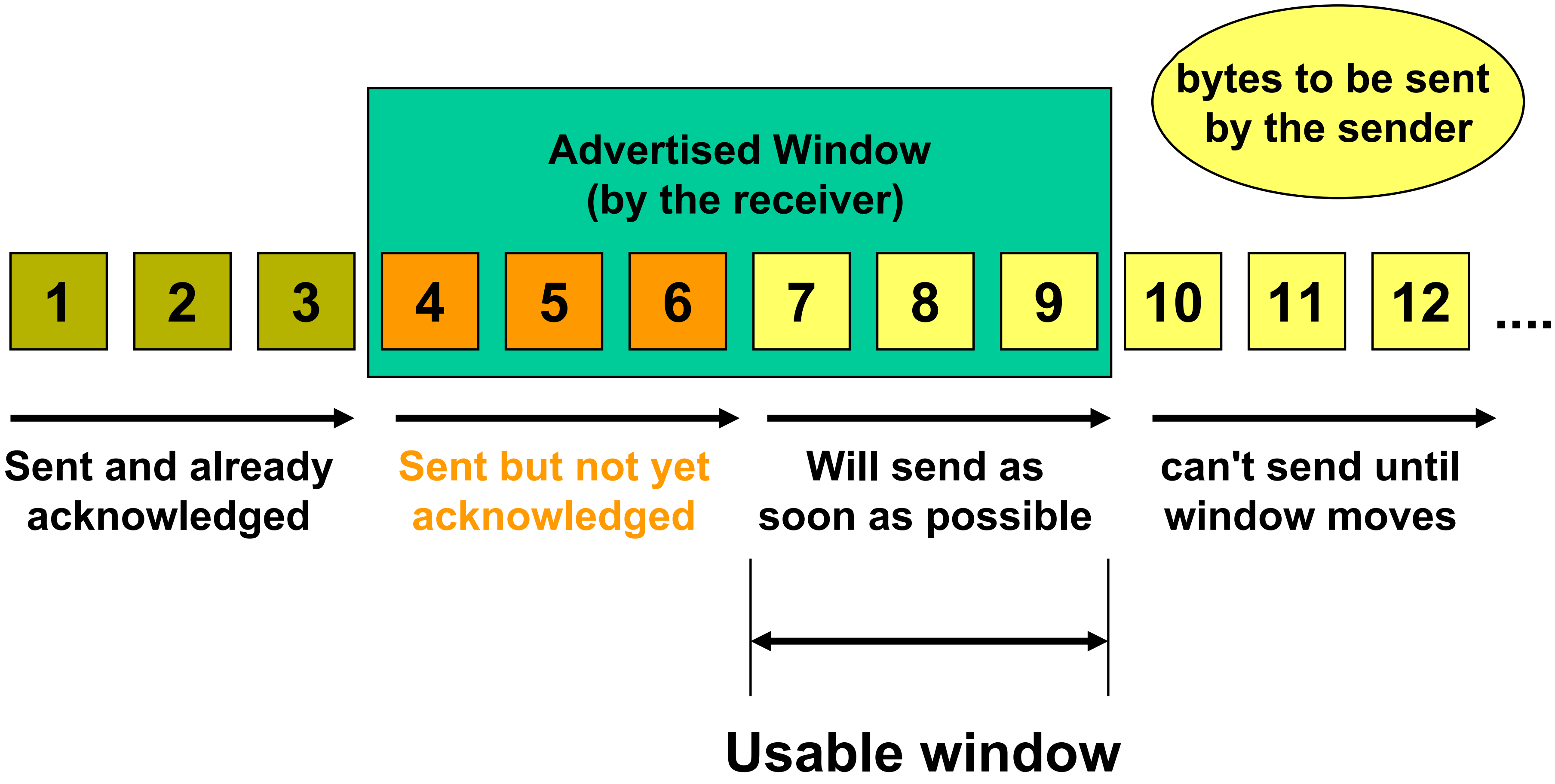


TCP Sliding Window

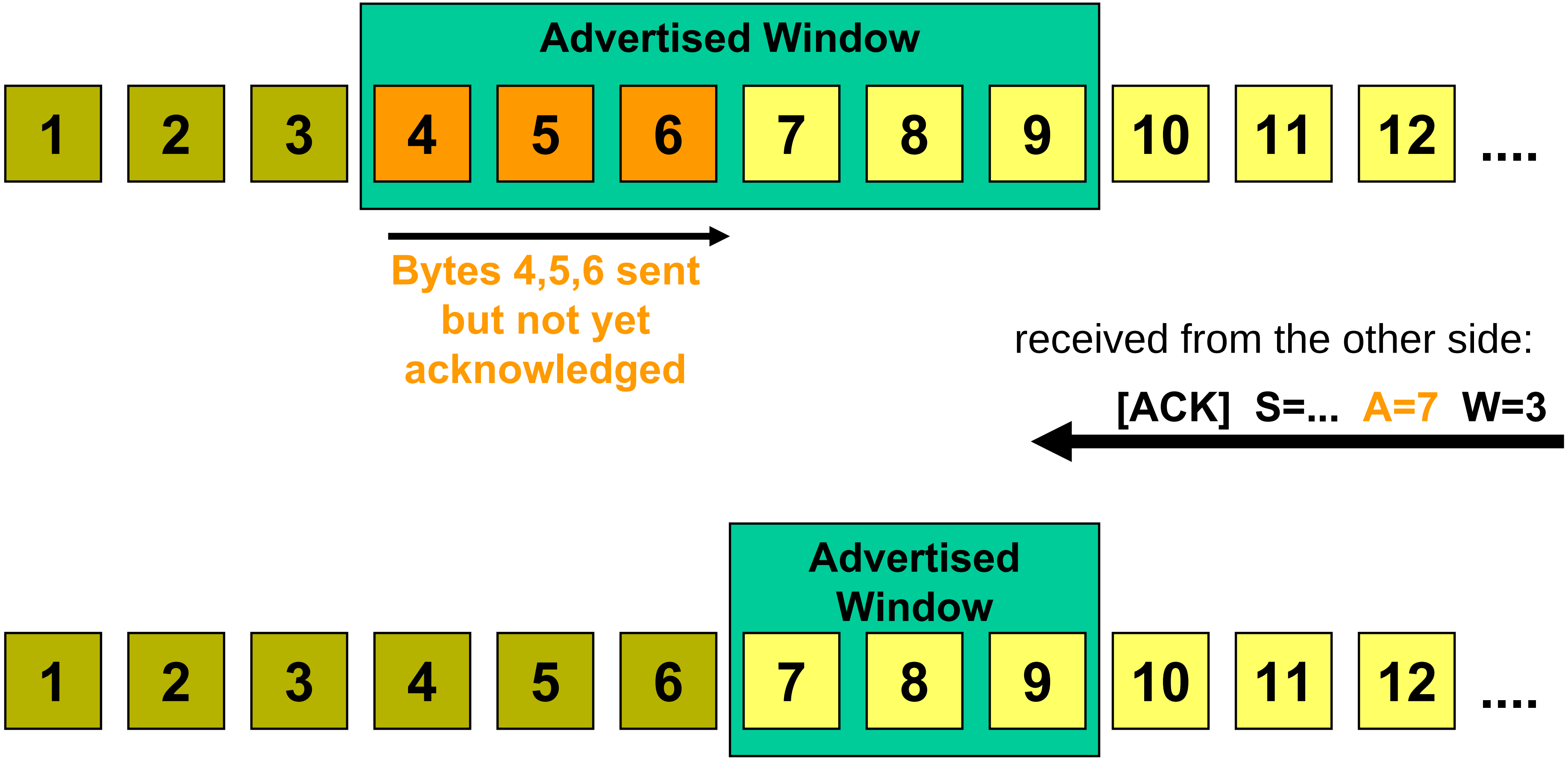
- During the transmission the sliding window moves from left to right, as the receiver acknowledges data
- The relative motion of the two ends of the window open or closes the window
  - The window closes when data is sent and acknowledged (the left edge advances to the right)
  - The window opens when the receiving process on the other end reads acknowledges data and frees up TCP buffer space (the right edge moves to the right)
- If the left edge reaches the right edge, the sender stops transmitting data - zero window

Sliding Window: Principle

Sender's point of view; sender got {ACK=4, WIN=6} from the receiver.

