L33 - Internet Transport Layer

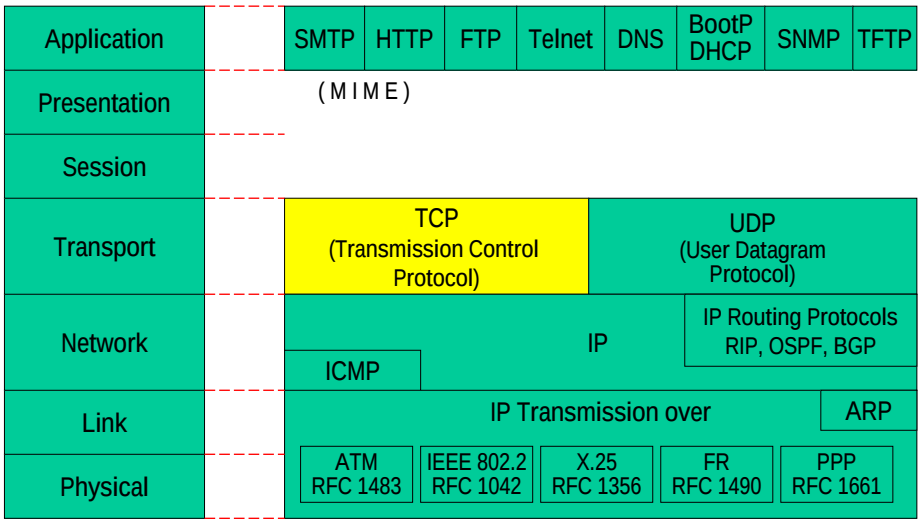
Internet Transport Layer

TCP Fundamentals, TCP Performance Aspects,
UDP (User Datagram Protocol)

Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

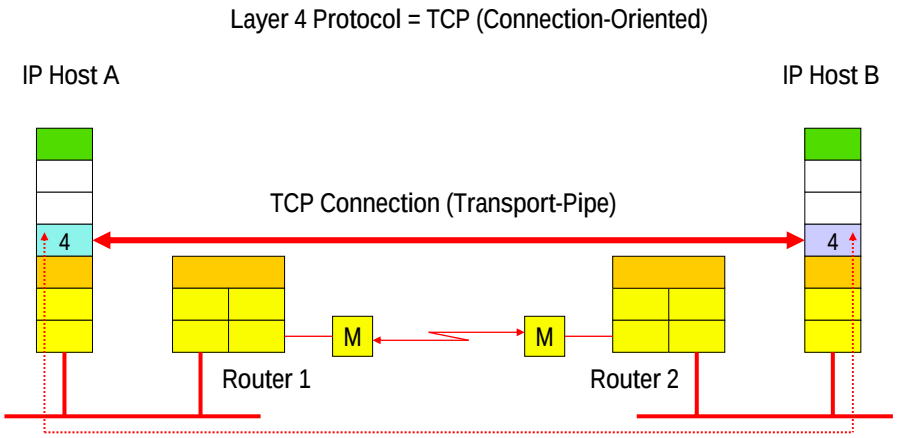
Transport Layer Protocols



TCP (Transmission Control Protocol)

- **TCP is a connection oriented layer 4 protocol (transport layer) and is transmitted inside the IP data field**
- **Provides a secure end-to-end transport of data between computer processes of different end systems**
- **Secure transport means:**
  - Error detection and recovery
  - Maintaining the order of the data without duplication or loss
  - Flow control
- **RFC 793**

TCP and OSI Transport Layer 4



TCP Protocol Functions 1

- Data transmission within segments
- ARQ protocol with Continuous Repeat Request technique and piggy-backed acknowledgments
- Error recovery through sequence numbers (based on octets !), positive & multiple acknowledgements and timeouts for each segment
- Flow control with sliding window and dynamically adjusted window size

TCP Protocol Functions 2

- In general, segments are encapsulated in single IP packets
- Maximum segment size depends on max. packet or frame size (fragmentation is possible)
- Call setup with "three way handshake"
- Hides the details of the network layer from the higher layers and frees them from the tasks of transmitting data through a specific network

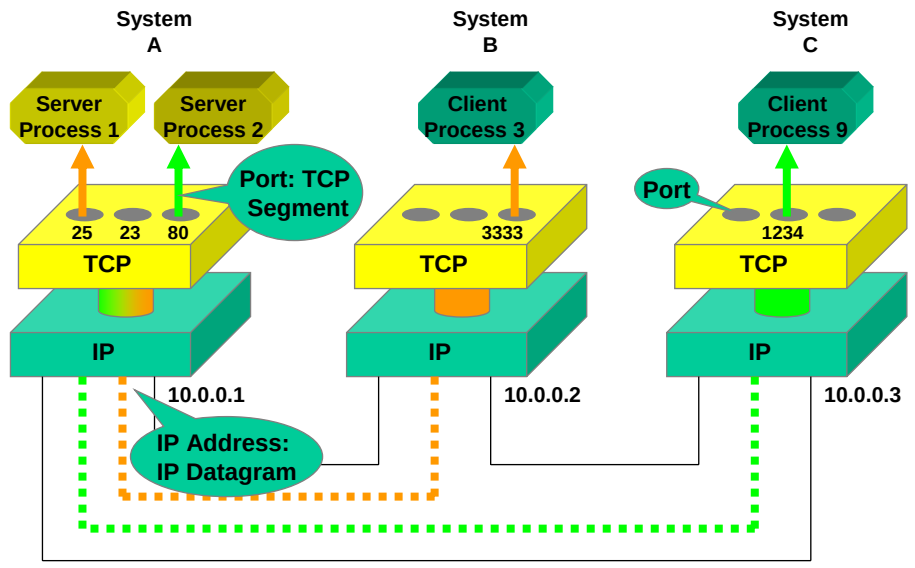
TCP Ports

- TCP provides its service to higher layers through ports
- A port can be compared to a SAP (OSI Service Access Point) or a Novell IPX-socket
- Each communicating computer process is assigned a port
  - identified by a port number

TCP Ports and Connections

- **By usage of ports the TCP software can serve multiple processes (browser, e-mail etc.) simultaneously**
- **The TCP software functions like a multiplexer and demultiplexer for TCP connections**
  - Port 25 on system A:
    - process 1, system A <--------> process 3, system B
  - Port 53 on system A:
    - process 2, system A <--------> proc. 9, system C
  - (see next slide)

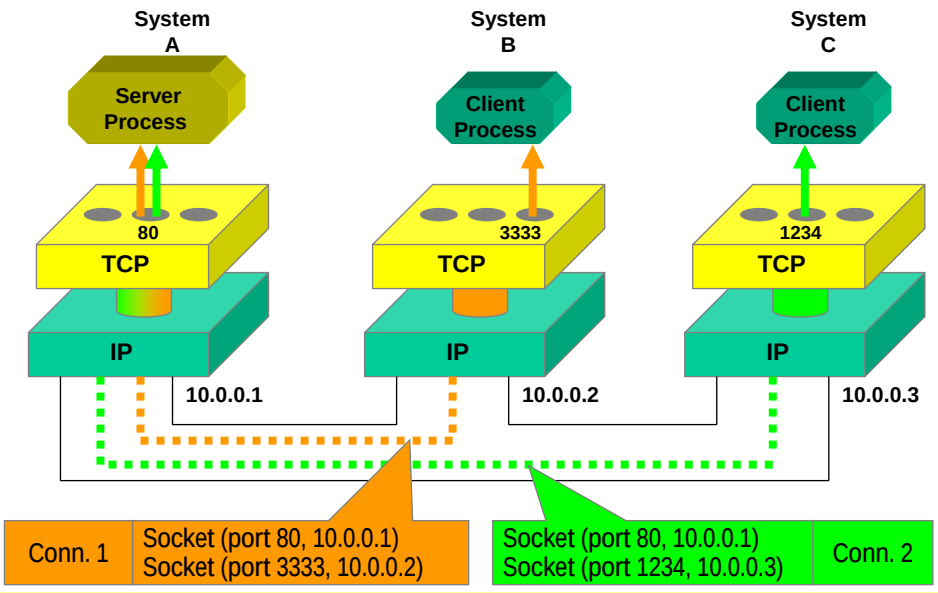
TCP Ports and Connections



TCP Sockets and Connections

- In a client-server environment a communicating server-process has to maintain several sessions (and also connections) to different targets at the same time
- Therefore, a single port has to multiplex several virtual connections; these connections are distinguished through sockets
- The combination IP address and port number is called a "socket"
  - similar to the OSI "CEP" Connection Endpoint Identifier
- Each socket pair uniquely identifies a connection

TCP Sockets and Connections



TCP Well Known Ports

- **Well known ports**
  - Are reserved for common applications and services (like Telnet, WWW, FTP etc.) and are in the range from 0 to 1023
  - Are controlled by IANA (Internet Assigned Numbers Authority)
- **Registered ports**
  - start at 1024 (e.g. Lotus Notes, Cisco XOT, Oracle, license managers etc.). They are not controlled by the IANA (only listed, see RFC1700)
- **Port concept und port numbers also used for UDP**

TCP Well-Known Ports

- **Well-known ports together with the socket concept allow several simultaneous connections (even from a single machine) to a specific server application**
- **Server applications listen on their well-known ports for incoming connections**

Use of Port Numbers

- **Client applications chose a free port number (which is not already used by another connection) as the source port**
- **The destination port is the well-known port of the server application**
- **Some services like FTP or Remote Procedure Call use dynamically assigned port numbers:**
  - Sun RPC (Remote Procedure Call) uses a portmapper located at port 111...
  - FTP uses the PORT and PASV commands...
- **...to switch to a non-standard port**

