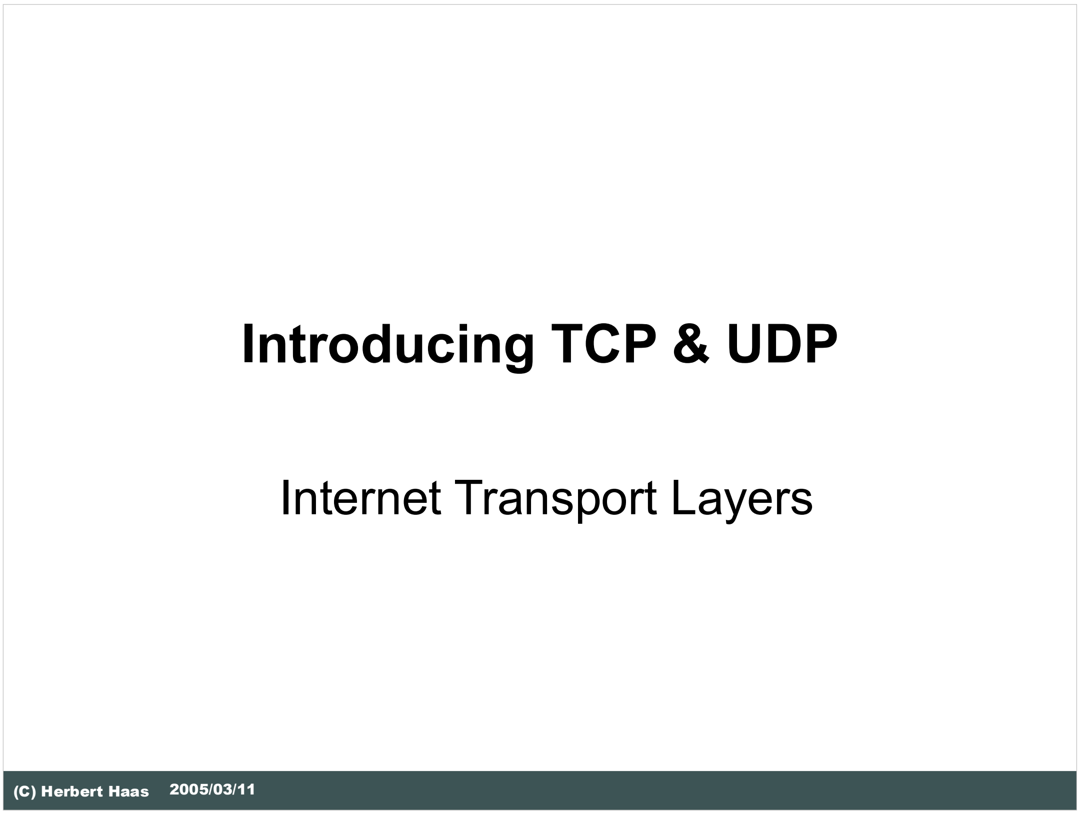
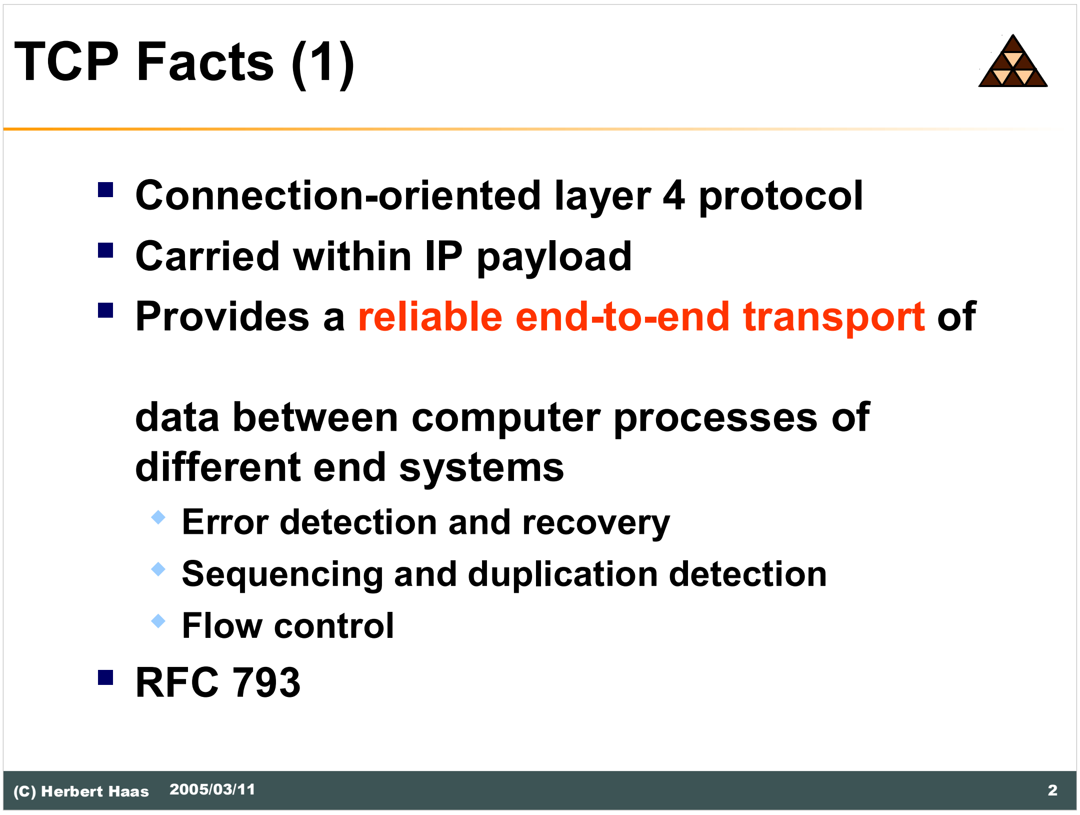