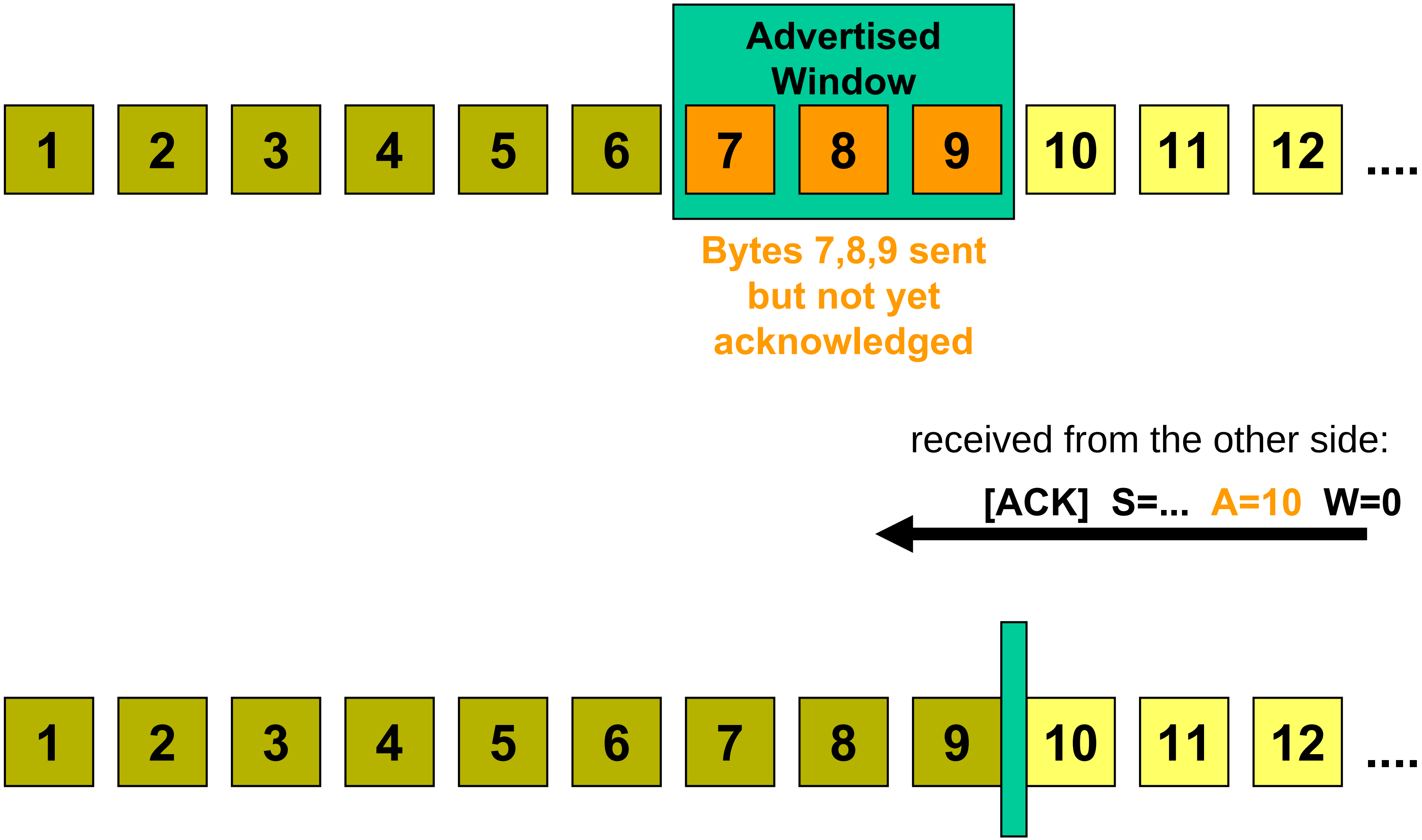


Closing the Sliding Window

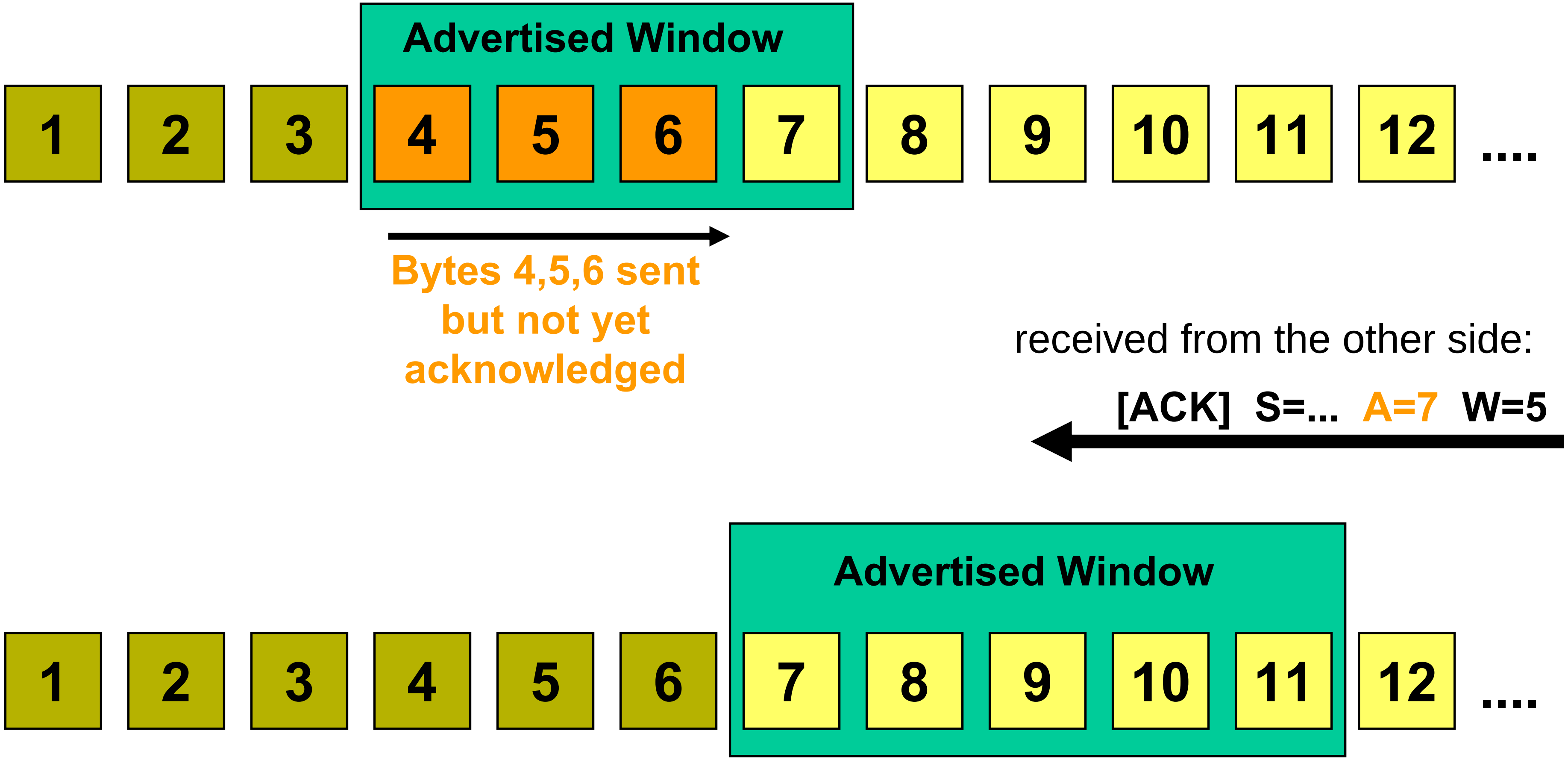


Now the sender may send bytes 7, 8, 9. The receiver didn't open the window (W=3, right edge remains constant) because of congestion. However, the remaining three bytes inside the window are already granted, so the receiver cannot move the right edge leftwards.

Flow Control -> STOP, Window Closed



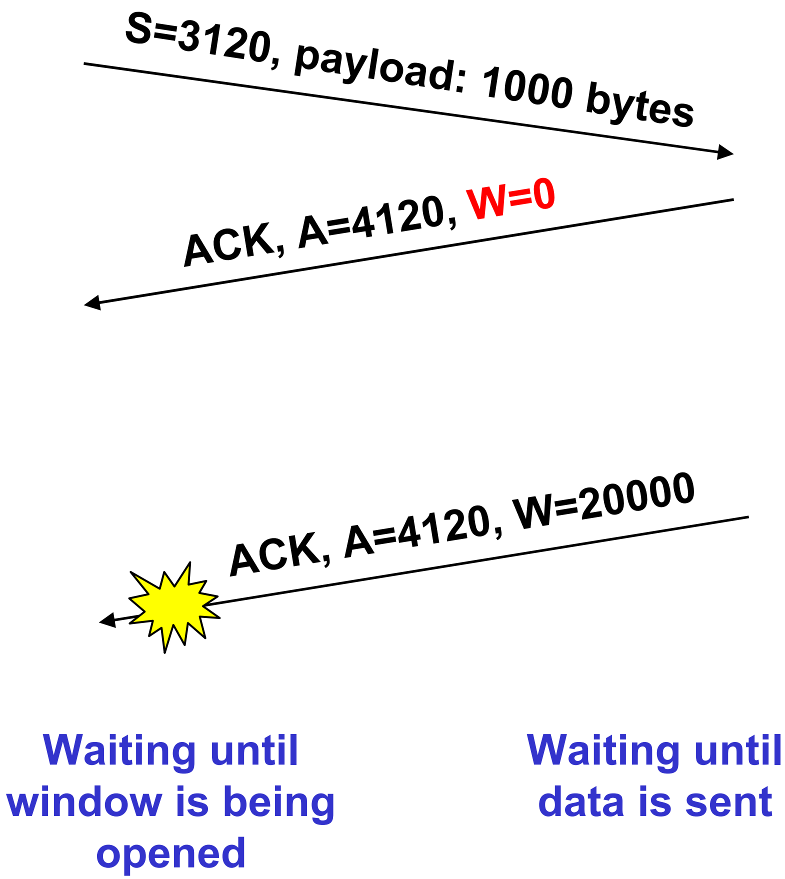
Opening the Sliding Window



The receiver's application read data from the receive-buffer and acknowledged bytes 4,5,6. Free space of the receiver's buffer is indicated by a window value that makes the right edge of the window move rightwards. Now the sender may send bytes 7, 8, 9,10,11.

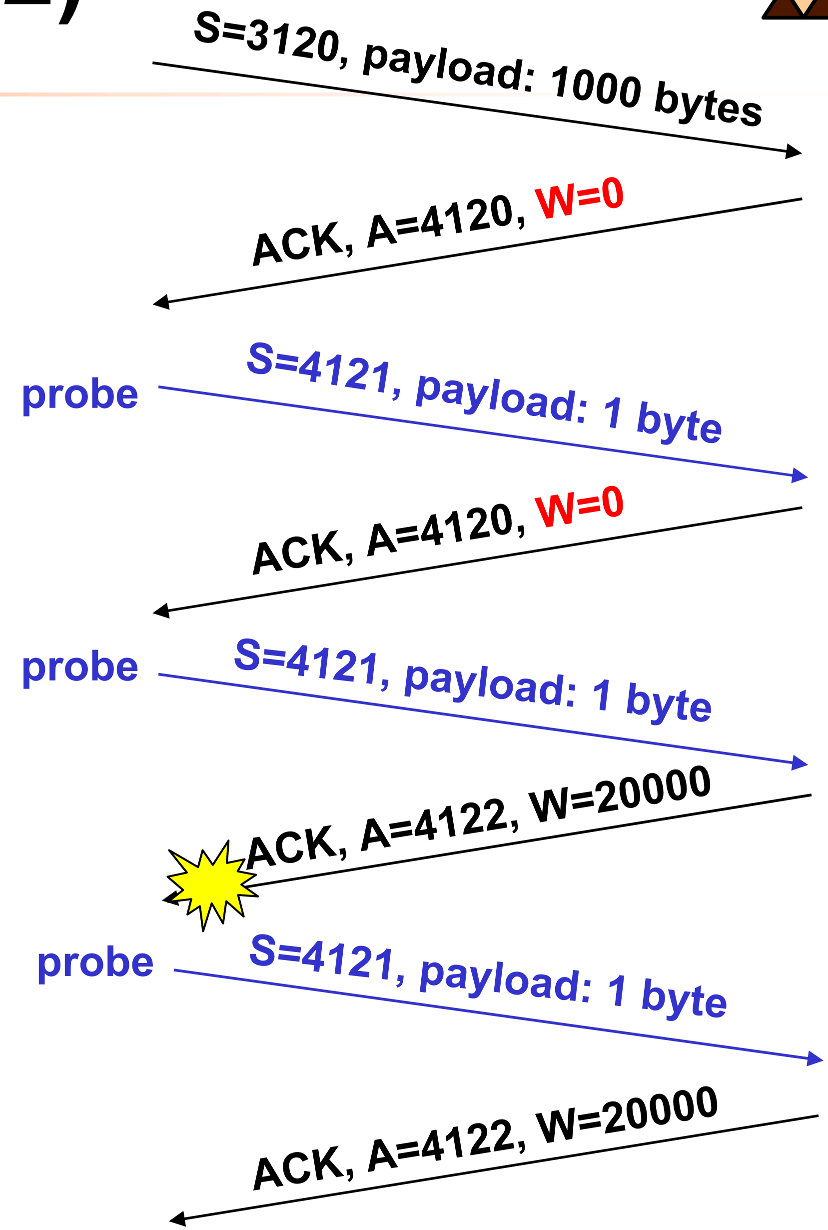
TCP Persist Timer (1/2)

- Deadlock possible: Window is zero and window-opening ACK is lost!
  - ACKs are sent unreliable!
  - Now both sides wait for each other!



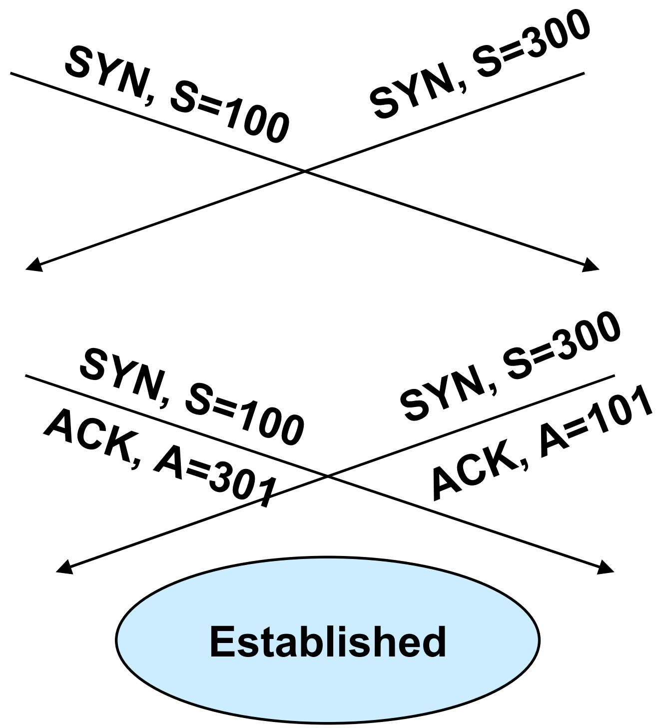
TCP Persist Timer (2/2)

- **Solution: Sender may send *window probes:***
  - **Send one data byte *beyond* window**
  - **If window remains closed then this byte is not acknowledged—so this byte keeps being retransmitted**
- **TCP sender remains in persist state and continues retransmission forever (until window size opens)**
  - **Probe intervals are increased exponentially between 5 and 60 seconds**
  - **Max interval is 60 seconds (forever)**



Simultaneous Open

- If an application uses well known ports for both client and server, a "simultaneous open" can be done
  - TCP explicitly supports this
  - A single connection (not two!) is the result
- Since both peers learn each others sequence number at the very beginning the session is established with a following SYN-ACK
- Hard to realize in practice
  - Both SYN packets must cross each other in the network
  - Rare situation!