Well Known Ports

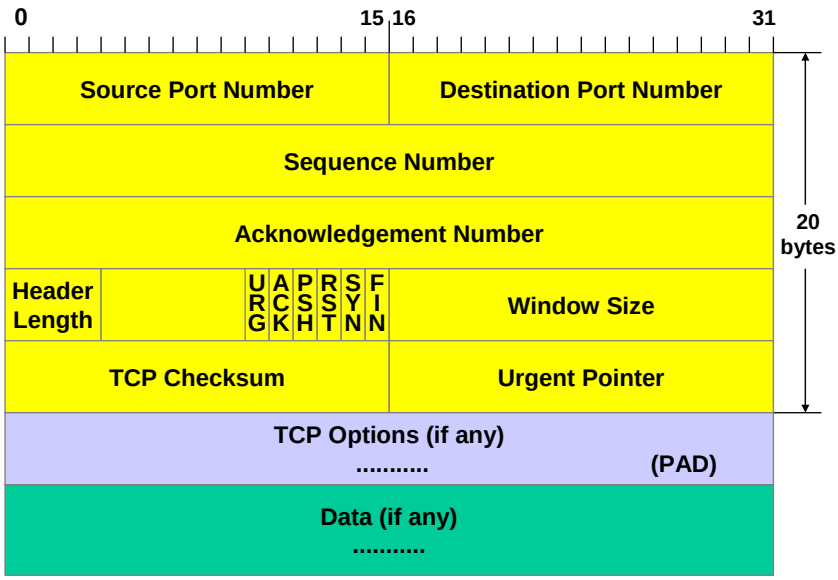
Some Well Known Ports	
7	Echo
20	FTP (Data), File Transfer Protocol
21	FTP (Control)
23	TELNET, Terminal Emulation
25	SMTP, Simple Mail Transfer Protocol
53	DNS, Domain Name Server
69	TFTP, Trivial File Transfer Protocol
80	HTTP Hypertext Transfer Protocol
111	Sun Remote Procedure Call (RPC)
137	NetBIOS Name Service
138	NetBIOS Datagram Service
139	NetBIOS Session Service
161	SNMP, Simple Network Management Protocol
162	SNMPTRAP
322	RTSP (Real Time Streaming Protocol) Server

Some Registered Ports	
1416	Novell LU6.2
1433	Microsoft-SQL-Server
1439	Eicon X25/SNA Gateway
1527	oracle
1986	cisco license managmt
1998	cisco X.25 service (XOT)
5060	SIP (VoIP Signaling)
6000	\
.....	> X Window System
6063	/

... etc.

(see RFC1700)

TCP Header



TCP Header Entries

- **Source and Destination Port**
  - Port number for source and destination process
- **Header Length**
  - Indicates the length of the header given as a multiple of 32 bit words (4 octets)
  - necessary, because of the variable header length

TCP Header Entries

- **Sequence Number (32 Bit)**
  - Position of the first octet of this segment within the data stream ("wraps around" to 0 after reaching 2^32 -1)
- **Acknowledge Number (32 Bit)**
  - Acknowledges the correct reception of all octets up to ack-number minus 1 and indicates the number of the next octet expected by the receiver

TCP Header Entries

- **Flags: SYN, ACK**
  - SYN: If set, the Sequence Number holds the initial value for a new session
    - SYN is used only during the connect phase (can be used to recognize who is the caller during a connection setup e.g. for firewall filtering)
  - Used for call setup (connect request)
  - ACK: If set, the Acknowledge Number is valid and indicates the sequence number of the next octet expected by the receiver

TCP Header Entries

- **Flags: FIN, RST**
  - FIN: If set, the Sequence Number holds the number of the last transmitted octet of this session
    - using this number a receiver can tell that all data have been received; FIN is used only during the disconnect phase
  - Used for call release (disconnect)
  - RST: If set, the session has to be cleared immediately
    - Can be used to refuse a connection-attempt or to "kill" a current connection.

TCP Header Entries

- **Window (16 Bit)**
  - Set by the source with every transmitted segment to signal the current window size; this "dynamic windowing" enables receiver-based flow control
  - The value defines how many additional octets will be accepted, starting from the current acknowledgment number
    - SeqNr of last octet allowed to sent: AckNr plus window value
  - Remarks:
    - Once a given range for sending data was given by a received window value, it is not possible to shrink the window size to such a value which gets in conflict with the already granted range
    - so the window field must be adapted accordingly in order to achieve the flow control mechanism STOP

TCP Header Entries

- **Checksum**
  - The checksum includes the TCP header and data area plus a 12 byte pseudo IP header
    - (one´s complement of the sum of all one´s complements of all 16 bit words)
  - The pseudo IP header contains the source and destination IP address, the IP protocol type and IP segment length (total length). This guarantees, that not only the port but the complete socket is included in the checksum
  - Including the pseudo IP header in the checksum allows the TCP layer to detect errors, which can't be recognized by IP (e.g. IP transmits an error-free TCP segment to the wrong IP end system)

TCP Header Entries

- **Flags: URG**
  - Is used to indicate important - "urgent" - data
  - If set, the 16-bit "Urgent Pointer" field is valid and points to the last octet of urgent data
    - sequence number of last urgent octet = actual segment sequence number + urgent pointer
    - RFC 793 and several implementations assume the urgent pointer to point to the first octet *after* the urgent data; However, the "Host Requirements" RFC 1122 states this as a mistake!
    - Note: There is no way to indicate the beginning of urgent data (!)
  - When a TCP receives a segment with the URG flag set, it notifies the application which switch into the "urgent mode" until the last octet of urgent data is reached
  - Examples for use: Interrupt key in Telnet, Rlogin, or FTP

TCP Header Entries

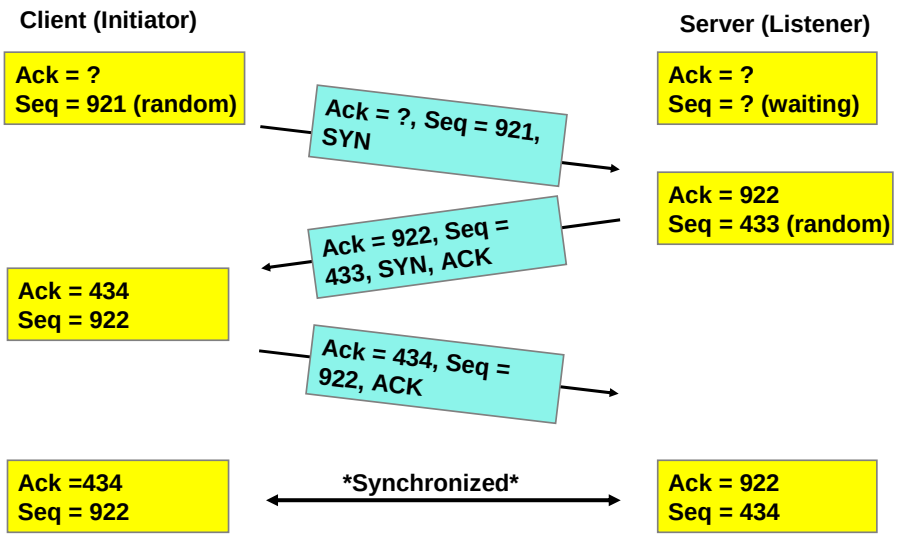
- **Urgent Pointer**
  - points to the last octet of urgent data
- **Options**
  - Only MSS (Maximum Segment Size) is used frequently Other options are defined in RFC1146, RFC1323 and RFC1693
- **Pad**
  - Is used to make the header length an integral number of 32 bits (4 octets) because of the variable length options

TCP Header Entries

- **Flags: PSH**
  - "PUSH": If set, the segment should be forwarded to the next layer immediately without buffering

    A TCP instance can decide on its own, when to send data to the next instance. One strategy could be, to collect data in a buffer and forward the data when the buffer exceeds a certain size. An application which needs a low latency or a constant data stream would like to bypass this buffer with the PSH flag. Also the last segment of a request could use the PSH flag.
  - Today often ignored

TCP Connect Phase



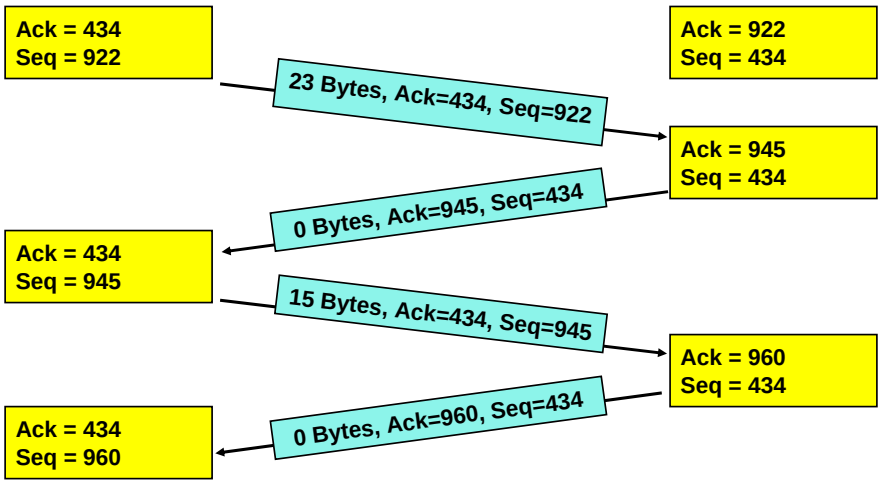
TCP Connect Phase

- TCP uses the unreliable service of IP, hence TCP segments of old sessions (e.g. retransmitted or delayed segments) could disturb a TCP connect
- Random starting sequence numbers and an explicit negotiation of starting sequence numbers makes a TCP connect immune against spurious packets
- Disturbing Segments (e.g. delayed TCP segments from old sessions etc.) and old "half-open" connections are deleted with the RST flag
- --> "Three Way Handshake"