Introducing TCP & UDP

Internet Transport Layers

(C) Herbert Haas 2005/03/11



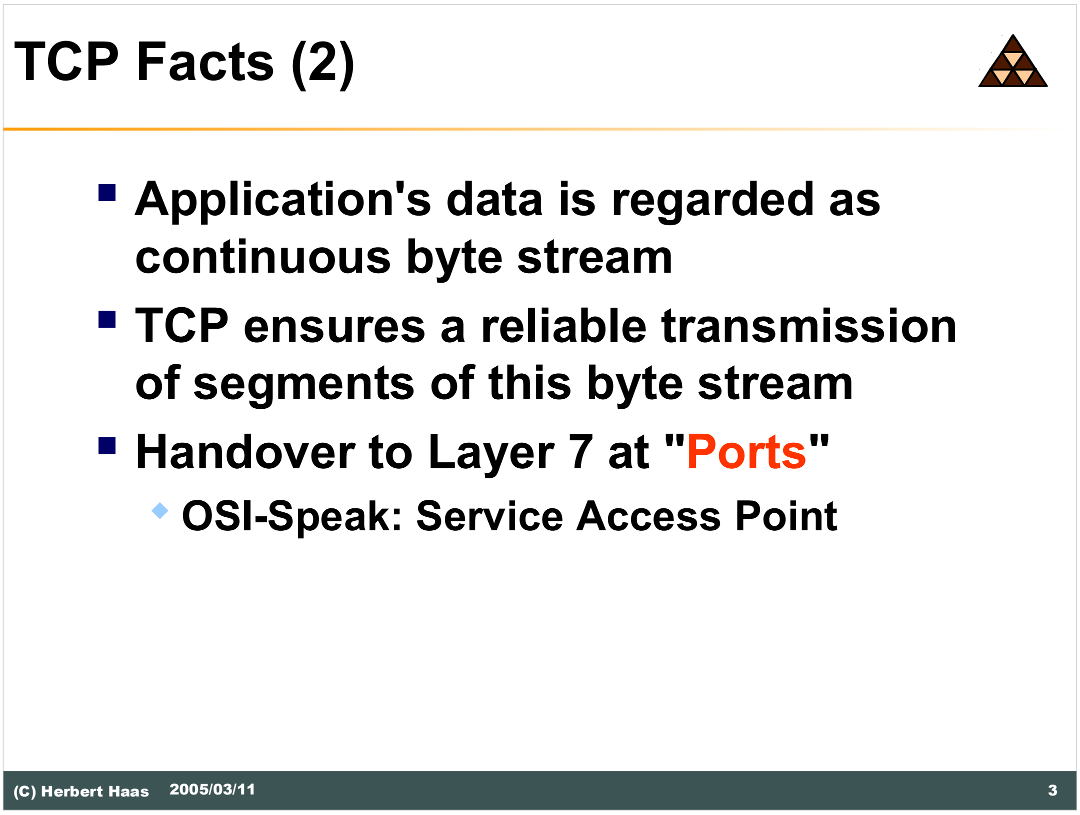
TCP Facts (1)

- 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

(C) Herbert Haas 2005/03/11 2

In this Chapter we talk about **TCP**. TCP is a connection-oriented layer 4 protocol and only works between the hosts. It synchronizes (connects) the hosts with each other via the “3-Way-Handshake” before the real transmission begins. After this a reliable end-to-end transmission is established. TCP was standardized in September 1981 in RFC 793. (Remember: IP was standardized in September 1981 too, RFC 791). TCP is always used with IP and it also protects the IP packet as its checksum spans over (almost) the whole IP packet.

TCP provides error recovery, flow control and sequencing. The most important thing with TCP is the **Port-Number**, we will discus later.

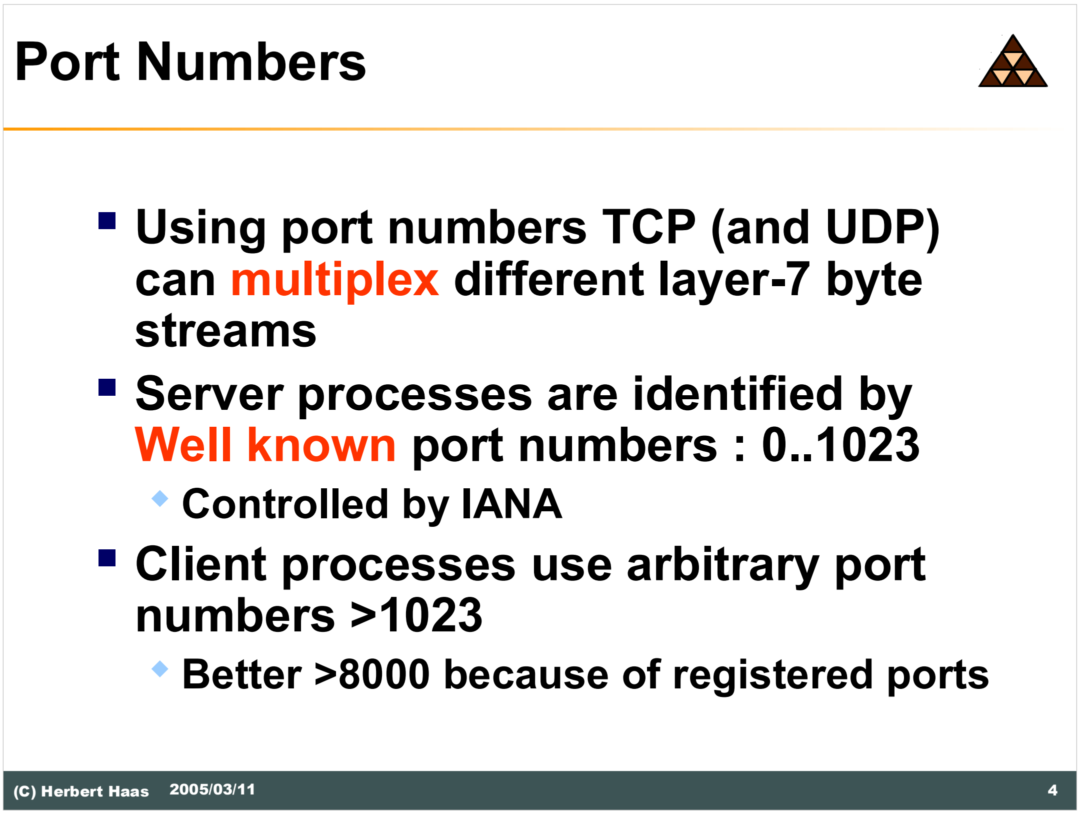


TCP Facts (2)

- 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

Every IP packet which is sent along with TCP will be acknowledgment (error recovery). From the TCP perspective we call each packet a segment.