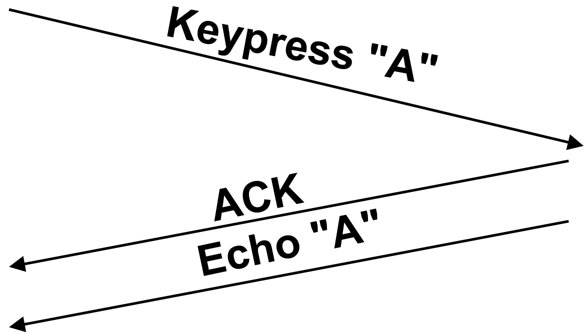
TCP Enhancements

- So far, only the very basic TCP procedures have been mentioned
- But TCP has much more magic built-in algorithms which are essential for operation in today's IP networks:
  - "Slow Start" and “Congestion Avoidance”
  - "Fast Retransmit" and "Fast Recovery"
  - "Delayed Acknowledgements"
  - "The Nagle Algorithm“
  - Selective Ack (SACK), Window Scaling
  - Silly windowing avoidance
  - ....
- Additionally, there are different implementations (Reno, Vegas, …)

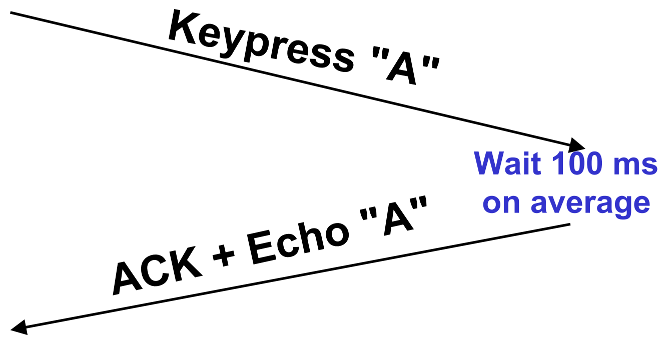
Delayed ACKs

- Goal: Reduce traffic, support piggy-backed ACKs
- Normally TCP, after receiving data, does not immediately send an ACK
- Typically TCP waits (typically) 200 ms and hopes that layer-7 provides data that can be sent along with the ACK

Example:
Telnet and no Delayed ACK



Example:
Telnet with Delayed ACK



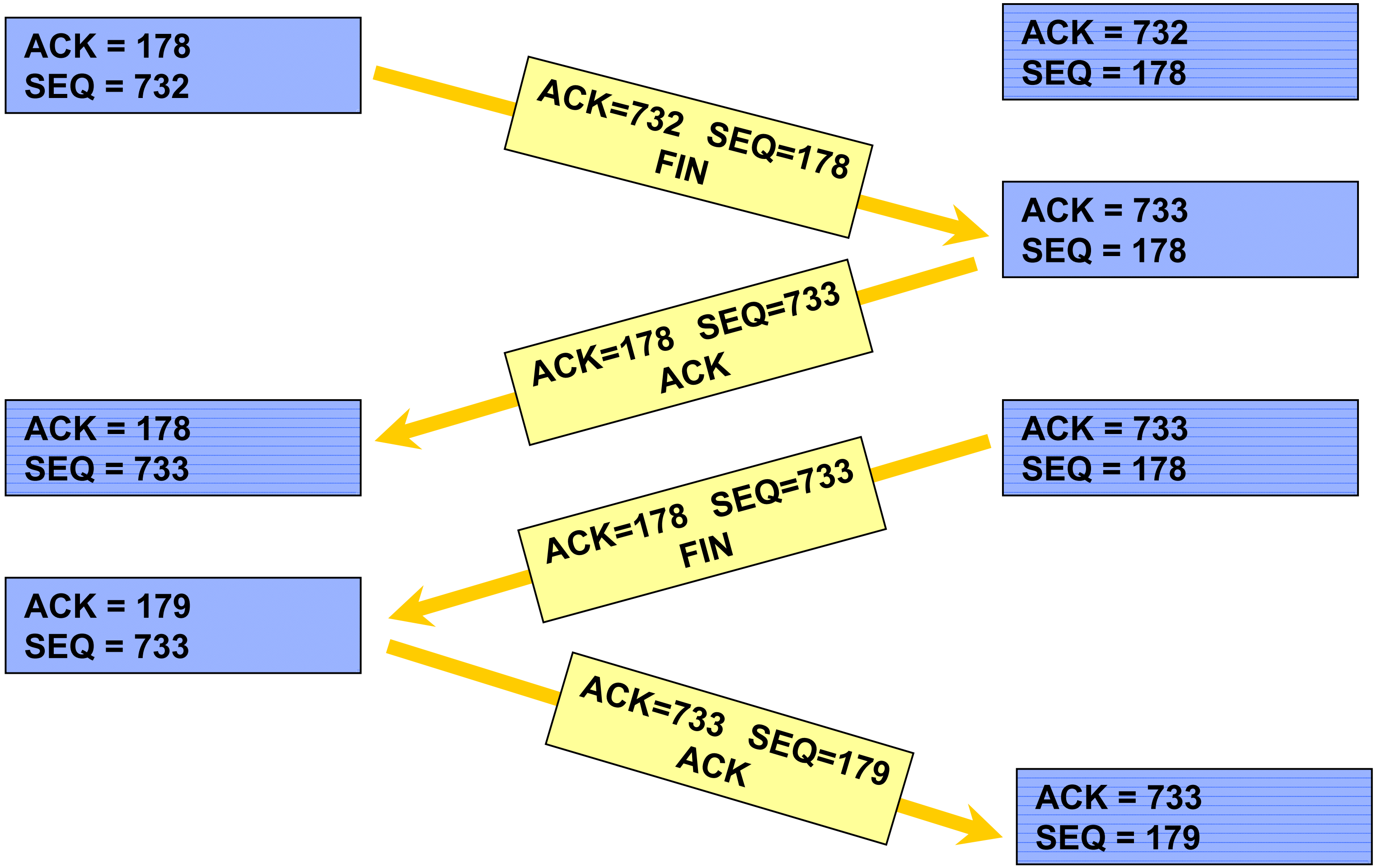
Nagle Algorithm

- Goal: Avoid *tinygrams* on expensive (and usually slow) WAN links
- In RFC 896 John Nagle introduced an efficient algorithm to improve TCP
- Idea: In case of outstanding (=unacknowledged) data, small segments should not be sent until the outstanding data is acknowledged
- In the meanwhile small amount of data (arriving from Layer 7) is collected and sent as a single segment when the acknowledgement arrives
- This simple algorithm is self-clocking
  - The faster the ACKs come back, the faster data is sent
- Note: The Nagle algorithm can be disabled!
  - Important for realtime services

TCP Keepalive Timer

- **Note that absolutely no data flows during an idle TCP connection!**
  - **Even for hours, days, weeks!**
- **Usually needed by a server that wants to know which clients are still alive**
  - **To close stale TCP sessions**
- **Many implementations provide an optional TCP keepalive mechanism**
  - **Not part of the TCP standard!**
  - **Not recommended by RFC 1122 (hosts requirements)**
  - **Minimum interval must be 2 hours**

TCP Disconnect



TCP Disconnect

- **A TCP session is disconnected similar to the three way handshake**
- **The FIN flag marks the sequence number to be the last one; the other station acknowledges and terminates the connection in this direction**
- **The exchange of FIN and ACK flags ensures, that both parties have received all octets**
- **The RST flag can be used if an error occurs during the disconnect phase**

TCP Congestion Control

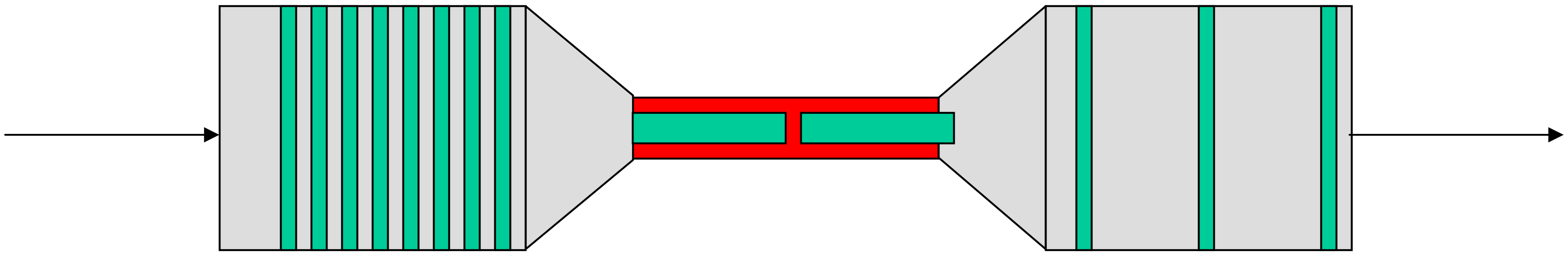
1. Slow Start & Congestion Avoidance
2. Random Early Discard
3. Explicit Congestion Notification

Once again: The Window Size

- The windows size (announced by the peer) indicates ***how many bytes I may send at once*** (=without having to wait for acknowledgements)
  - Either using big or small packets
- Before 1988, TCP peers tend to exploit the whole window size which has been announced during the 3-way handshake
  - Usually no problem for hosts
  - But led to frequent network congestions

Goal of Slow Start

- TCP should be "ACK-clocking"
  - Problem (buffer overflows) appears at bottleneck links
  - New packets should be injected at the rate at which ACKs are received



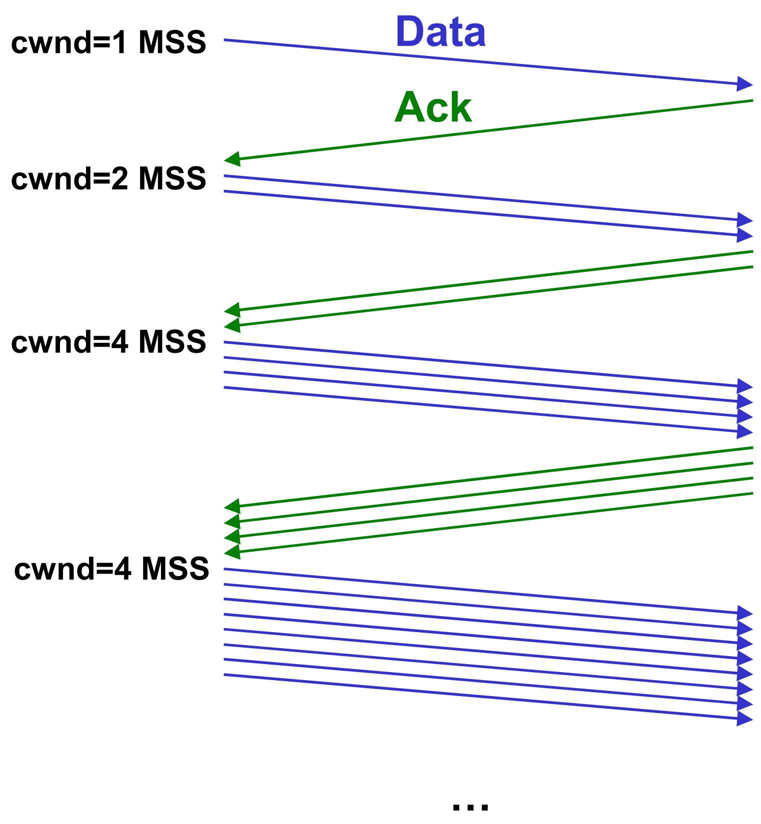
Pipe modell of a network path: Big fat pipes (high data rates) outside, a bottleneck link in the middle. The green packets are sent at the maximum achievable rate so that the interpacket delay is almost zero at the bottleneck link; however there is a significant interpacket gap in the fat pipes.