TCP Duplicates after Connect

- Duplicates of old TCP sessions (same source/destination IP address and TCP socket) can disturb a new session
- Thus sequence numbers must be unique for different sessions of the same socket.
- Initial sequence number (ISN) must be chosen with a good algorithm
- 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 increment additionally by 1

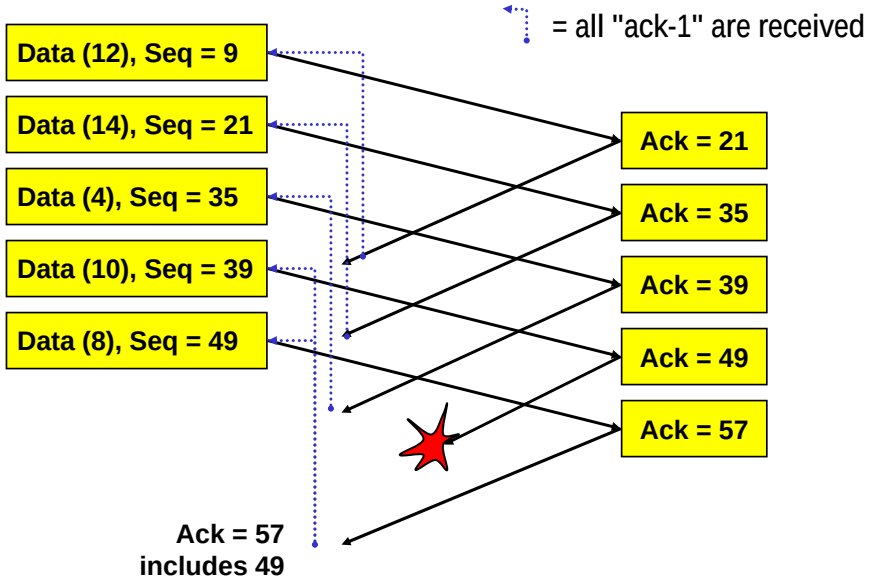
TCP Data Transfer Phase



TCP Data Transfer Phase

- **Acknowledgements are generated for all octets which arrived in sequence without errors (positive acknowledgement)**
  - Note: duplicates are also acknowledged
  - If a segment arrives out of sequence, no acknowledges are sent until this "gap" is closed
- **The acknowledge number is equal to the sequence number of the next octet to be received**
  - Acknowledges are "cumulative": Ack(N) confirms all octets with sequence numbers up to N-1
  - Thus, lost acknowledgements are not critical since the following ack confirms all previous segments

TCP "Cumulative" Acknowledgement



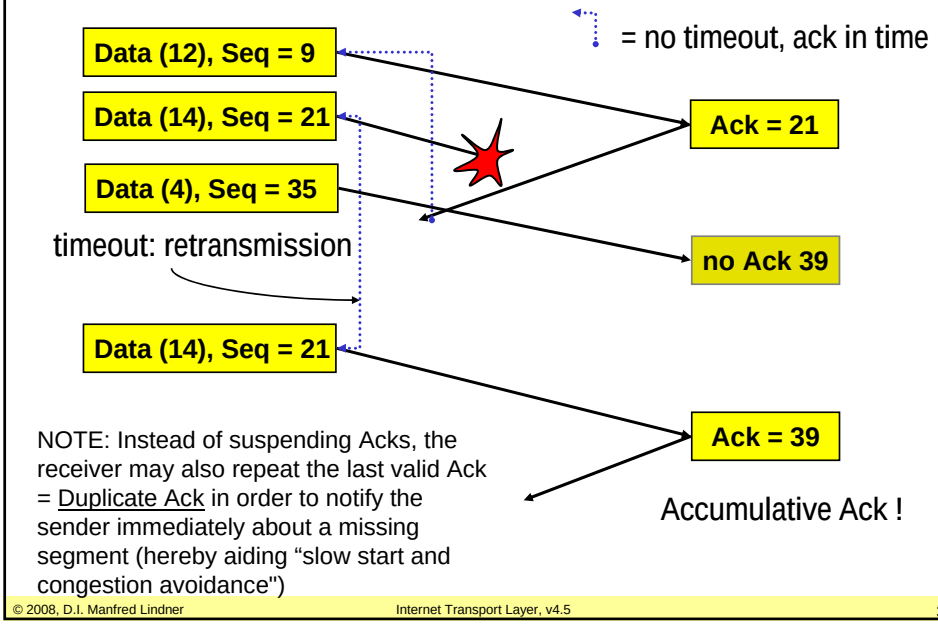
TCP Timeout

- **Timeout will initiate a retransmission of unacknowledged data**
- **Value of retransmission timeout influences performance (timeout should be in relation to round trip delay)**
  - High timeout results in long idle times if an error occurs
  - Low timeout results in unnecessary retransmissions
- **Adaptive timeout**
  - KARN algorithm uses a backoff method to adapt to the actual round trip delay

TCP Duplicates

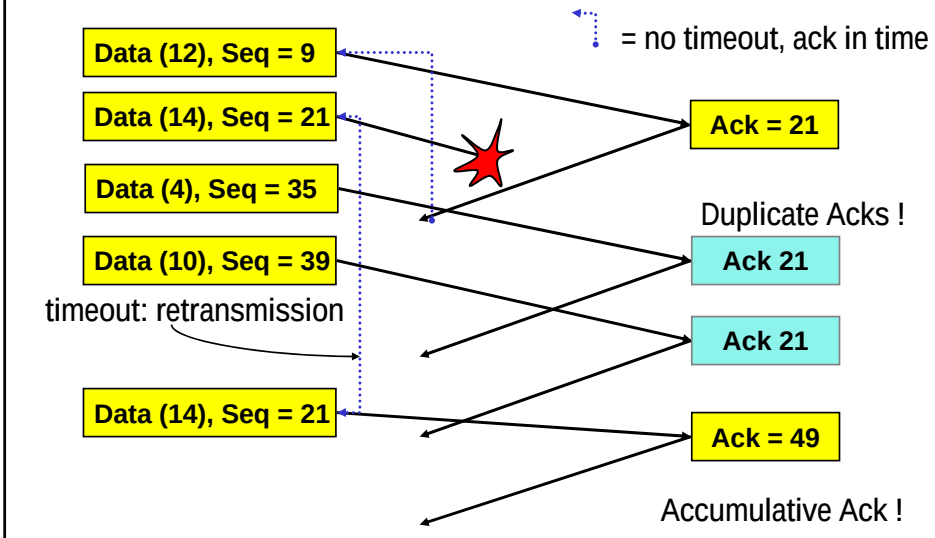
- **Reasons for retransmission:**
  - Because original segment was lost:
    No problem, retransmitted segment fills gap, no duplicate
  - Because ACK was lost or retransmit timeout expired:
    No problem, segment is recognized as duplicate through the sequence number
  - Because original was delayed and timeout expired:
    No problem, segment is recognized as duplicate through the sequence number
- **32 bit sequence numbers provide enough "space" to differentiate duplicates from originals**
  - 2^{32} Octets with 2 Mbit/s means 9h for wrap around (compare to usual TTL = 64 seconds)

TCP Duplicates, Lost Original (old TCP)

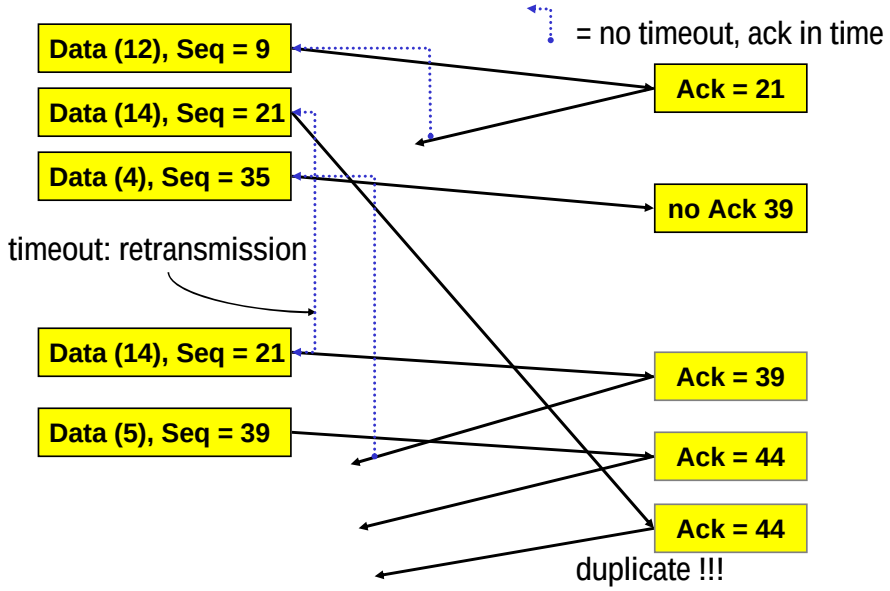


NOTE: Instead of suspending Acks, the receiver may also repeat the last valid Ack = Duplicate Ack in order to notify the sender immediately about a missing segment (hereby aiding “slow start and congestion avoidance")

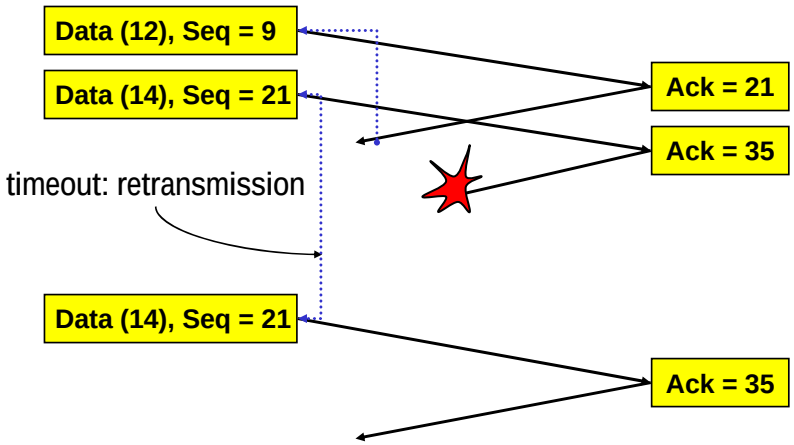
TCP Duplicate-Ack (new TCP)