TCP 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 provides its service to higher layer through ports (OSI: Service Access Points).

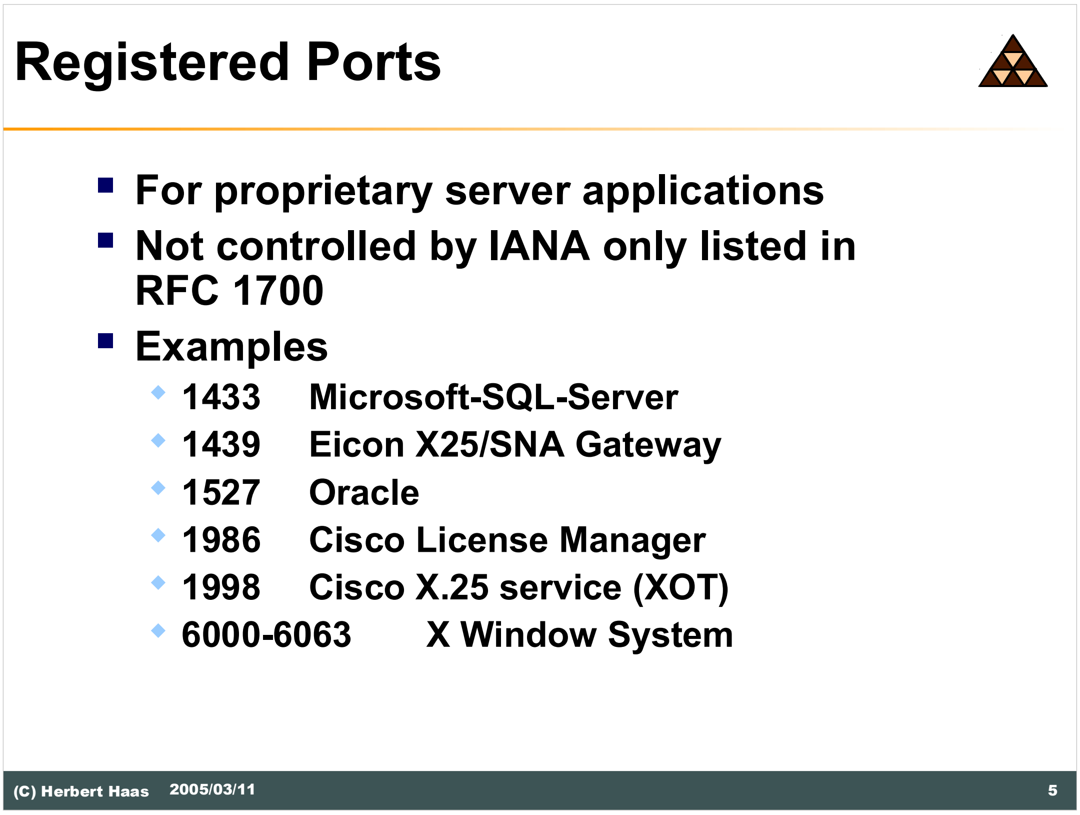


Port Numbers

- 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

(C) Herbert Haas 2005/03/11 4

Each communicating computer process is assigned a locally unique port number. Using port numbers TCP can service multiple processes such as a web browser or an E-Mail client simultaneously through a single IP address. In summary TCP works like a stream multiplexer and demultiplexer.



Registered Ports

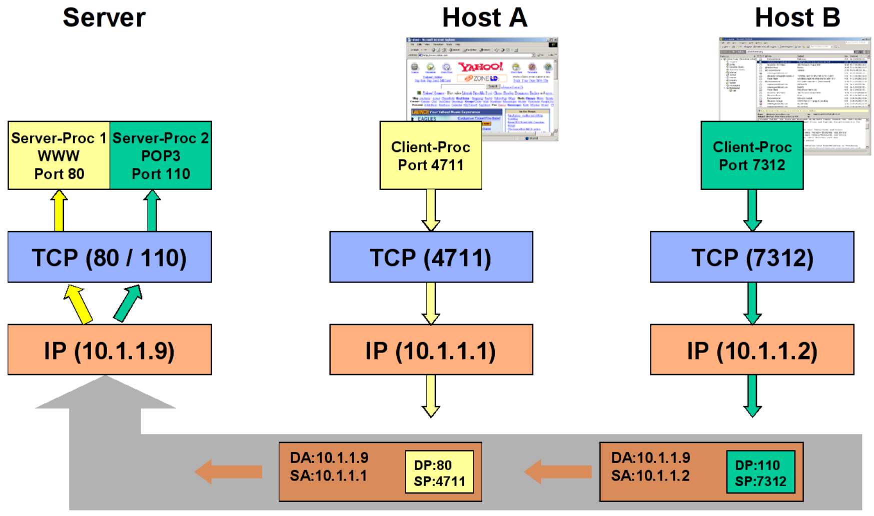
- For proprietary server applications
- Not controlled by IANA only listed in RFC 1700
- Examples

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

Only the **well known ports** are reserved for common applications and services, such as Telnet, WWW, FTP etc. They are in the range from 0 to 1023. These are controlled by the Internet Assigned Numbers Authority (IANA).