Preconditions of Slow Start

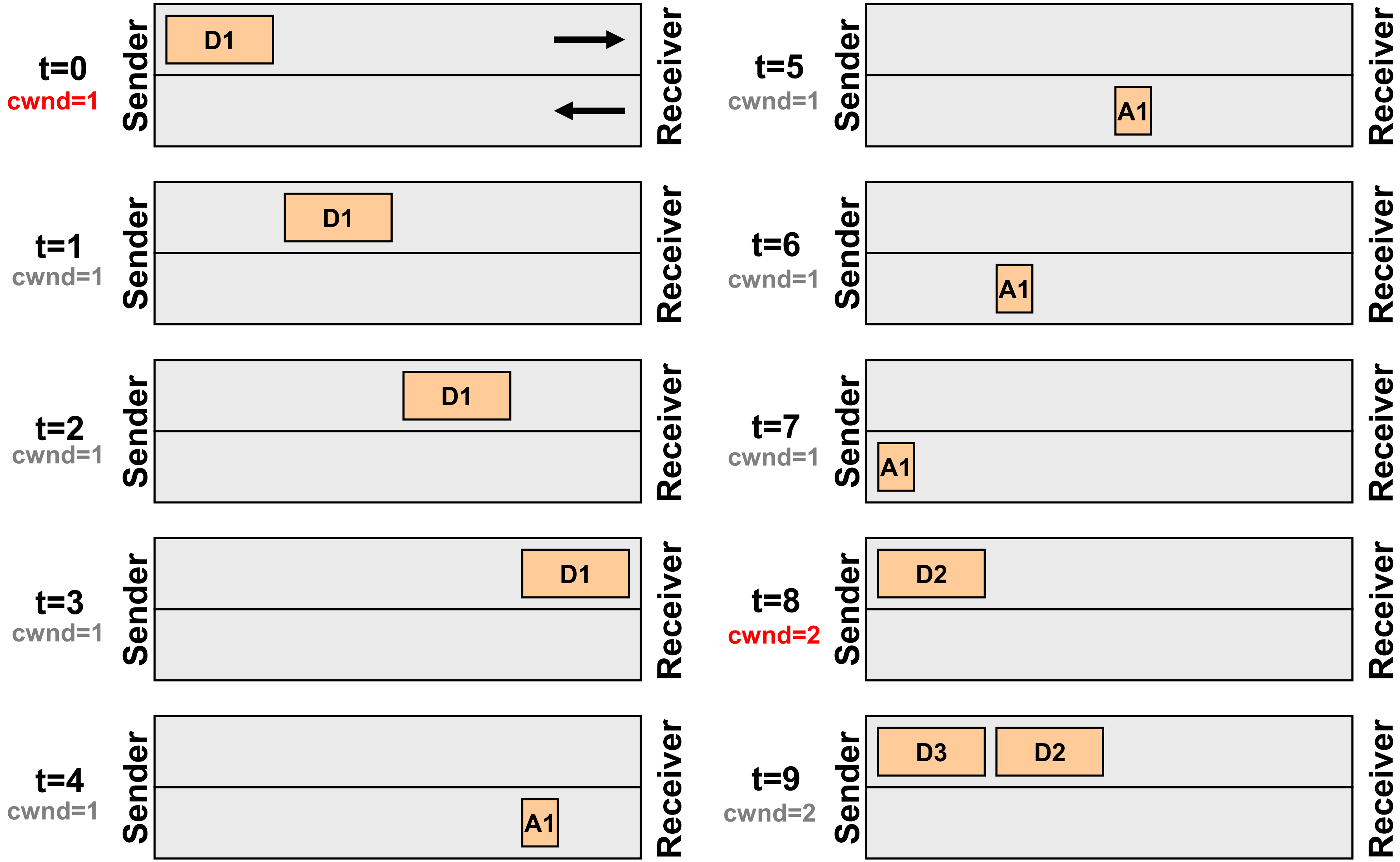
- Two important parameters are communicated during the TCP three-way handshake
  - The maximum segment size (MSS)
  - The Window Size
- Now Slow Start introduces the *congestion window (cwnd)*
  - Only locally valid and locally maintained
  - Like window field stores a byte count

Idea of Slow Start

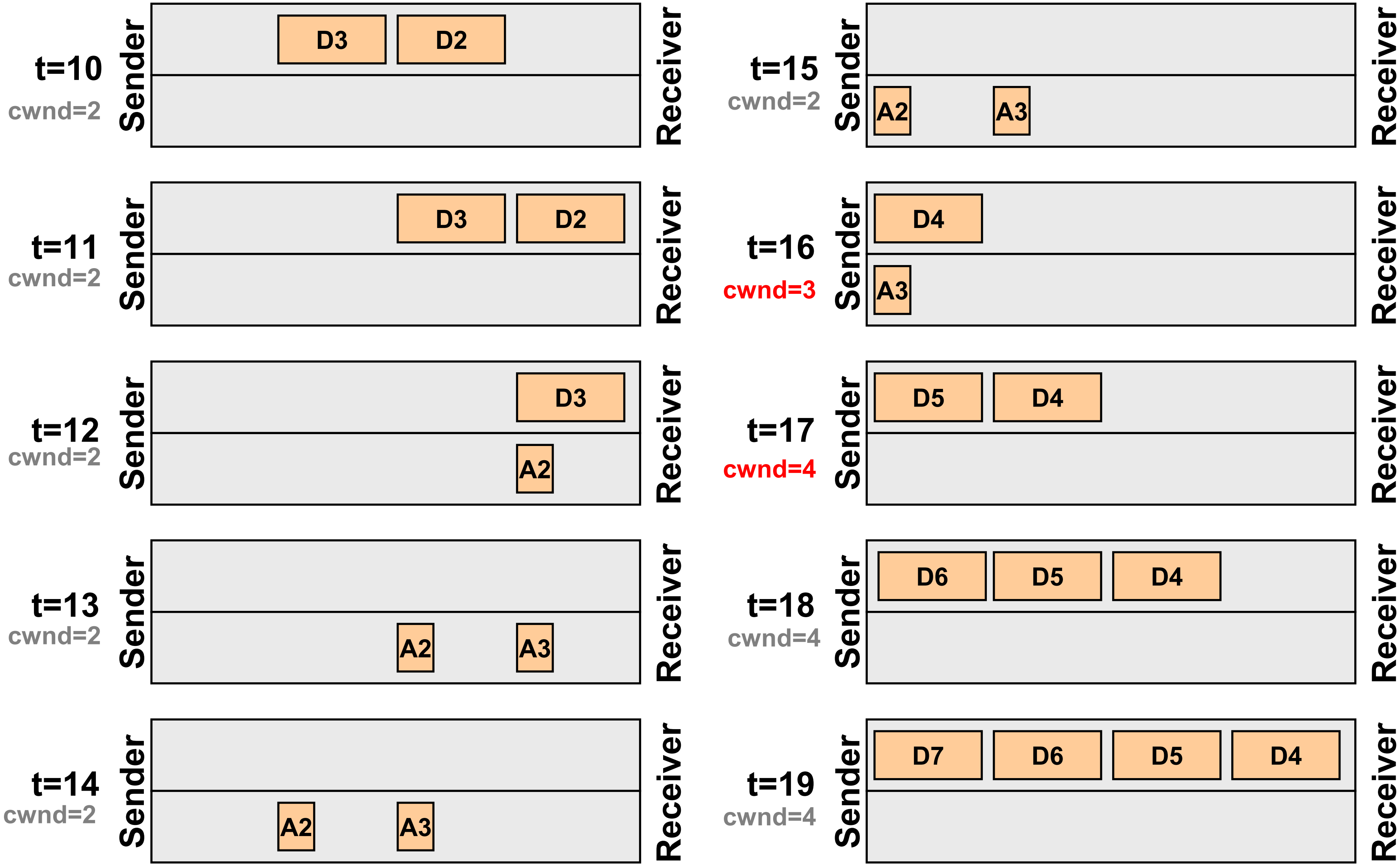
- Upon new session, cwnd is initialized with MSS (= 1 segment)
- Allowed bytes to be sent: Min(W, cwnd)
- Each time an ACK is received, cwnd is incremented by 1 segment
  - That is, cwnd doubles every RTT (!)
  - Exponential increase!



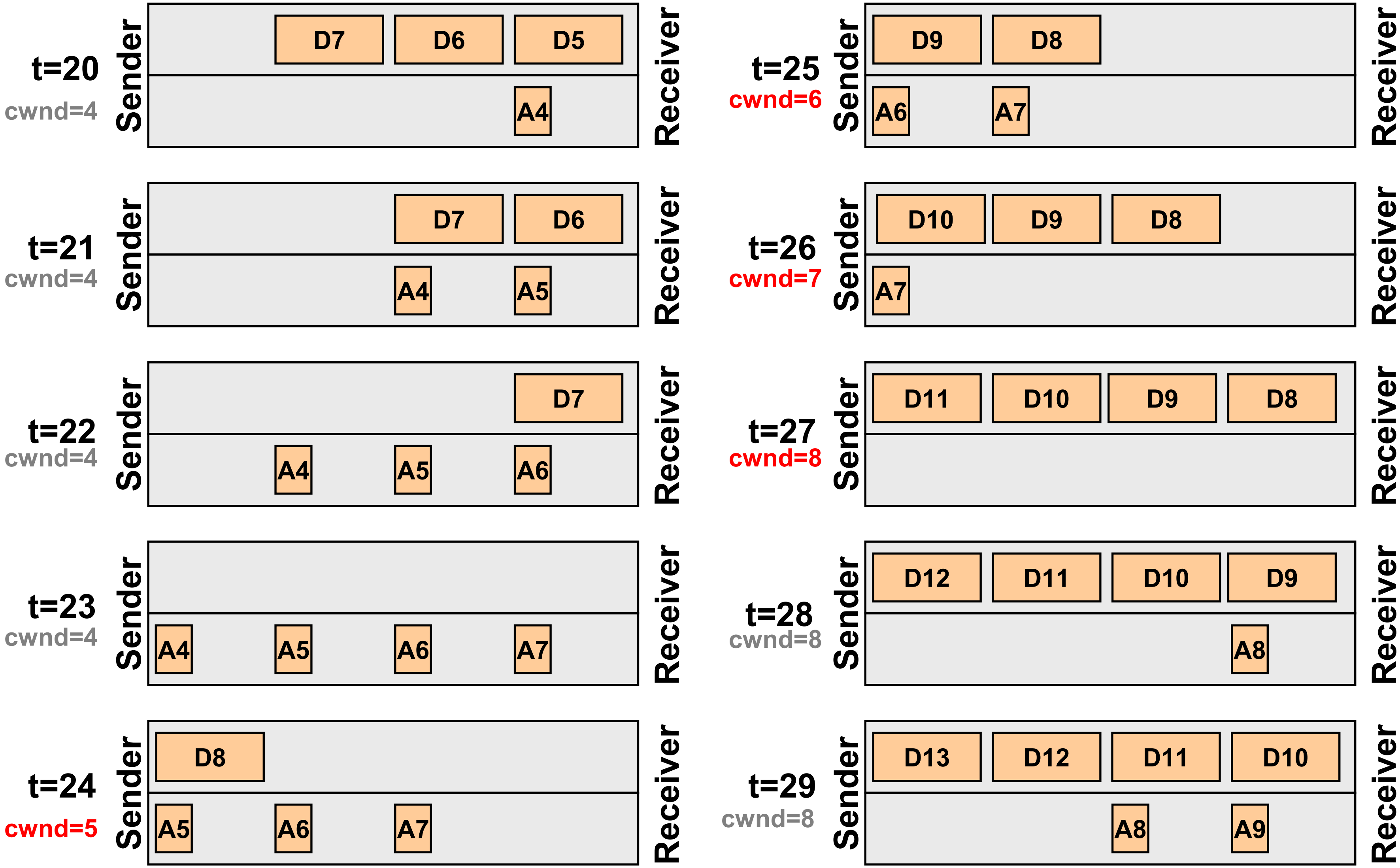
Graphical illustration (1/4)



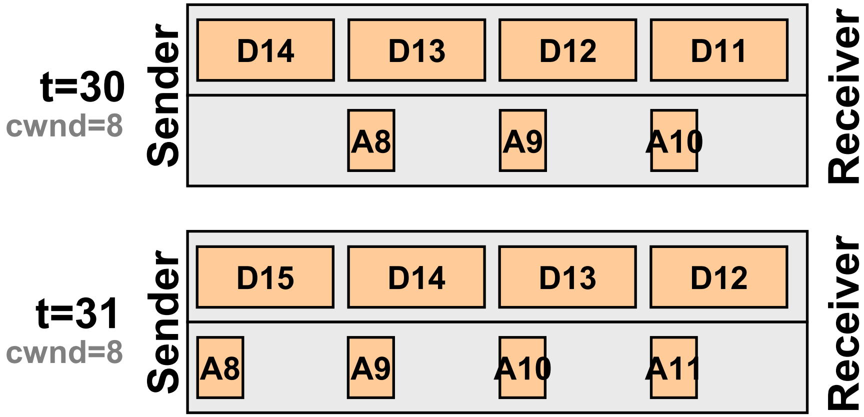
Graphical illustration (2/4)



Graphical illustration (3/4)



Graphical illustration (4/4)



cwnd=8 => Pipe is full (ideal situation) – cwnd should not be increased anymore!