TCP Duplicates, Delayed Original



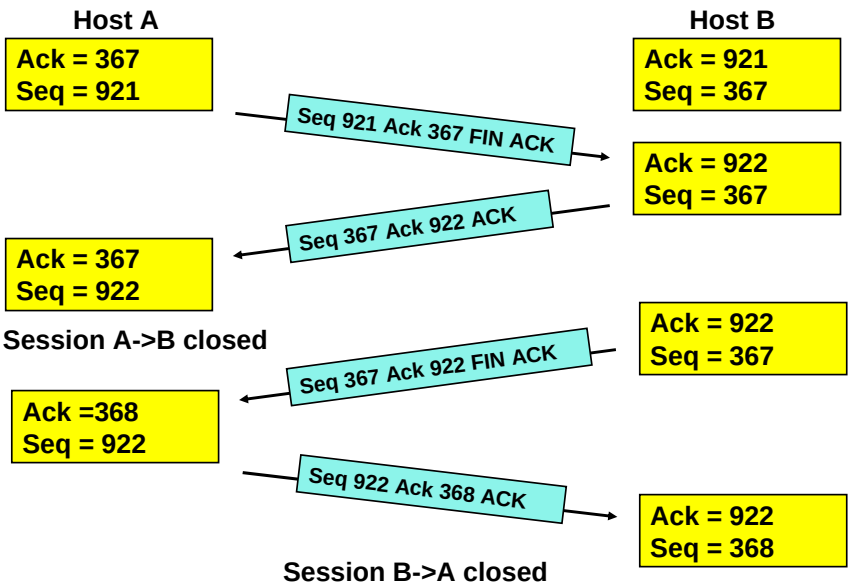
TCP Duplicates, Lost Acknowledgement



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

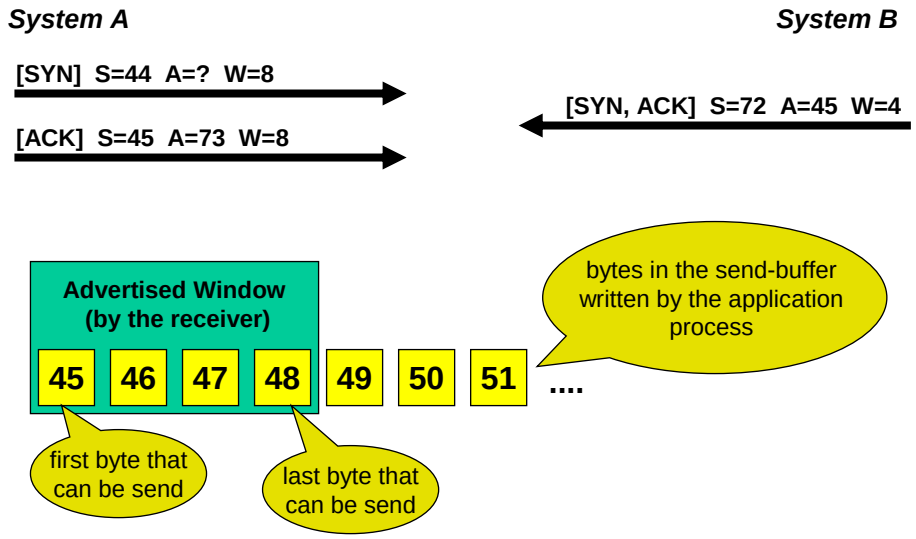
TCP Disconnect



Flow control: "Sliding Window"

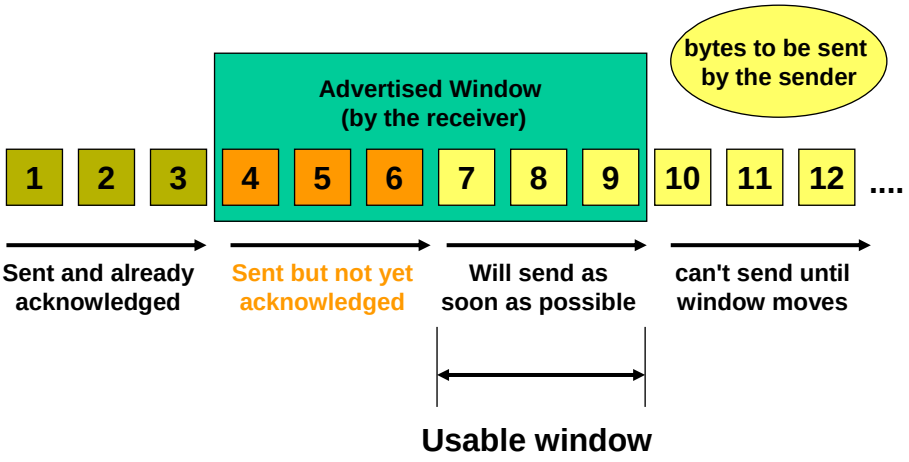
- **TCP flow control is done with dynamic windowing using the sliding window protocol**
- **The receiver advertises the current amount of octets it is able to receive**
  - using the window field of the TCP header
  - values 0 through 65535
- **Sequence number of the last octet a sender may send = received 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

Sliding Window: Initialization



Sliding Window: Principle

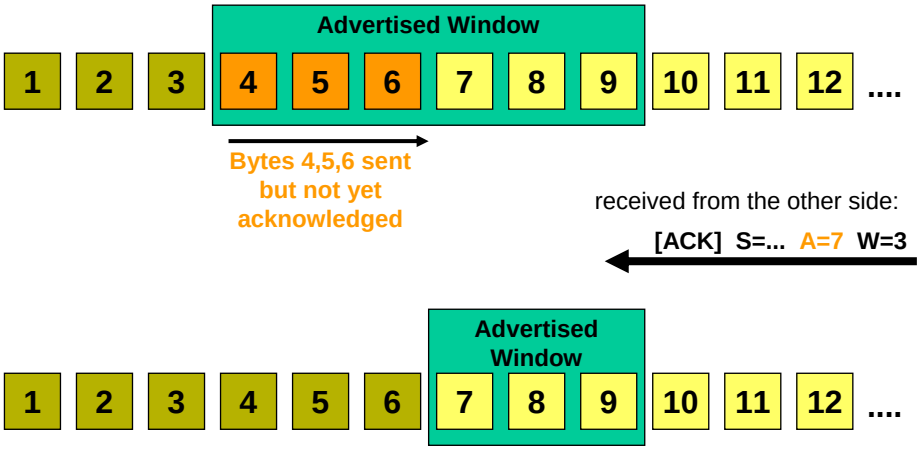
Sender's point of view; sender got {ACK=4, WIN=6} from the receiver.



Sliding Window

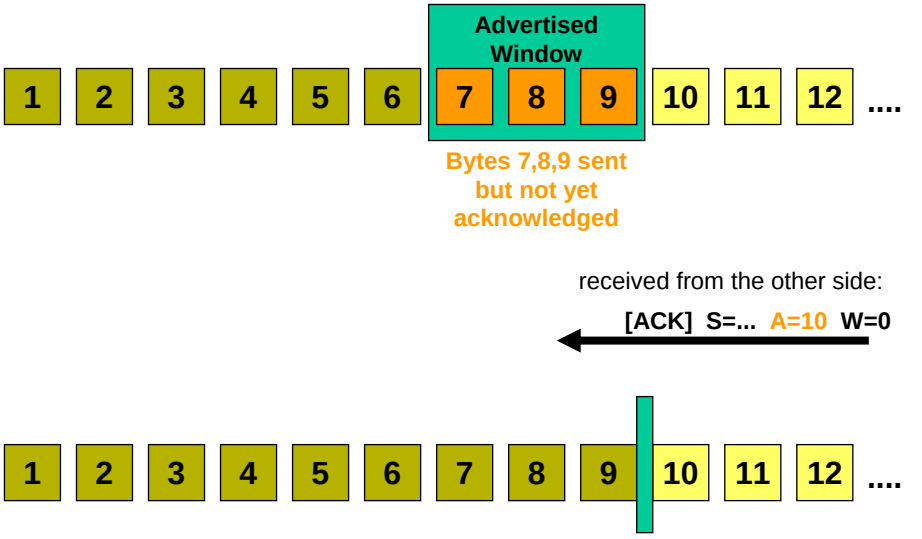
- **During the transmission the sliding window moves from left to right, as the receiver acknowledges data**
- **The relative motion of the two ends of the window *open* or *closes* the window**
  - the window closes when data - already sent - is acknowledged (the left edge advances to the right)
  -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

Closing the Sliding Window

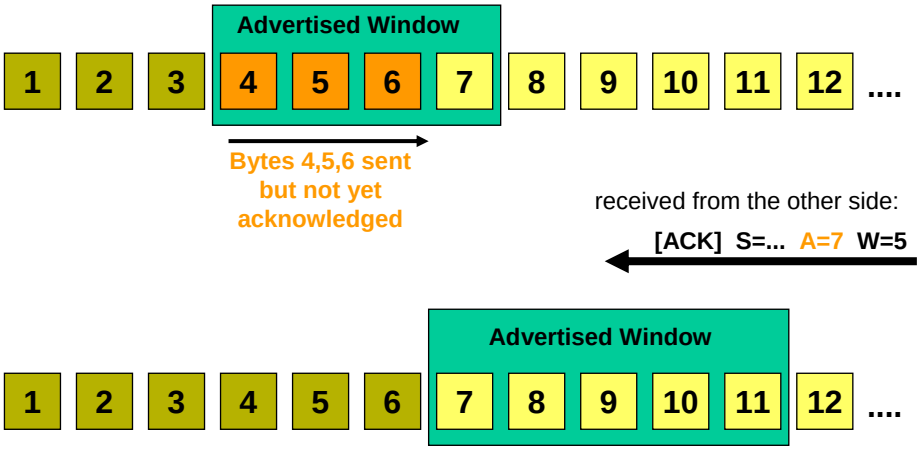


Now the sender may send bytes 7, 8, 9. The receiver didn't open the window (W=3, right edge remains constant) because of congestion. However, the remaining three bytes inside the window are already granted, so the receiver cannot move the right edge leftwards.