There are also many **registered ports** which start at 1024 (e.g. Lotus Notes, Cisco XOT, Oracle, license managers etc.). They are not controlled by the IANA, only listed in RFC1700.

TCP Communications



Server

Host A

Host B

Server-Proc 1
WWW
Port 80

Server-Proc 2
POP3
Port 110

Client-Proc
Port 4711

Client-Proc
Port 7312

TCP (80 / 110)

TCP (4711)

TCP (7312)

IP (10.1.1.9)

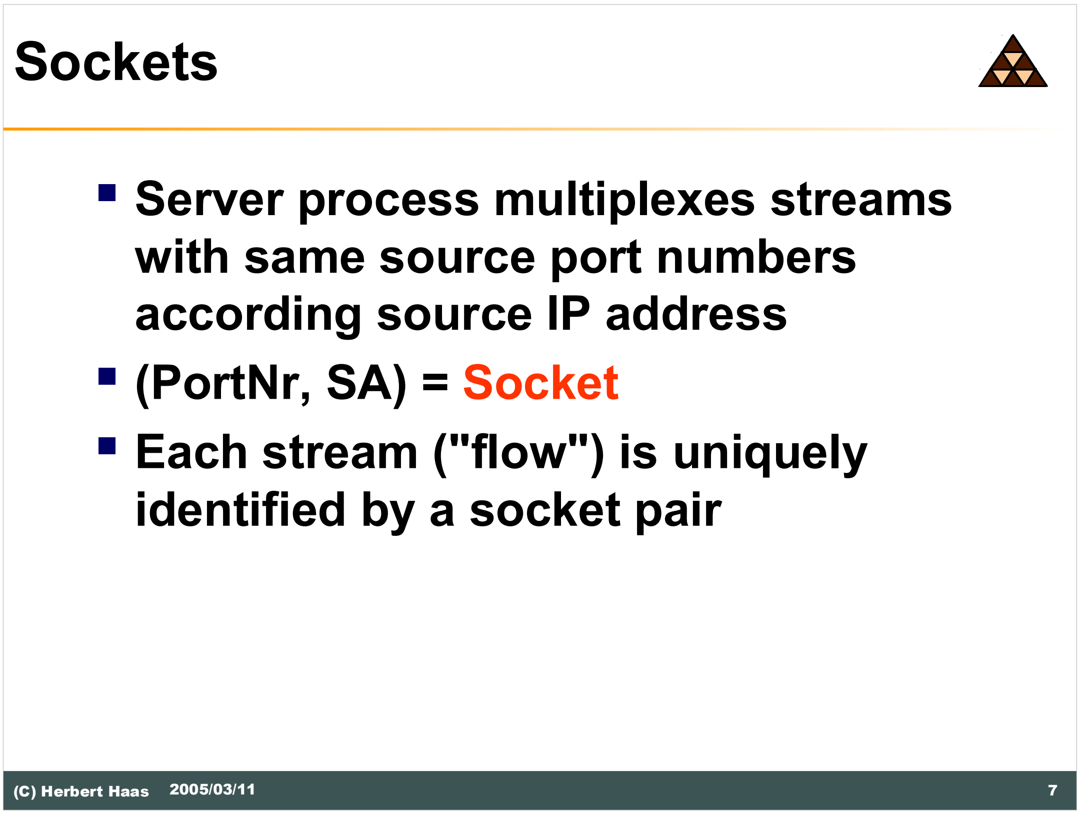
IP (10.1.1.1)

IP (10.1.1.2)

DA:10.1.1.9
SA:10.1.1.1
DP:80
SP:4711

DA:10.1.1.9
SA:10.1.1.2
DP:110
SP:7312

The 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. For example: Host B runs a Mail-Program (POP3) and the client application uses the source port 7312. The packet is send to the Server with a destination-port of 110. Now the Server knows Host B makes a Mail-Check over POP3.



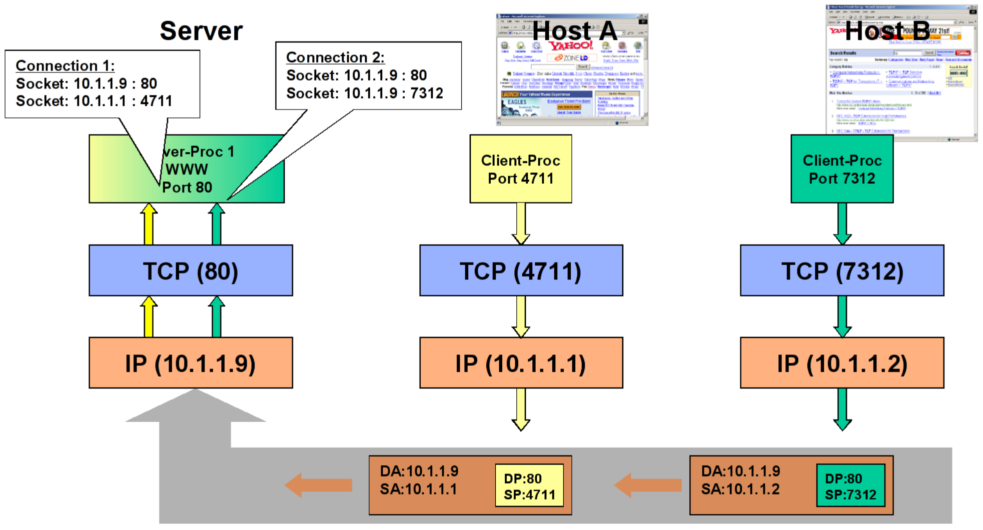
Sockets

- 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

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**“.