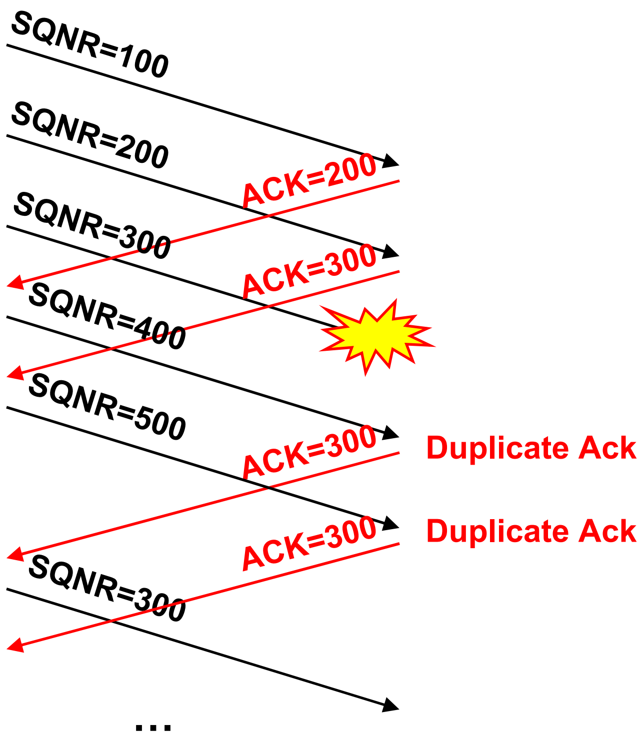
- TCP is *"self-clocking"*
  - The spacing between the ACKs is the same as between the data segments
  - The number of ACKs is the same as the number of data segments
- In our example, cwnd=8 is the optimum
  - This is the delay-bandwidth product (8 = RTT x BW)
  - In other words: the pipe can accept 8 packets per round-trip-time

End of Slow Start

- **Slow start leads to an exponential increase of the data rate until some network bottleneck is congested: Some packets get dropped!**
- ***How does the TCP sender recognize network congestions?***
- **Answer: *Upon receiving Duplicate Acknowledgements !!!***

Once again: Duplicate ACKs

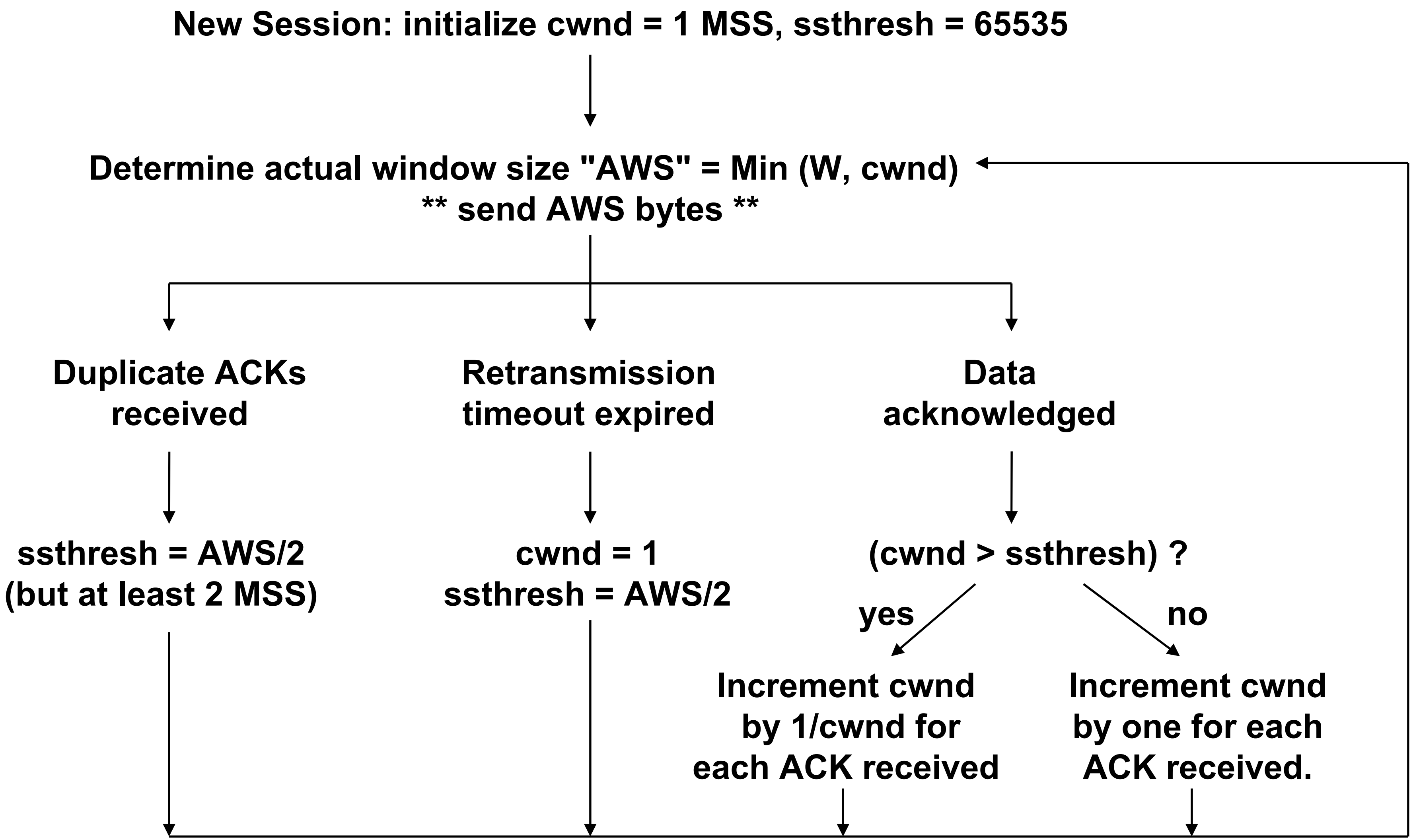
- **TCP receivers send duplicate ACKs if segments are missing**
  - **ACKs are cumulative (each ACK acknowledges all data until specified ACK-number)**
  - **Duplicate ACKs should not be delayed**
- **ACK=300 means: *"I am <u>still</u> waiting for packet with SQNR=300"***



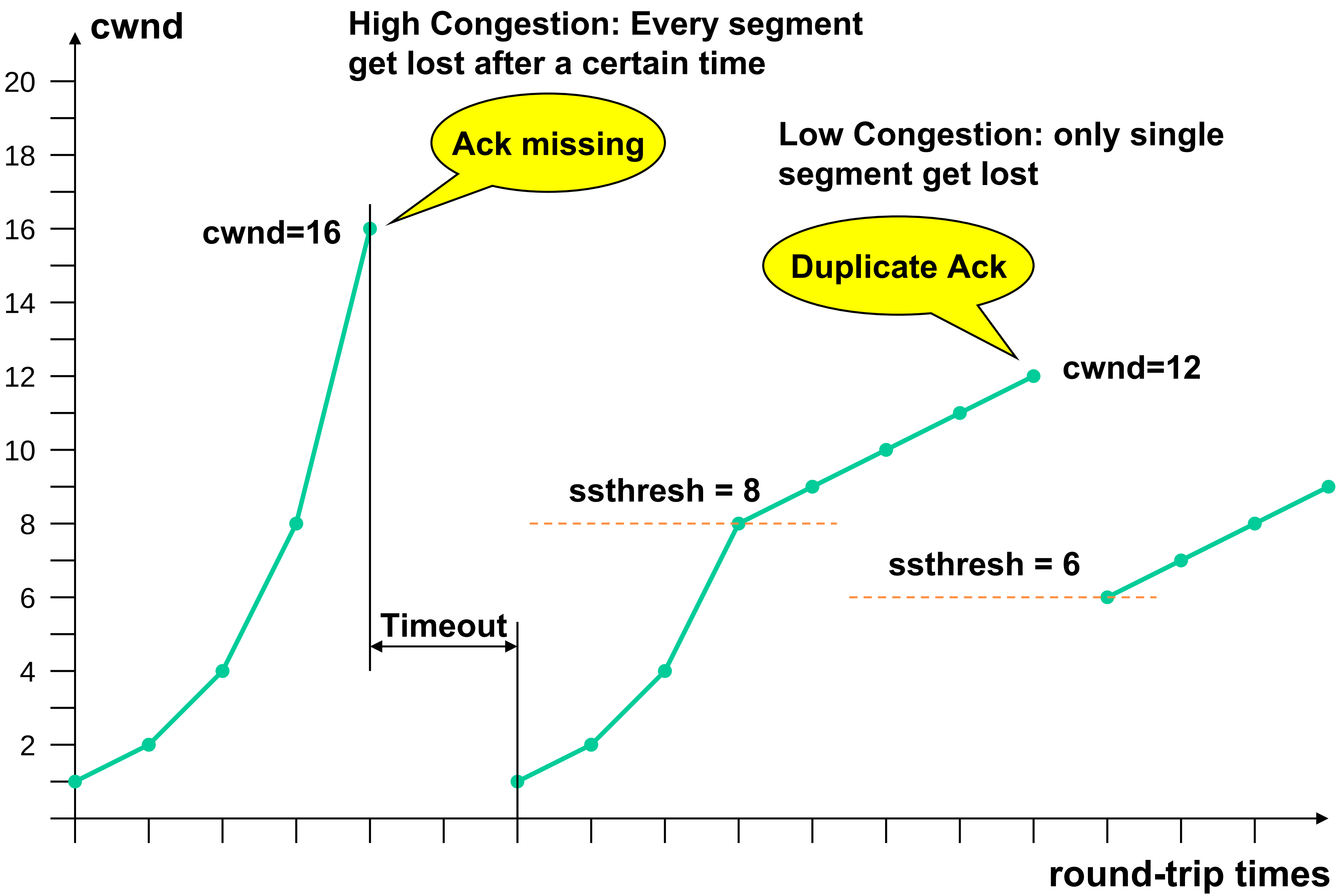
Congestion Avoidance (1)

- **Congestion Avoidance is the companion algorithm to Slow Start – both are usually implemented together !**
- **Idea: Upon congestion (=duplicate ACKs) reduce the sending rate by half and now increase the rate *linearly* until duplicate ACKs are seen again (and repeat this continuously)**
  - **Introduces another variable: the Slow Start threshold (ssthresh)**
- **Note this central TCP assumption: Packets are dropped because of buffer overflows and NOT because of bit errors!**
  - **Therefore packet loss indicates congestion somewhere in the network**