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.

Sliding Window

- **The right edge of the window must not move leftward !**
  - however, TCP must be able to cope with a peer doing this
  - called *shrinking* window
- **The left edge cannot move leftward because it is determined by the acknowledgement number of the receiver**
  - only a duplicate Ack would imply to move the left edge leftwards, but duplicate Acks are discarded

TCP Enhancements

- **So far, only basic TCP procedures have been mentioned**
- **TCP's development still continues; it has been already enhanced with additional functions which are essential for operation of TCP sessions in today's IP networks:**
  - Slow Start and Congestion Avoidance Mechanism
  - Fast Retransmit and Fast Recovery Mechanism
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - ....

TCP Enhancements

- **Slow Start and Congestion Avoidance Mechanism:**
  - controls the rate of packets which are put into a network (sender-controlled flow control as add on to the receiver-controlled flow control based on the window field)
- **Fast Retransmit and Fast Recovery Mechanism:**
  - to avoid waiting for the timeout in case of retransmission and to avoid slow start after a fast retransmission

Delayed Acknowledgements

- **Immediate acknowledgements may cause an unnecessary amount of data transmissions**
  - normally, an acknowledgement would be send immediately after the receiving of data
  - but in interactive applications, the send-buffer at the receiver side gets filled by the application soon after an acknowledgement has been send (e.g. Telnet echoes)
- **In order to support piggy-backed acknowledgements (i.e. acks combined with user data), the TCP stack waits 200 ms before sending the acknowledgement**
  - during this time, the application might also have data to send

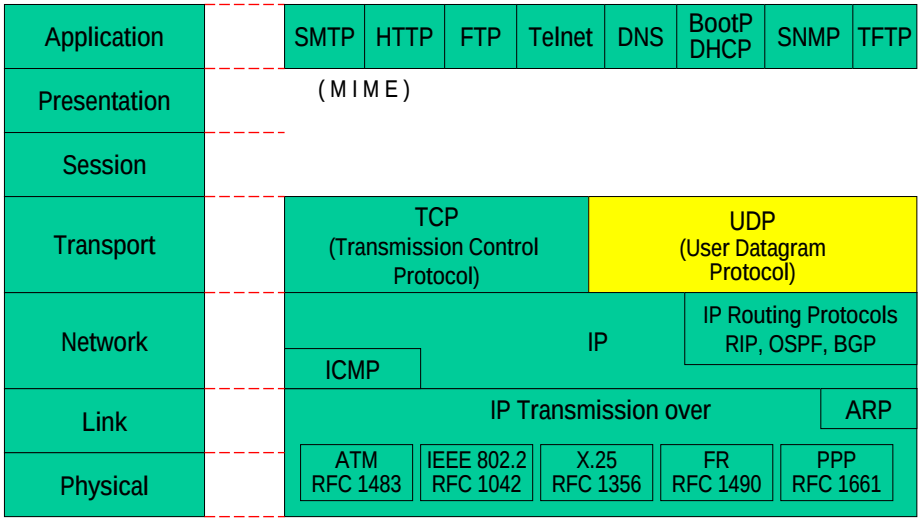
The Nagle Algorithm

- **Several applications send only very small segments - "tinygrams"**
  - e.g. Telnet or Rlogin where each key-press generates 41 bytes to be transmitted: 20 bytes IP header, 20 bytes TCP header and only 1 byte of data (!)
- **Frequent tinygrams can lead to congestion at slow WAN connections**
- **Nagle Algorithm:**
  - When a TCP connection waits for an acknowledgement, small segments must not be sent until the acknowledgement arrives
  - In the meanwhile, TCP can collect small amounts of application data and send them in a single segment when the acknowledgement arrives

Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

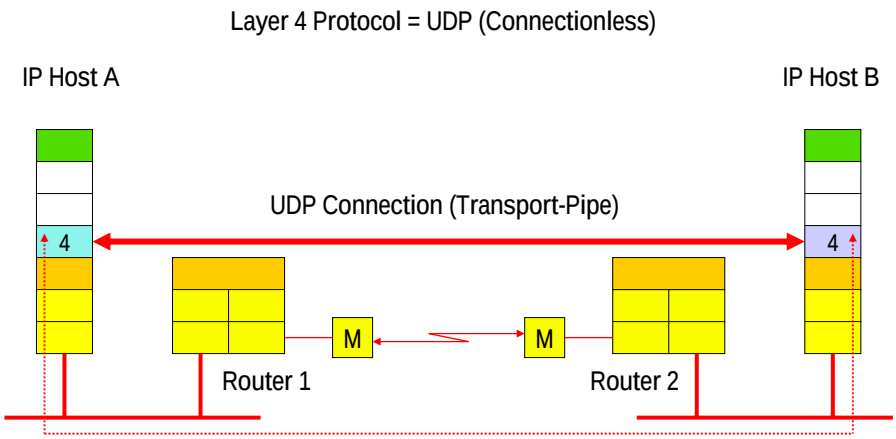
Transport Layer Protocols



UDP (User Datagram Protocol, RFC 768)

- 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)
- Less complex than TCP, easier to implement

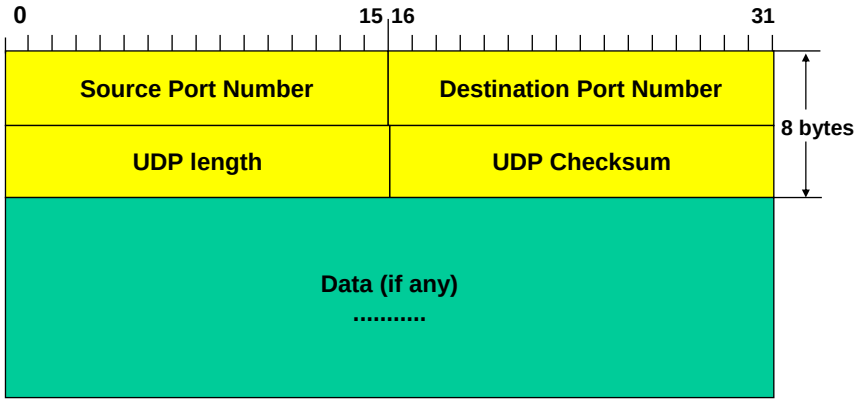
UDP and OSI Transport Layer 4



UDP Usage

- **UDP is used**
  - where the overhead of a connection oriented service is undesirable
    - e.g. for short DNS request/reply
  - where the implementation has to be small
    - e.g. BootP, TFTP, DHCP, SNMP
  - where retransmission of lost segments makes no sense
    - Voice over IP
    - note: digitized voice is critical concerning delay but not against loss
      - Voice is encapsulated in RTP (Real-time Transport Protocol)
      - RTP is encapsulated in UDP
      - RTCP (RTP Control Protocol) propagates control information in the opposite direction
      - RTCP is encapsulated in UDP

UDP Header



UDP Header Entries

- **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

Important UDP Port Numbers

7	Echo
53	DOMAIN, Domain Name Server
67	BOOTPS, Bootstrap Protocol Server
68	BOOTPC, Bootstrap Protocol Client
69	TFTP, Trivial File Transfer Protocol
79	Finger
111	SUN RPC, Sun Remote Procedure Call
137	NetBIOS Name Service
138	NetBIOS Datagram Service
161	SNMP, Simple Network Management Protocol
162	SNMP Trap
322	RTSP (Real Time Streaming Protocol) Server
520	RIP
5060	SIP (VoIP Signaling)
xxxx	RTP (Real-time Transport Protocol)
xxxx+1	RTCP (RTP Control Protocol)

Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

How to Improve TCP Performance?

- **Problem:**
  - in case of packet loss sender can use the window given by the receiver but when window becomes closed the sender must wait until retransmission timer times out
  - that means during that time sender cannot use offered bandwidth of the network
  - -> TCP performance degradation
- **Assumption:**
  - packet loss in today´s networks are mainly caused by congestion but not by bit errors on physical lines
    - optical transmission
    - digital transmission

Congestion

- **Problem of bottle-neck inside of a network**
  - Some intermediate router must queue packets
  - Queue overflow -> retransmission -> more overflow !
  - Can't be solved by traditional receiver-imposed flow control (using the window field)
- **Ideal case: rate at which new segments are injected into the network = acknowledgment-rate of the other end**
  - Requires a sensitive algorithm to catch the equilibrium point between high data throughput and packet dropping due to queue overflow:
    Van Jacobson's Slow Start and Congestion Avoidance