For example: 10.1.1.2:80 [IP-Address : Port-Number]

TCP Communications



Server

Connection 1:
Socket: 10.1.1.9 : 80
Socket: 10.1.1.1 : 4711

Connection 2:
Socket: 10.1.1.9 : 80
Socket: 10.1.1.9 : 7312

ver-Proc 1
WWW
Port 80

TCP (80)

IP (10.1.1.9)

Host A

Client-Proc
Port 4711

TCP (4711)

IP (10.1.1.1)

Host B

Client-Proc
Port 7312

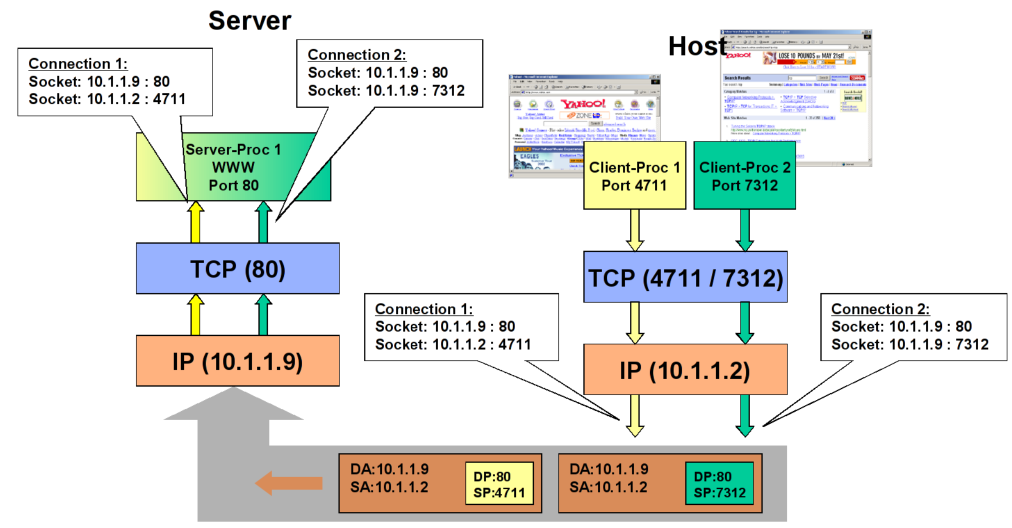
TCP (7312)

IP (10.1.1.2)

DA:10.1.1.9
SA:10.1.1.1
DP:80
SP:4711

DA:10.1.1.9
SA:10.1.1.2
DP:80
SP:7312

TCP Communications



Server

Connection 1:
Socket: 10.1.1.9 : 80
Socket: 10.1.1.2 : 4711

Connection 2:
Socket: 10.1.1.9 : 80
Socket: 10.1.1.9 : 7312

Server-Proc 1
WWW
Port 80

TCP (80)

IP (10.1.1.9)

Host

Client-Proc 1
Port 4711

Client-Proc 2
Port 7312

TCP (4711 / 7312)

IP (10.1.1.2)

Connection 1:
Socket: 10.1.1.9 : 80
Socket: 10.1.1.2 : 4711

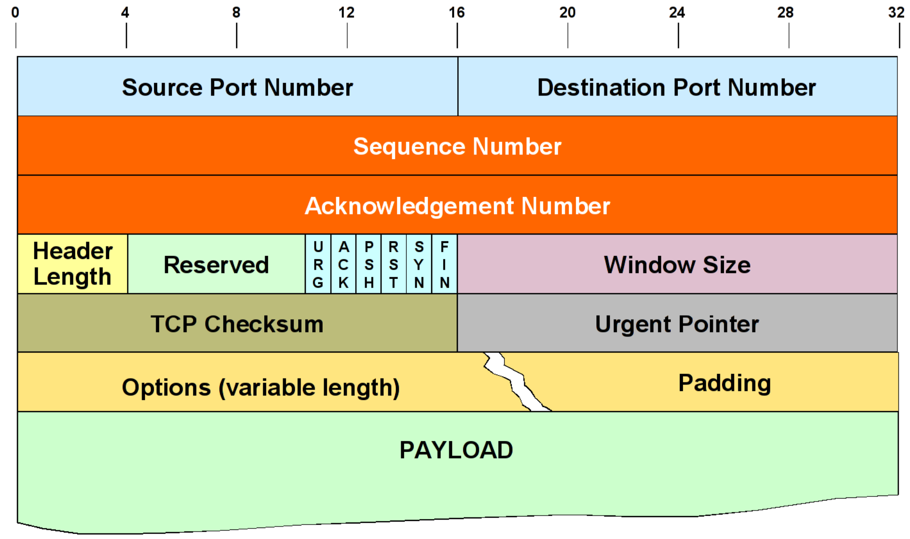
Connection 2:
Socket: 10.1.1.9 : 80
Socket: 10.1.1.9 : 7312

DA:10.1.1.9
SA:10.1.1.2
DP:80
SP:4711

DA:10.1.1.9
SA:10.1.1.2
DP:80
SP:7312

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.