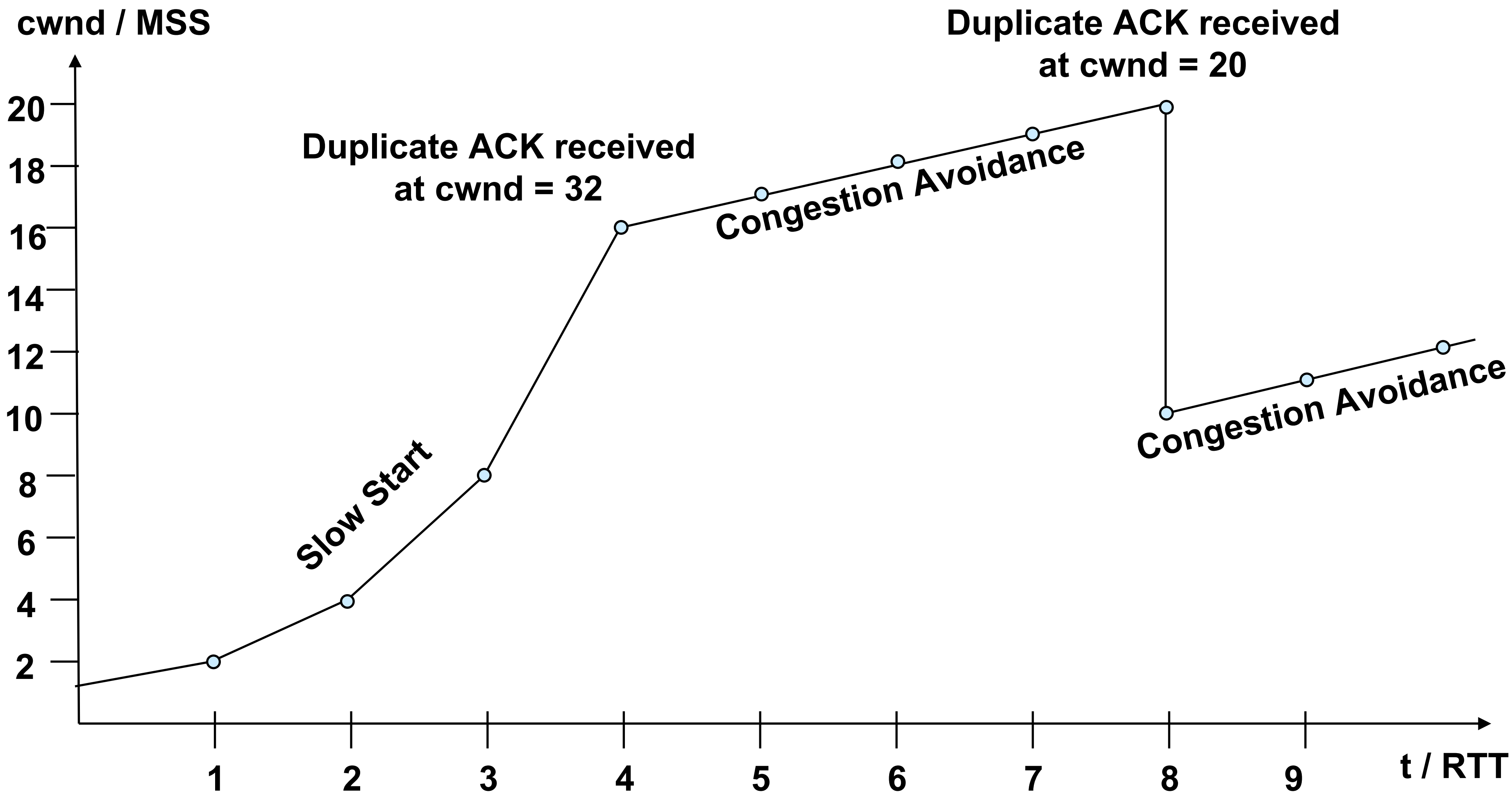
The combined algorithm



Slow Start and Congestion Avoidance



Slow Start and Congestion Avoidance



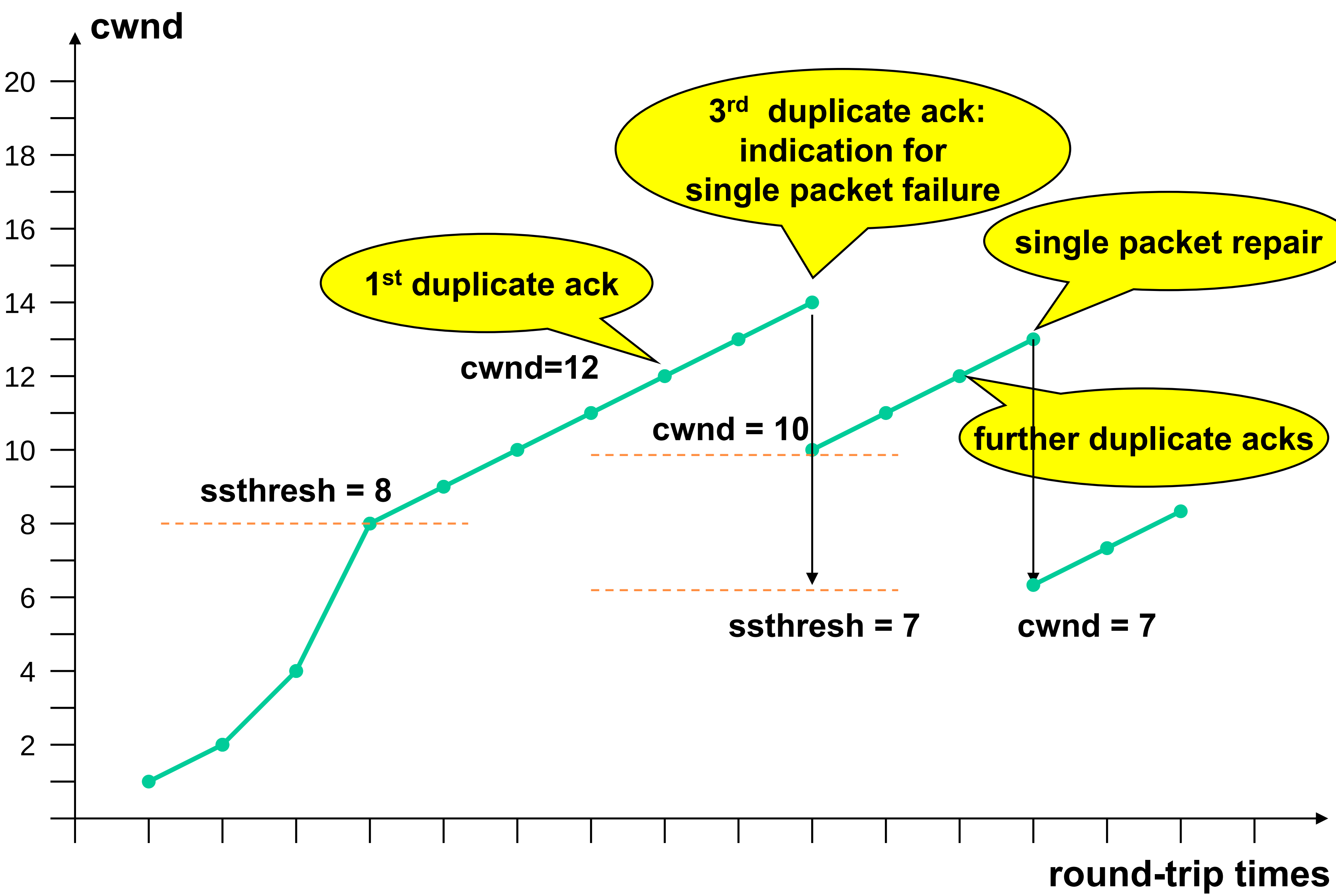
"Fast Retransmit"

- **Note that duplicate ACKs are also sent upon packet reordering**
- **Therefore TCP waits for 3 duplicate ACKs before it really assumes congestion**
  - **Immediate retransmission (don't wait for timer expiration)**
- **This is called the *Fast Retransmit* algorithm**

"Fast Recovery"

- After Fast Retransmit TCP continues with Congestion Avoidance
  - Does NOT fall back to Slow Start
- Every another duplicate ACK tells us that a "good" packet has been received by the peer
  - cwnd = cwnd + MSS
  - => Send one additional segment
- As soon a normal ACK is received
  - cwnd = ssthresh = Min(W, cwnd)/2
- This is called Fast Recovery

Fast Retransmit and Fast Recovery



All together!

Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery

New Session: initialize cwnd = 1 MSS, ssthresh = 65535

Determine actual window size "AWS" = Min (W, cwnd)
**** send AWS bytes ****

3 duplicate ACKs received

ssthresh = AWS/2
(but at least 2 MSS),
retransmit the segment,
cwnd = ssthresh+3 MSS,
for each 3+nth duplicate ACK
increase cwnd by 1 MSS;
then set cwnd=ssthresh upon
first "normal" ACK

Retransmission timeout expired

cwnd = 1
ssthresh = AWS/2

Data acknowledged

(cwnd > ssthresh) ?

yes

Increment cwnd by 1/cwnd for each ACK received

no

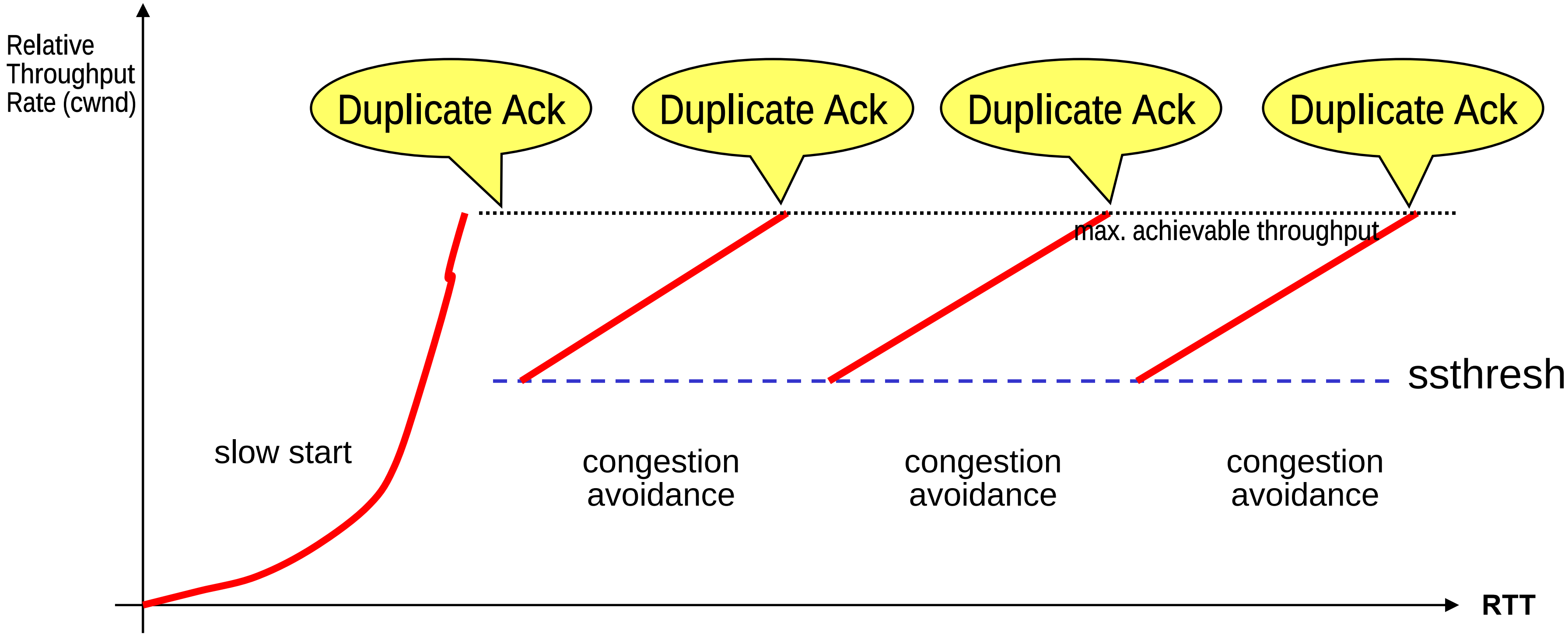
Increment cwnd by one for each ACK received.