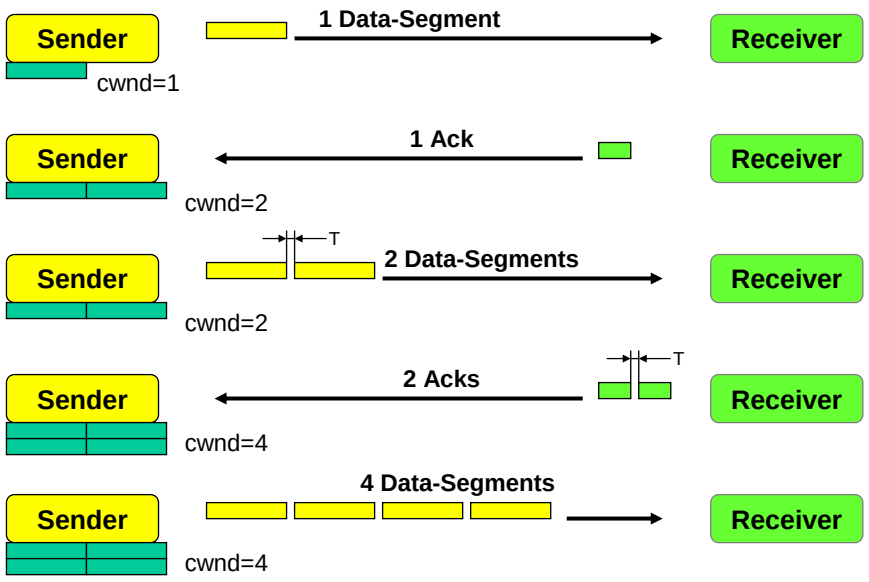
Slow Start 1

- **Duplicate Ack can be used for catching this equilibrium**
  - That is the base for Slow Start and Congestion Avoidance
- **Slow start (and congestion avoidance) is mandatory for today's TCP implementations !**
- **Slow start requires TCP to maintain an additional window: the "congestion window" (cwnd)**
  - Rule: *The sender may transmit up to the minimum of the congestion window and the advertised window*

Slow Start 2

- **When a new TCP connection is established, the congestion window is initialized to one segment**
  - Using the maximum segment size (MSS) of the receiver (learned via a TCP option field)
    - Note: RFC 2414 suggests increasing the initial congestion window to 2-4 segments
- **Each time the sender receives an acknowledgment, the congestion window is increased by one segment size**
- **This way, the segment send rate doubles every round trip time until congestion occurs; then the sender has to slow down again**

Slow Start 3



Congestion

- **Slow start encounters congestion, when**
  - routers connect pipes with different bandwidth
  - routers combine several input pipes and the bandwidth of the output pipe is less than the sum of the input bandwidths
  - and hence TCP segment(s) is (are) dropped by a router
- **Congestion can be detected by the sender through timeouts or duplicate acknowledgements**
- **Slow start reduces its sending rate with the help of another algorithm, called "Congestion Avoidance"**

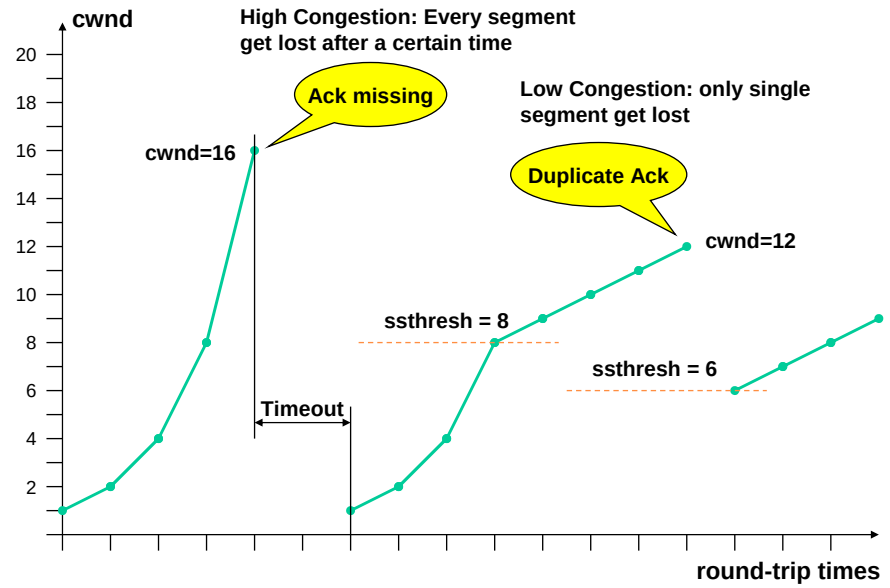
Congestion Avoidance 1

- **Slow start with congestion avoidance is a sender-imposed flow control**
  - Congestion Avoidance requires TCP to maintain a variable called "slow start threshold" (ssthresh)
  - Initially, ssthresh is set to TCPs maximum possible MSS (i.e. 65,535 octets)
- **On detection of congestion, ssthresh is set to half the current window size**
  - here, window size means: minimum of advertised window and congestion window (but at least 2 segments)
  - Note: ssthresh marks a safe window size because congestion occurred at a window size of 2 x ssthresh

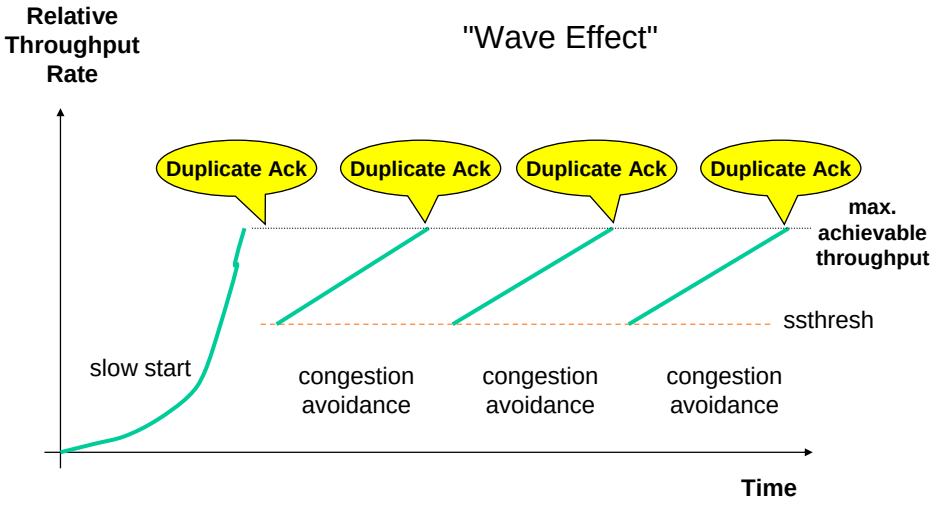
Congestion Avoidance 2

- **If the congestion is indicated by**
  - a timeout:
    - cwnd is set to 1 -> forcing slow start again
  - a duplicate ack:
    - cwnd is set to ssthresh (= 1/2 current window size)
- **cwnd ≤ ssthresh:**
  - slow start, doubling cwnd every round-trip time
  - exponential growth of cwnd
- **cwnd > ssthresh:**
  - congestion avoidance, cwnd is incremented by MSS × MSS / cwnd every time an ack is received
  - linear growth of cwnd

Slow Start and Congestion Avoidance



Long Term View of TCP Throughput



Limitation of ARQ Protocols

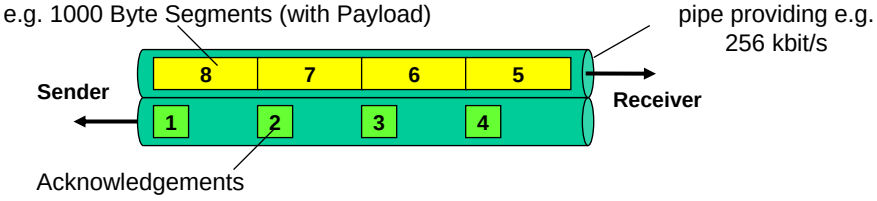
- **The performance of any connection-oriented protocol with error-recovery (ARQ) is limited by bandwidth and delay by nature!**
  - Optimum can be achieved by using Continous RQ with sliding window technique where the window is large enough to avoid stopping of sending
  - Large enough means to cover the time of the serialization and propagation delays
  - Note: senders and receivers window size maybe also be limited because of memory constraints

Limitation of ARQ Protocols

- **The sender's window must be big enough so that the sender can fully utilize the channel volume**
- **Channel volume is increased**
  - by delays caused by buffers
  - limited signal speed
  - Bandwidth
- **The channel volume can be expressed by the Delay-Bandwidth Product**

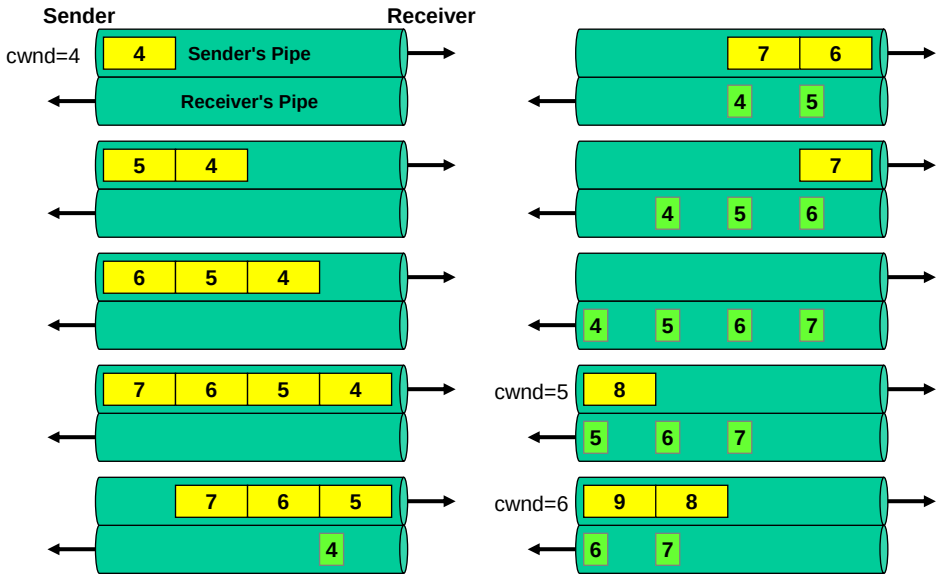
The Delay-Bandwidth Product

- **In order to fill the "pipe" between sender and receiver with data packets:**
  - the window-size advertised by the receiver should be not smaller than the Delay-Bandwidth Product of the pipe
  - window size ≥ capacity of pipe (bits)
    = bandwidth (bits/sec) x round-trip time (sec)