TCP Header



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

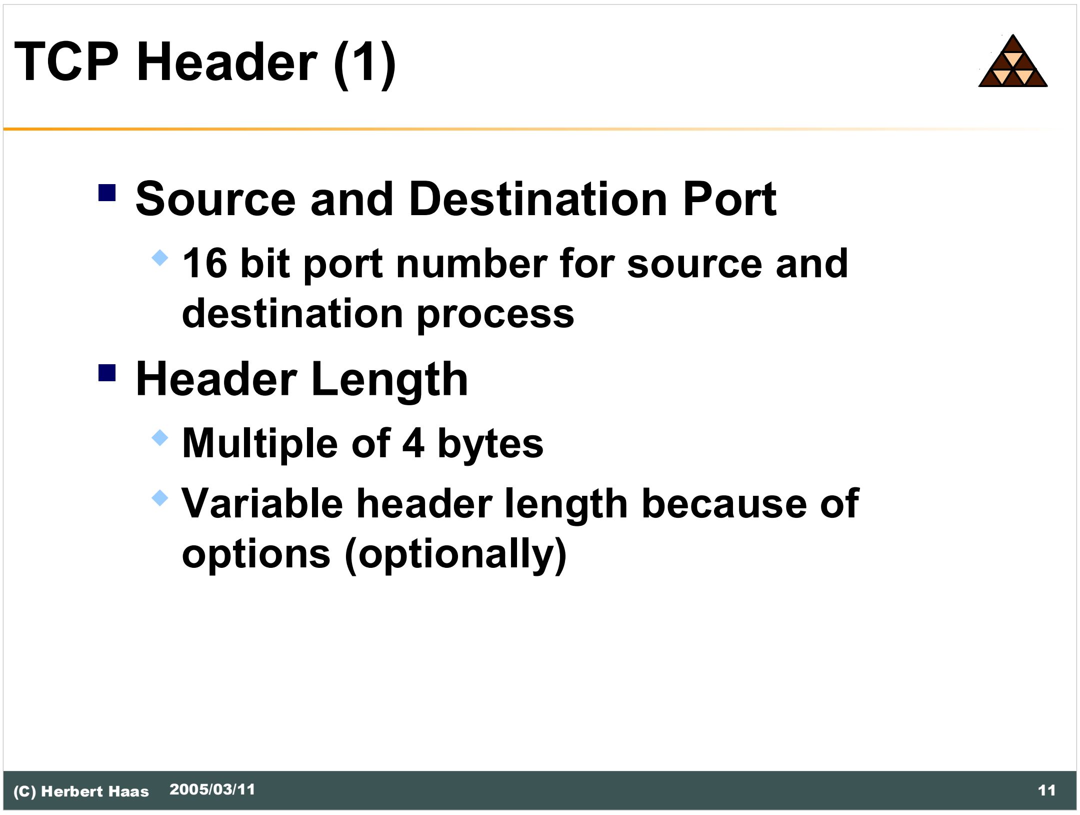
Options (variable length) | Padding

PAYLOAD

The picture above shows the 20 byte TCP header plus optional options. Remember that the IP header has also 20 bytes, so the total sum of overhead per TCP/IP packet is 40 bytes.

It is important to know these header fields, at least the most important parts:

1) The Port numbers – most important, to address applications
2) The Sequence numbers (SQNR and Ack) – used
3) The Window field – used for flow control
4) The flags SYN, ACK, RST, and FIN – for session control

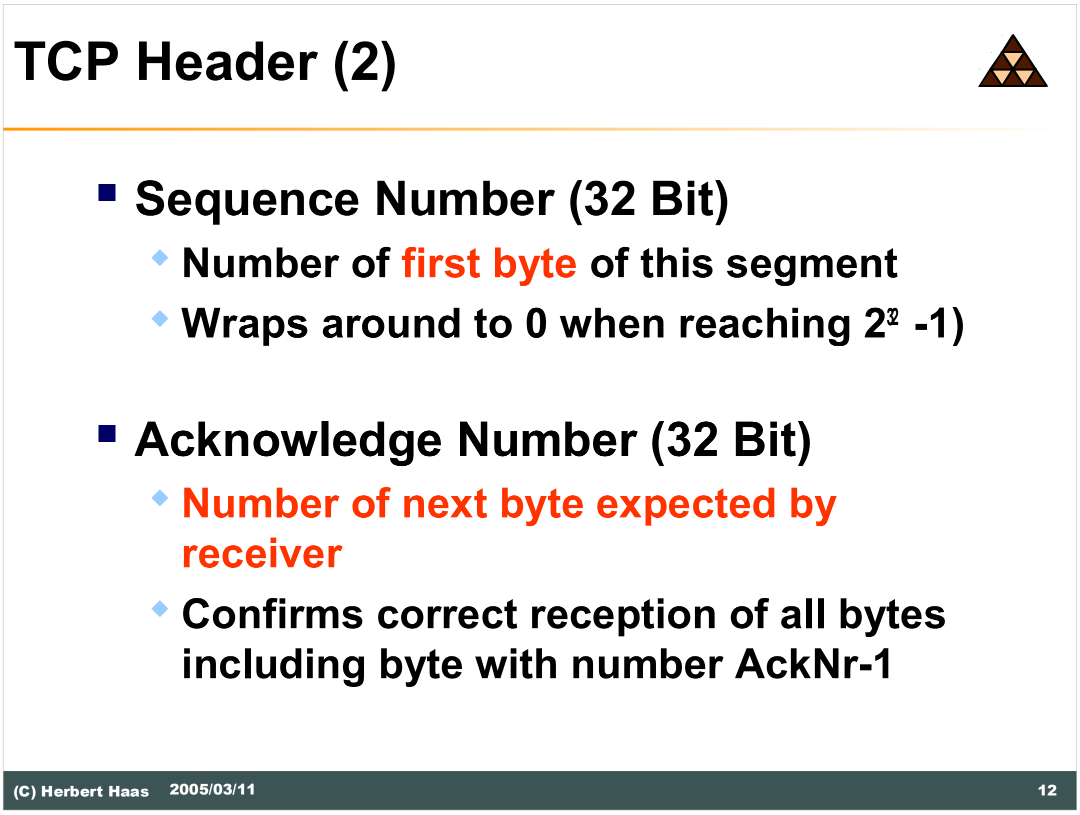


TCP Header (1)

- 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)

The **Source** and **Destination Port** fields are 16 bits and used by the application.