Real TCP Performance

- TCP always tries to minimize the data delivery time
- Good and proven self-regulating mechanism to avoid congestion
- TCP is "hungry but fair"
  - Essentially fair to other TCP applications
  - Unreliable traffic (e. g. UDP) is not fair to TCP…

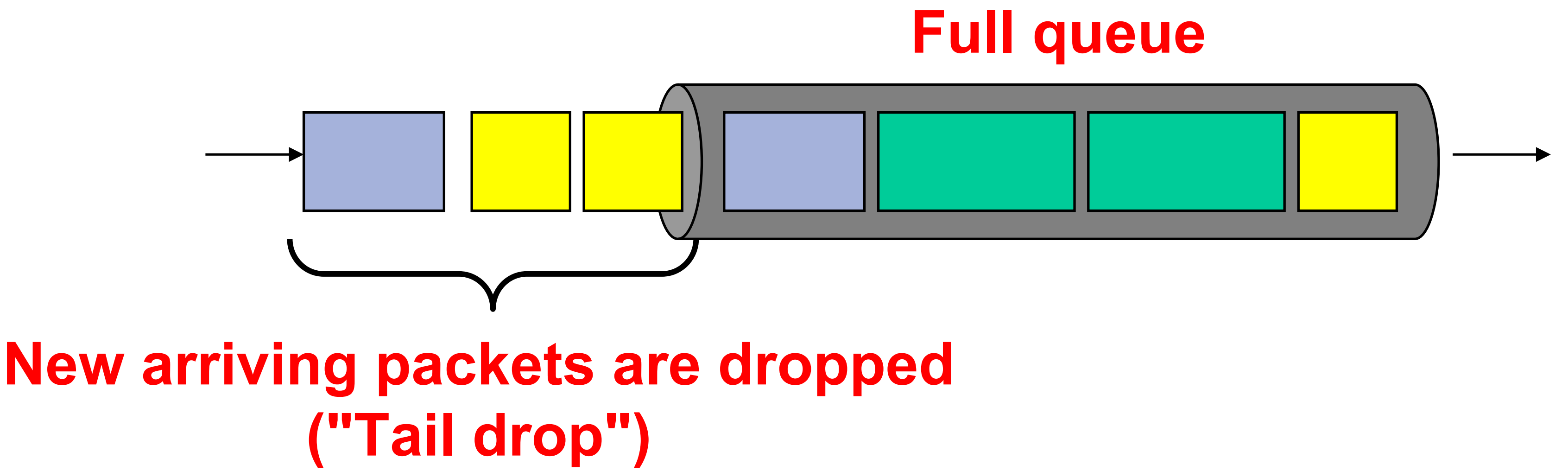
Summary: The TCP "wave"

- Tries to fill the "pipe" using
  - Slow Start and
  - Congestion Avoidance



What's happening in the network?

- ***Tail-drop queuing*** **is the standard dropping behavior in FIFO queues**
  - **If queue is full all subsequent packets are dropped**



Tail-drop Queuing (cont.)

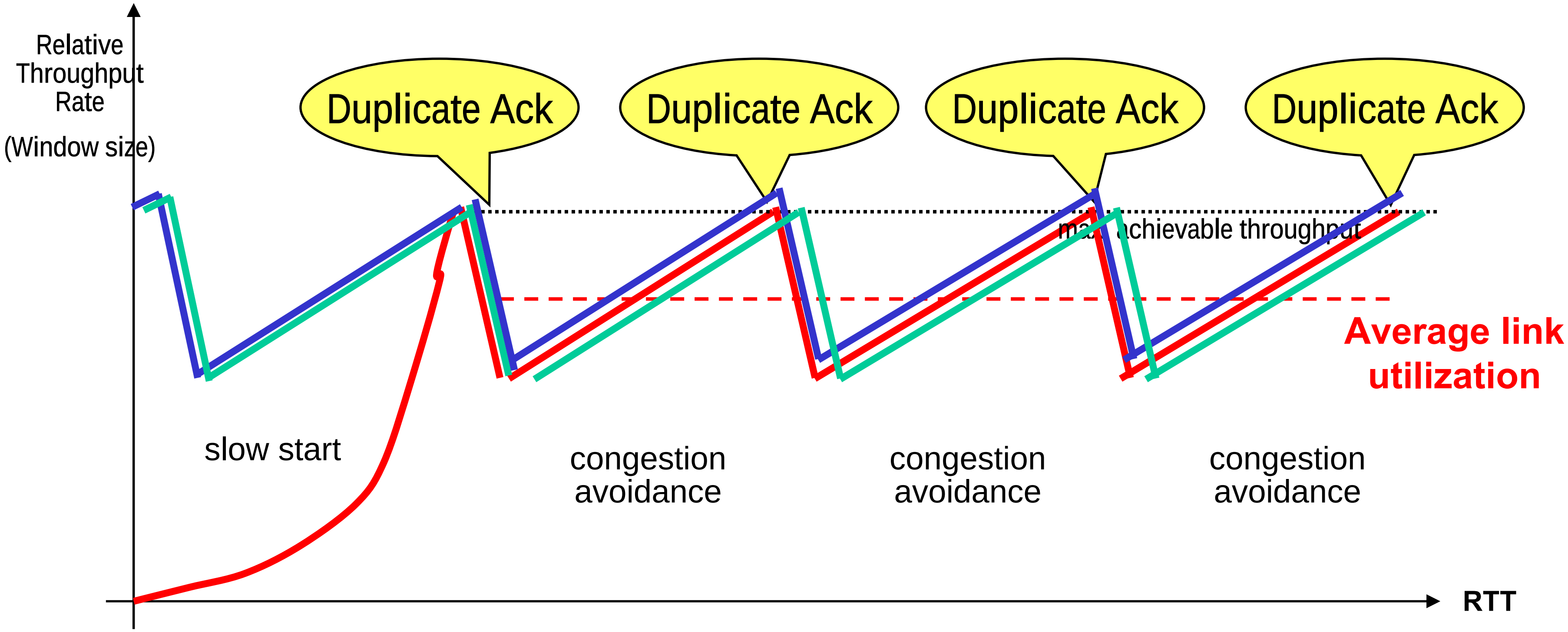
- **Another representation: Drop probability versus queue depth**

Tail-drop Problems

- No flow differentiation
- TCP starvation upon multiple packet drop
  - TCP receivers may keep quiet (not even send Duplicate ACKs) and sender falls back to slow start
    – worst case!
  - TCP fast retransmit and/or selective acknowledgement may help
- TCP synchronization

TCP Synchronization

- **Tail-drop drops many packets of different sessions at the same time**
- **All these sessions experience duplicate ACKs and perform synchronized congestion avoidance**



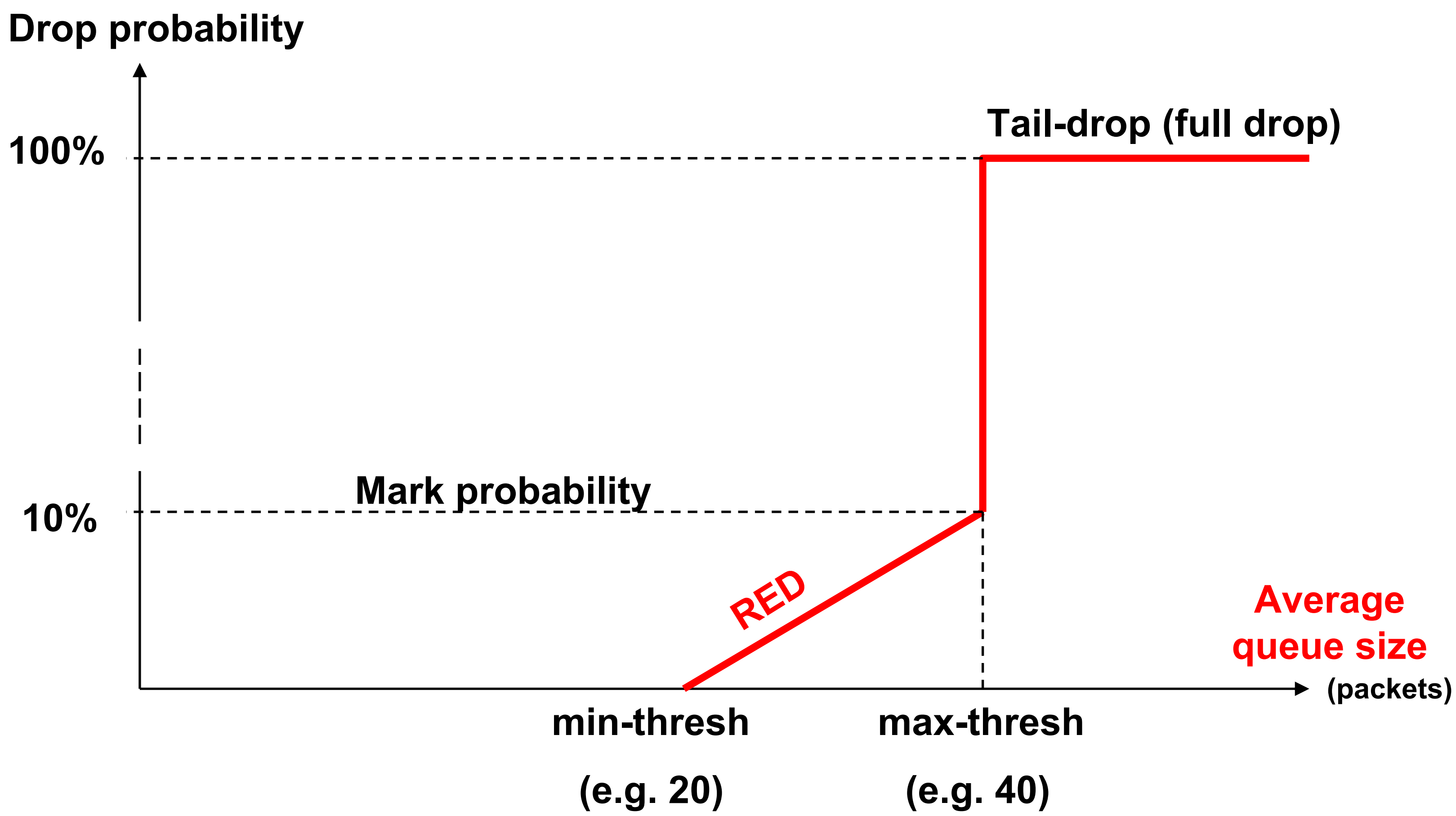
Random Early Detection (RED)

- Utilizes TCP specific behavior
  - TCP dynamically adjusts traffic throughput to accommodate to minimal available bandwidth (bottleneck) via reduced window size
- "Missing" (dropped) TCP segments cause window size reduction!
  - Idea: Start dropping TCP packets before queuing "tail-drops" occur
  - Make sure that "important" traffic is not dropped
- RED randomly drops packets before queue is full
  - Drop probability increases linearly with queue depth

RED

- Important RED parameters
  - Minimum threshold
  - Maximum threshold
  - Average queue size (running average)
- RED works in three different modes
  - No drop
    - If average queue size is between 0 and minimum threshold
  - Random drop
    - If average queue size is between minimum and maximum threshold
  - Full drop
    - If average queue size is equal or above maximum threshold = "tail-drop"

RED Parameters



Weighted RED (WRED)

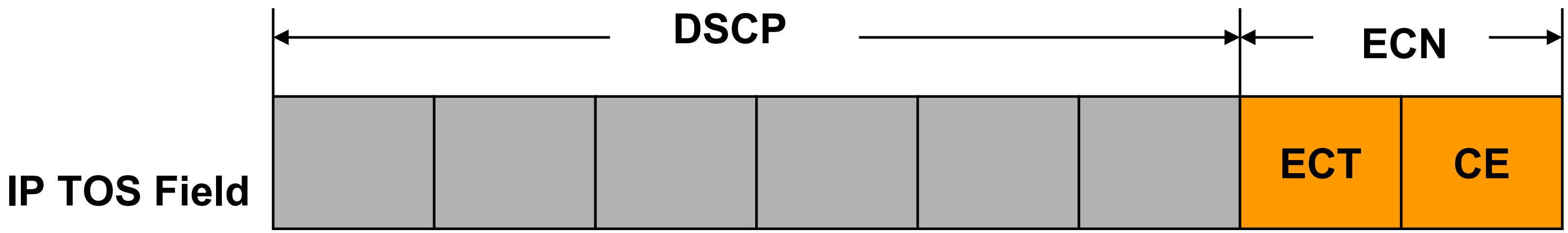
- Drops less important packets more aggressively than more important packets
- Importance based on:
  - IP precedence 0-7
  - DSCP value 0-63
- Classified traffic can be dropped based on the following parameters
  - Minimum threshold
  - Maximum threshold
  - Mark probability denominator
    (Drop probability at maximum threshold)