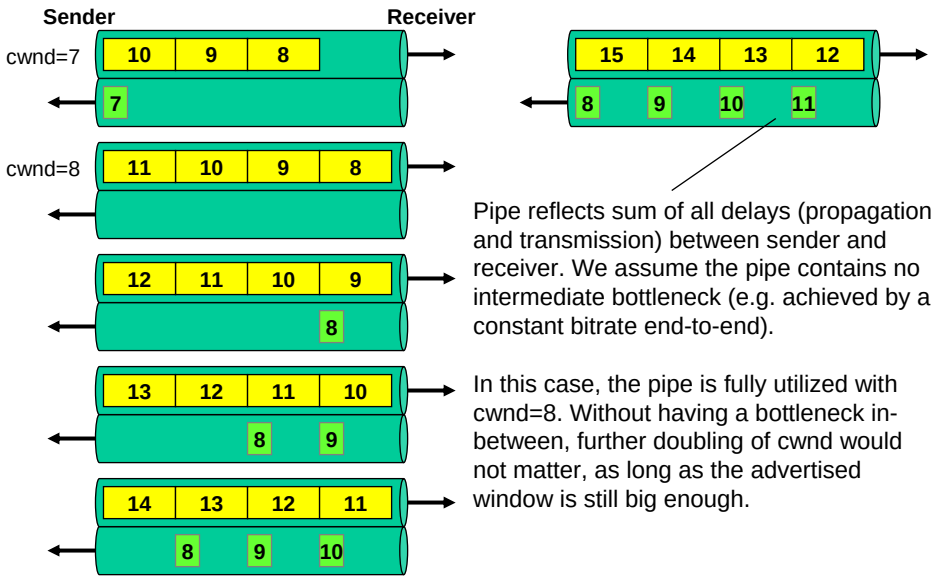


Example: For a given RTT = 0.25 s (Round-trip time = elapsed time between the sending of segment n and the receiving of the corresponding ack n) the minimum window size is 256 kbit/s x 0.25 sec = 64 kbit. Using a segment size of 1 kB, the sender can transmit 8 segments before waiting for any acknowledgement.

Filling the Pipe (1/2)



Filling the Pipe (2/2)

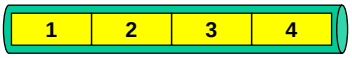


Pipe reflects sum of all delays (propagation and transmission) between sender and receiver. We assume the pipe contains no intermediate bottleneck (e.g. achieved by a constant bitrate end-to-end).

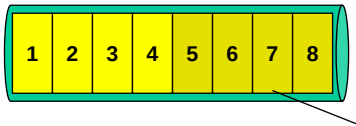
In this case, the pipe is fully utilized with cwnd=8. Without having a bottleneck in-between, further doubling of cwnd would not matter, as long as the advertised window is still big enough.

Delay-Bandwidth Relations

Given pipe with given RTT and bandwidth:

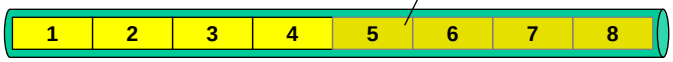


1) Doubled bandwidth:



Additional capacity

2) Doubled RTT:



Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

Receiver Requirements

- **Initially, TCP could detect packet loss only by expiration of the retransmission timer**
  - receiver stops sending Acks until the sender retransmits all missing segments
  - causes long delays
- **Fast Retransmit requires a receiver to send an immediate duplicate acknowledgement in order to notify the sender which segments are (still) expected by the receiver**

Sender Aspects and Fast Retransmit

- **When should retransmission occur?**
  - Note: the receiver will also send duplicate acknowledgements when segments are arriving in the wrong order
  - Typically reordering problems cause one or two duplicate acks on the average
- **Therefore, TCP sender awaits two duplicate acknowledgements and starts retransmission after the third duplicate acknowledgement**
  - that mechanism is called “Fast Retransmit”

Fast Recovery 1

- **"Fast Retransmit" automatically commences "Fast Recovery" in order to rapidly repair single packet loss**
- **"Fast Recovery" mechanism:**
  - ssthresh is set to half the current window size
  - cwnd is set to ssthresh plus 3 times the segment size
    - Remember:
      Fast Retransmit waits for 3 duplicate acks; from this can be concluded that the receiver must have received 3 segments already

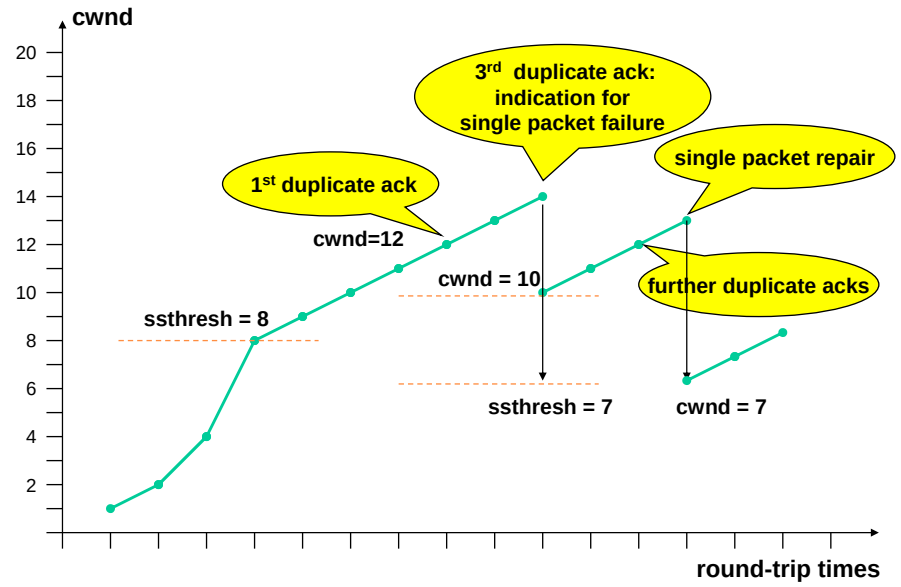
Fast Recovery 2

- **"Fast Recovery" mechanism cont.**
  - congestion avoidance, but not slow start is performed
    - Remember:
      The receiver can only generate a duplicate ack when another segment is received. That is: there are still segments flowing through the network! Slow start would reduce this flow abruptly!
  - For each additional duplicate ack, the sender increases cwnd by 1 segment size
  - Upon receiving an ack that acknowledges new data
    - cwnd is set to ssthresh
    - sender resumes normal congestion avoidance mode

Fast Recovery 3

- **Fast Recovery**
  - allows the sender to continue to maintain the ack-clocked data rate for new data while the packet loss repair is being undertaken
    - note: if send window would be closed abruptly the synchronization via duplicate acks would be lost
  - still the single packet loss indicates congestion and back off to normal congestion avoidance mode must be done

Fast Retransmit and Fast Recovery



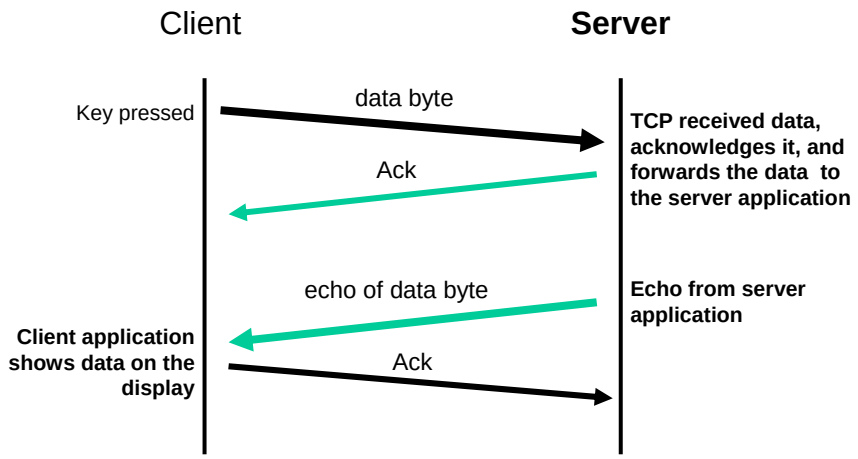
Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

Interactive Traffic

- **Immediate acknowledgements may cause an unnecessary amount of data transmissions**
- **Normally, an acknowledgement would be send immediately after the receiving of data**
- **But in interactive applications, the send-buffer at the receiver side gets filled by the application soon after an acknowledgement has been send (e.g. Telnet echoes)**

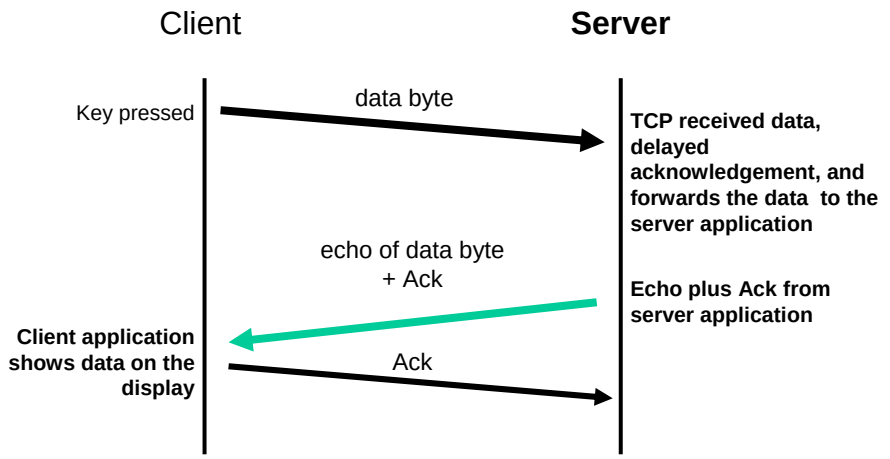
Interactive Traffic



Delayed Acknowledgements

- In order to support piggy-backed acknowledgements (i.e. Acks combined with user data), the TCP stack waits 200 ms before sending the acknowledgement
- During this time, the receiving application might also have data to send
- That is: 50% less (interactive!) traffic using delayed acknowledgements