The **Header Length** indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

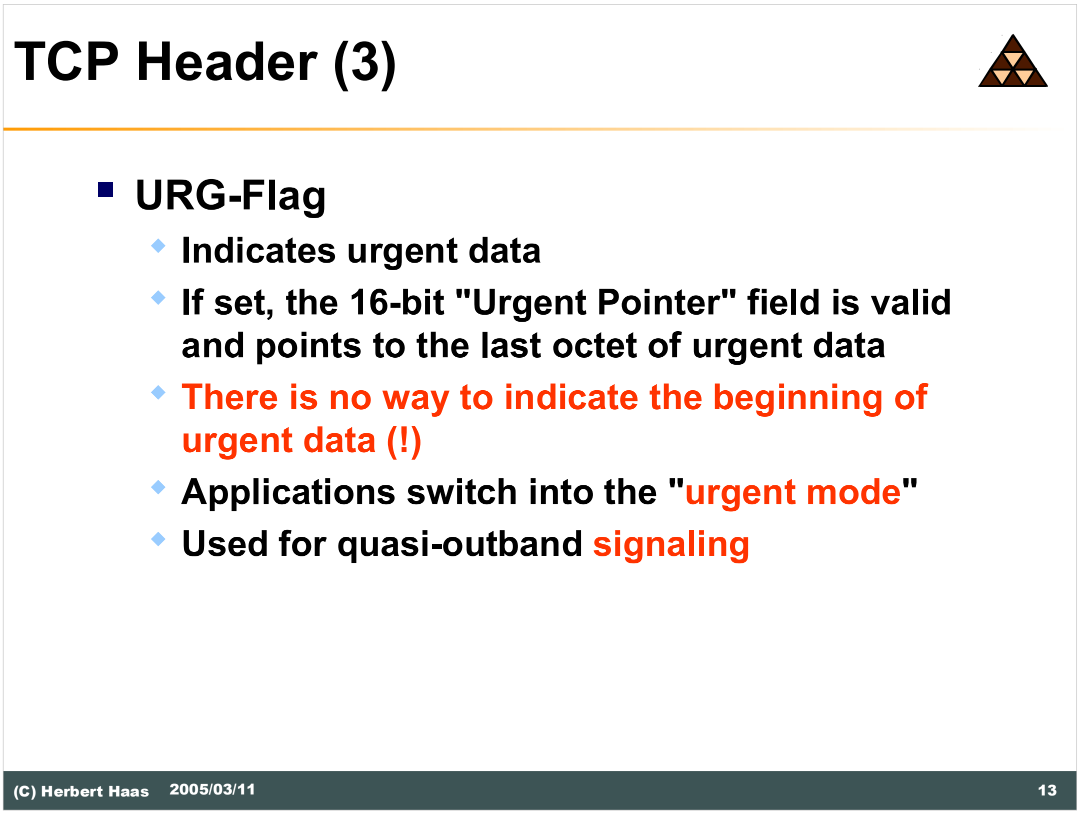


TCP Header (2)

- 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

Sequence Number: 32 bit. Number of the first byte of this segment. If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.

Acknowledge Number: 32 bit. If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.



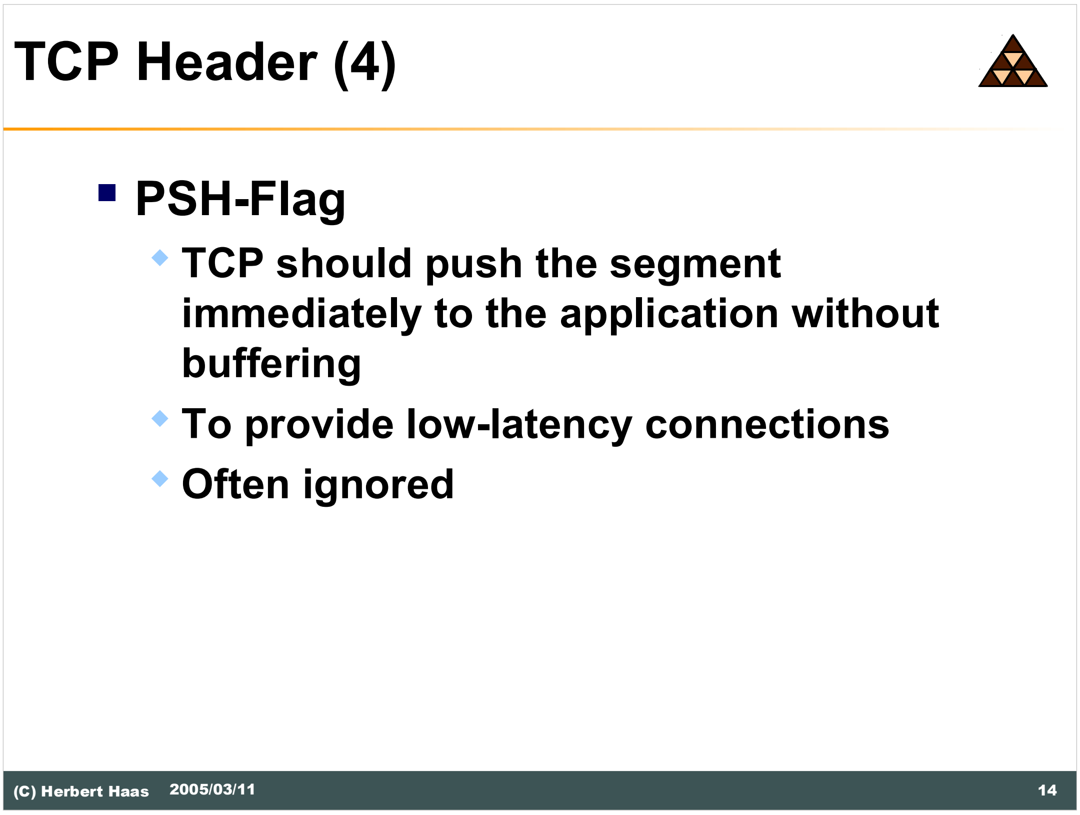
TCP Header (3)

- 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

URG-Flag: 1 Bit. Control Bit.

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* urgent data. However, the "Host Requirements" RFC 1122 states this as a mistake! 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 received. Examples for use: Interrupt key in Telnet, Rlogin, or FTP.