RED Problems

- RED performs "Active Queue Management" (AQM) and drops packets before congestion occurs
  - But an uncertainty remains whether congestion will occur at all
- RED is known as "difficult to tune"
  - Goal: Self-tuning RED
  - Running estimate weighted moving average (EWMA) of the average queue size

Explicit Congestion Notification (ECN)

- Traditional TCP stacks only use packet loss as indicator to reduce window size
  - But some applications are sensitive to packet loss and delays
- Routers with ECN enabled mark packets when the average queue depth exceeds a threshold
  - Instead of randomly dropping them
  - Hosts may reduce window size upon receiving ECN-marked packets
- Least significant two bits of IP TOS used for ECN



Obsolete (but widely used) RFC 2481 notation of these two bits:

ECT	ECN-Capable Transport
CE	Congestion Experienced

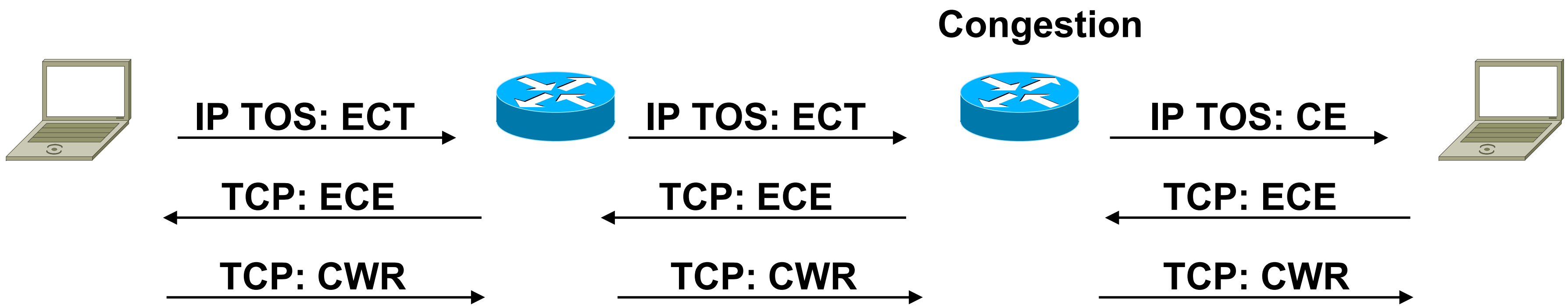
Usage of CE and ECT

- RFC 3168 redefines the use of the two bits: ECN-supporting hosts should set one of the two ECT code points
  - ECT(0) or ECT(1)
  - ECT(0) SHOULD be preferred
- Routers that experience congestion set the CE code point in packets with ECT code point set (otherwise: RED)
- If average queue depth is exceeding max-threshold: Tail-drop
- If CE already set: forward packet normally (abuse!)

Non ECN-capable transport	0	0	
Codepoints for ECN-capable transport	0	1	ECT(1)
	1	0	ECT(0)
CE codepoint	1	1	

CWR and ECE

- RFC 3168 also introduced two new TCP flags
  - ECN Echo (ECE)
  - Congestion Window Reduced (CWR)
- Purpose:
  - ECE used by data receiver to inform the data sender when a CE packet has been received
  - CWR flag used by data sender to inform the data receiver that the congestion window has been reduced



Part of TCP header:

Header Length	Reserved	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size

ECN Configuration

- **Note: ECN is an extension to WRED**
  - **Therefore WRED must be enabled first !**
- **ECN will be applied on that traffic that is identified by WRED (e. g. dscp-based)**

```
(config-pmap-c)# random-detect
(config-pmap-c)# random-detect ecn

# show policy-map interface s0/1  !!! shows ECN setting
```

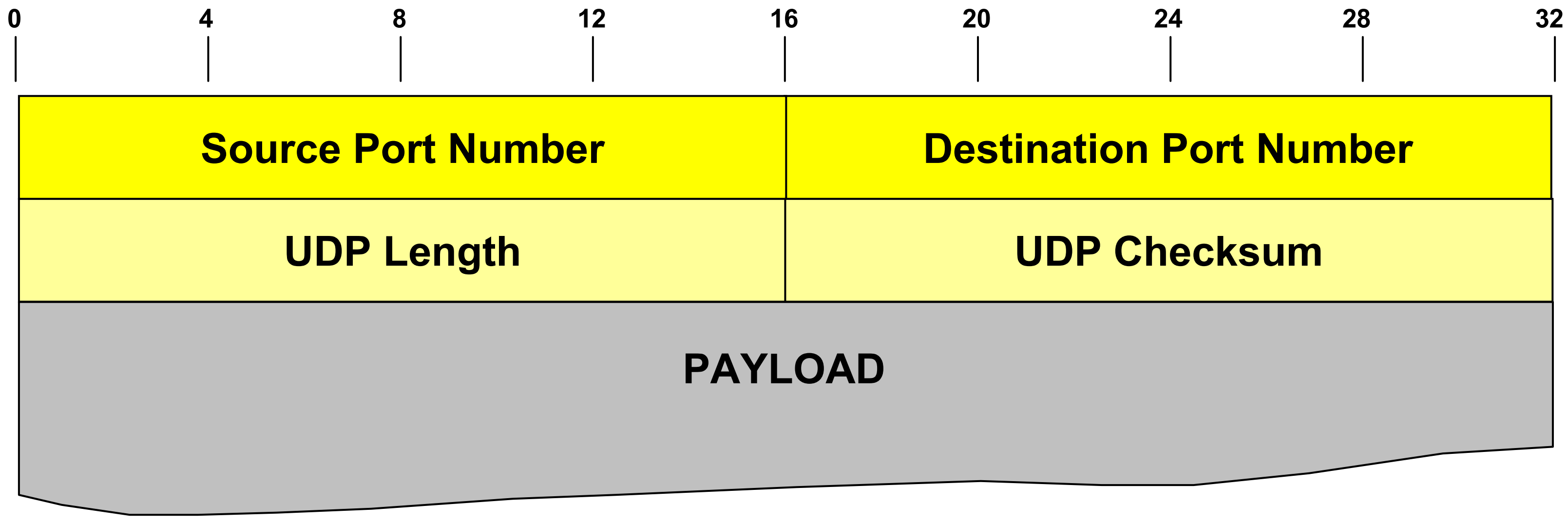
Note

- CE is only set when average queue depth exceeds a threshold
  - End-host would react immediately
  - Therefore ECN is not appropriate for short term bursts (similar as RED)
- Therefore ECN is different as the related features in Frame Relay or ATM which acts also on short term (transient) congestion

UDP

- UDP is a connectionless layer 4 service (datagram service)
- Layer 3 Functions are extended by port addressing and a checksum to ensure integrity
- UDP uses the same port numbers as TCP (if applicable)
- UDP is used, where the overhead of a connection oriented service is undesirable or where the implementation has to be small
  - DNS request/reply, SNMP get/set, booting by TFTP
- Less complex than TCP, easier to implement

UDP Header


0
4
8
12
16
20
24
28
32
Source Port Number
Destination Port Number
UDP Length
UDP Checksum
PAYLOAD

UDP

- **Source and Destination Port**
  - Port number for addressing the process (application)
  - Well known port numbers defined in RFC1700
- **UDP Length**
  - Length of the UDP datagram (Header plus Data)
- **UDP Checksum**
  - Checksum includes pseudo IP header
    (IP src/dst addr., protocol field),
    UDP header and user data;
    one´s complement of the sum of all one´s complements

Other Transport Layer Protocols

SCTP

UDP Lite

DCCP

Stream Control Transmission Protocol (SCTP)

- A newer improved alternative to TCP (RFC 4960)
- Supports
  - Multi-homing
  - Multi-streaming
  - Heart-beats
  - Resistance against SYN-Floods (via Cookies) and 4-way handshake)
- Seldom used today
  - Base for the Reliable Server Pooling Protocol (RSerPool)

UDP Lite

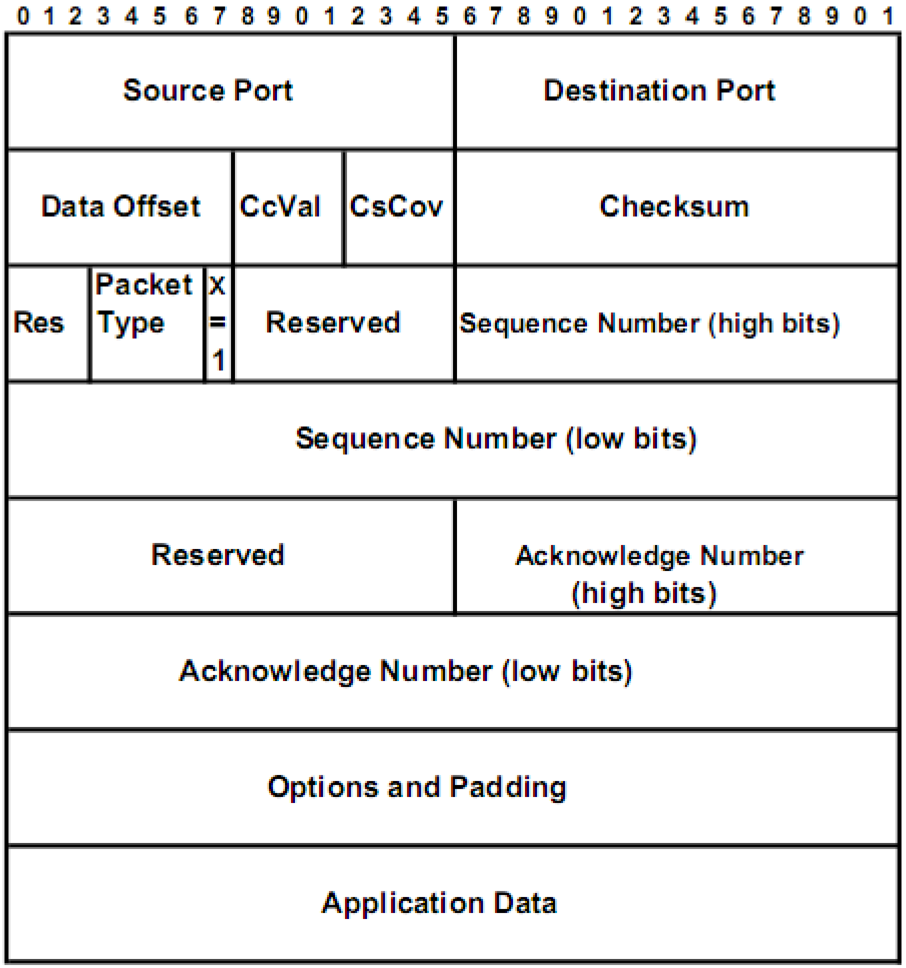
- Problem: Lots of applications would like to receive even (slightly) corrupted data
  - E. g. multimedia
- UDP Lite (RFC 3828) defines a different usage of the UDP length field
  - UDP length field indicates how many bytes of the datagram are really protected by the checksum ("checksum coverage")
  - True length shall be determined by IP length field
- Currently only supported by Linux

Datagram Congestion Control Protocol (DCCP)

- Problem: More and more applications use UDP instead of TCP
- But UDP does not support congestion control – networks might collapse!
- DCCP adds a congestion control layer to UDP
  - RFC 4340
  - First implementations now in FreeBSD and Linux

DCCP (cont.)

- 4-way handshake
- Different procedures compared to TCP regarding sequence number handling and session creation



Summary

- TCP & UDP are Layer 4 (Transport) Protocols above IP
- TCP is "Connection Oriented"
- UDP is "Connection Less"
- TCP implements "Fault Tolerance" using "Positive Acknowledgement"
- TCP implements "Flow Control" using dynamic window-sizes
- The combination of IP-Address and TCP/UDP-Port is called a "Socket"