Interactive Traffic with Delayed ACK



Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

The Nagle Algorithm

- **Several applications send only very small segments - "tinygrams"**
  - E.g. Telnet or Rlogin where each key-press generates 41 bytes to be transmitted: 20 bytes IP header, 20 bytes TCP header and only 1 byte of data (!)
- **Frequent tinygrams can lead to congestion at slow WAN connections**

The Nagle Algorithm

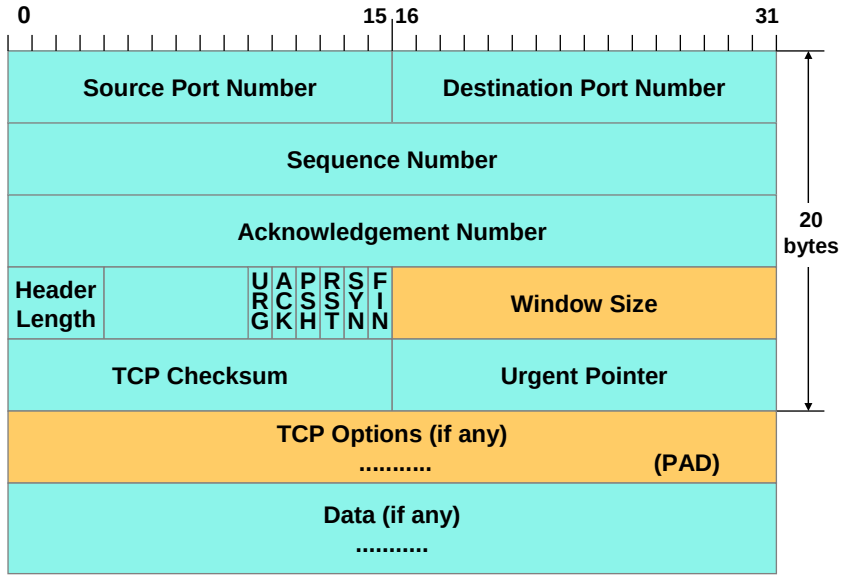
Nagle Algorithm:

- **When a TCP connection waits for an acknowledgement, small segments must not be sent until the acknowledgement arrives**
- **In the meanwhile, TCP can collect small amounts of application data and send them in a single segment when the acknowledgement arrives**

Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

TCP Header Window Field



TCP Options

- **Window-scale option**
  - a maximum segment size of 65,535 octets is inefficient for high delay-bandwidth paths
  - the window-scale option allows the advertised window size to be left-shifted (i.e. multiplication by 2)
  - enables a maximum window size of 2^30 octets !
  - negotiated during connection establishment
- **SACK (Selective Acknowledgement)**
  - if the SACK-permitted option is set during connection establishment, the receiver may selectively acknowledge already received data even if there is a gap in the TCP stream (Ack-based synchronization maintained)

Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

The Limits of Interpreting Symptoms Only

- **Slow start and congestion avoidance try to maximize the traffic throughput without inclusion of network information**
  - Host-based congestion control
  - Original IP idea: "Keep the network simple !"
  - Slow start and songestion avoidance suspects congestion only by observing symptoms of the network
- **Further improvements require an active inclusion of the intermediate network**
- **Led to the introduction of an Explicit Congestion Notification, which requires the help from routers that are expecting congestion**

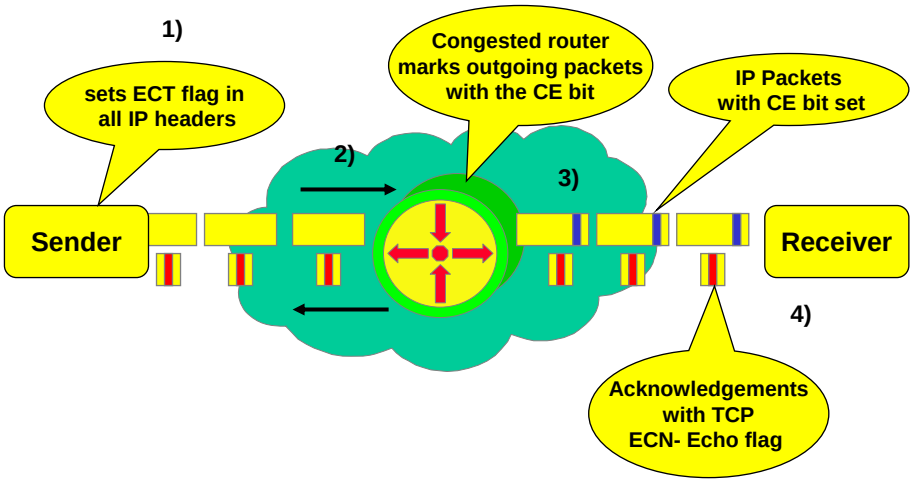
Explicit Congestion Notification (ECN)

- **During TCP connection establishment, the ECN capability is negotiated**
  - ECN utilizes bit 6 and 7 of the IPv4 TOS field
    - ECT (Explicit Congestion Notification Transport System)
    - CE (Congestion Experienced)
  - Additionally ECN requires the two TCP options
    - "ECN-Echo" and "Congestion Window Reduced" (CWR)
- **Then the sender**
  - sets the ECT bit in the IP header of all datagram it sends
- **When routers experience congestion**
  - they may mark the IP header of such packets with an explicit CE bit flag

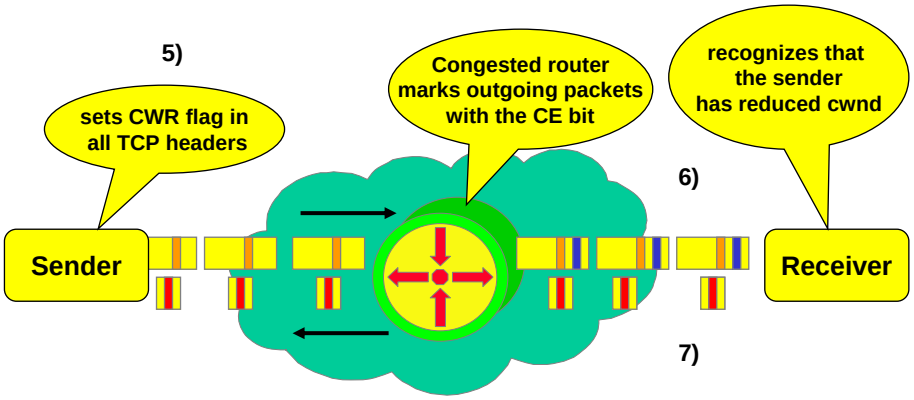
Explicit Congestion Notification (ECN)

- **The receiver detects the CE flag**
  - and sets the TCP ECN-Echo flag in its acknowledgement segment
- **If the sender receives this acknowledgement segment with the ECN-echo flag set,**
  - the sender reduces its congestion window (-> congestion avoidance)
  - the sender sets the TCP CWR flag in its next segment in order to notify the receiver that the sender has reacted upon the congestion
- **Main advantage:**
  - the sender does not have to wait for three duplicate acks to detect the congestion

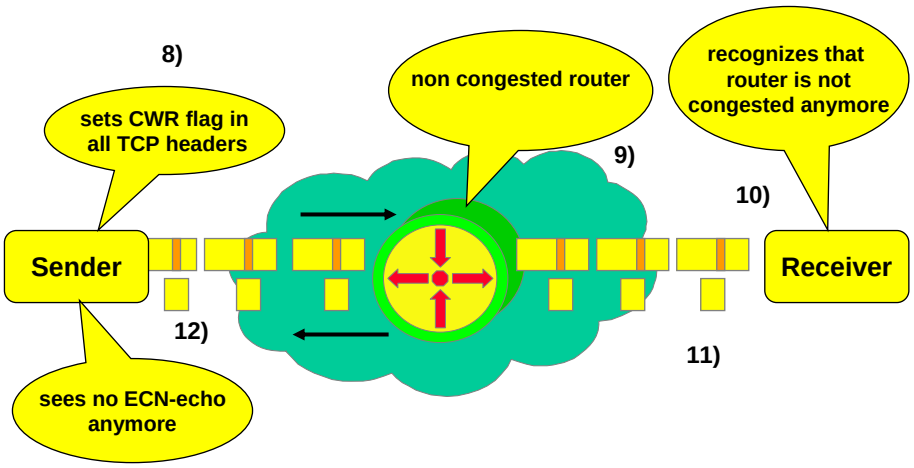
Explicit Congestion Notification (ECN) 1



Explicit Congestion Notification (ECN) 2



Explicit Congestion Notification (ECN) 3



Agenda

- **TCP Fundamentals**
- **UDP**
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - Delayed Acknowledgements
  - The Nagle Algorithm
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **RFC Collection**

RFCs

- **0761 - TCP**
- **0813 - Window and Acknowledgement Strategy in TCP**
- **0879 - The TCP Maximum Segment Size**
- **0896 - Congestion Control in TCP/IP Internetworks**
- **1072 - TCP Extension for Long-Delay Paths**
- **1106 - TCP Big Window and Nak Options**
- **1110 - Problems with Big Window**
- **1122 - Requirements for Internet Hosts -- Com. Layer**
- **1185 - TCP Extension for High-Speed Paths**
- **1323 - High Performance Extensions (Window Scale)**

RFCs

- 2001 - Slow Start and Congestion Avoidance (Obsolete)
- 2018 - TCP Selective Acknowledgement (SACK)
- 2147 - TCP and UDP over IPv6 Jumbograms
- 2414 - Increasing TCP's Initial Window
- 2581 - TCP Slow Start and Congestion Avoidance (Current)
- 2873 - TCP Processing of the IPv4 Precedence Field
- 3168 - TCP Explicit Congestion Notification (ECN)