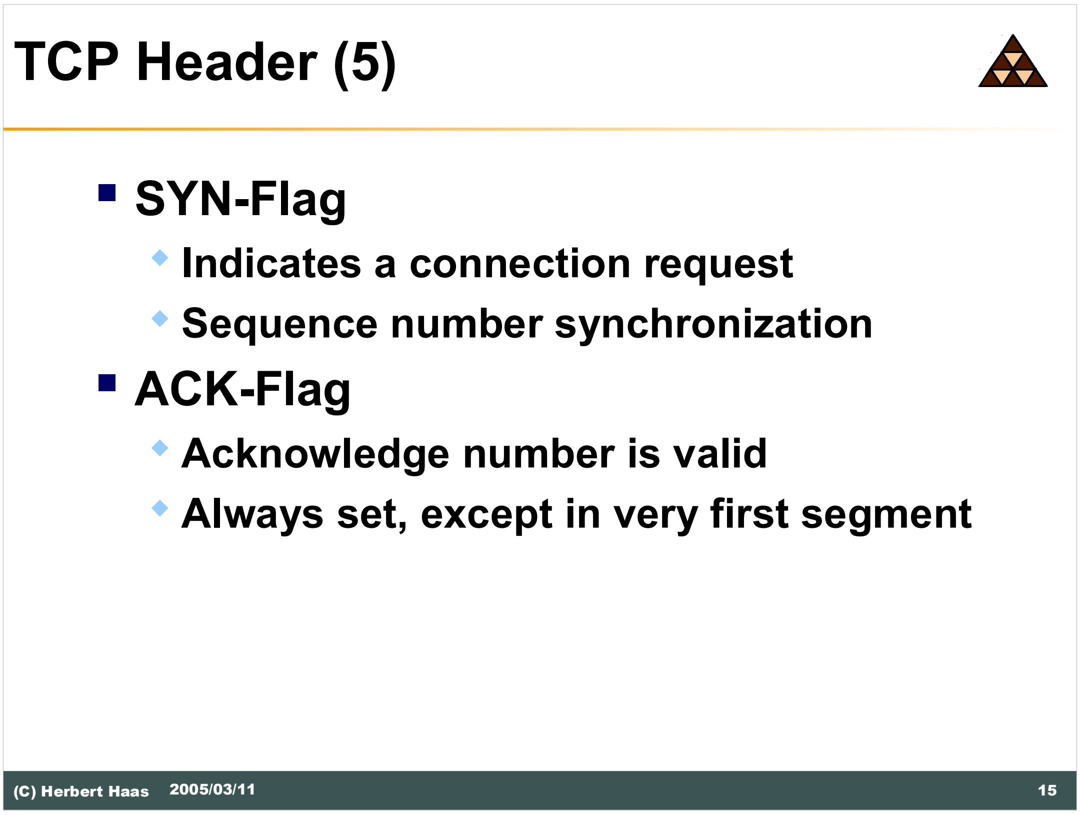


TCP Header (4)

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

PSH-Flag: 1 Bit. Control Bit.

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. To provide a low-latency connection sometimes the PSH Flag is set to 1. Then TCP should push the segment immediately to the application without buffing. But typically the PSH-Flag is ignored.



TCP Header (5)

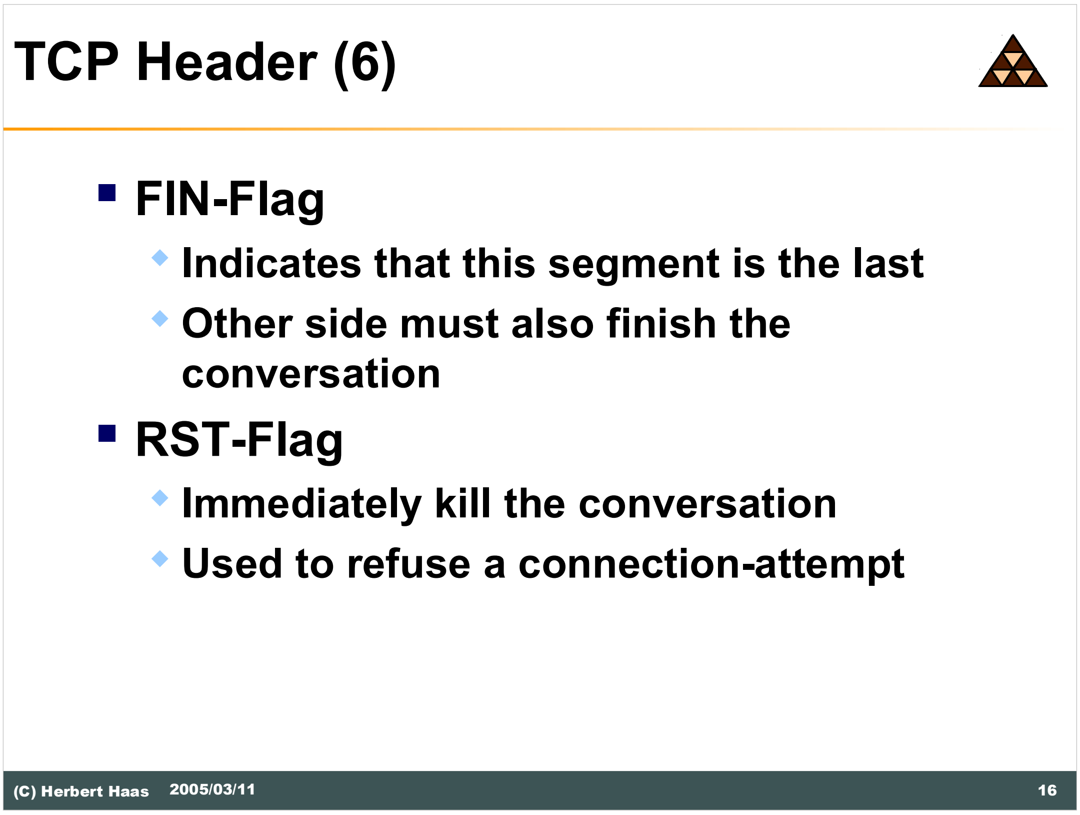
- SYN-Flag
  - Indicates a connection request
  - Sequence number synchronization
- ACK-Flag
  - Acknowledge number is valid
  - Always set, except in very first segment

SYN-Flag: 1 Bit. Control Bit.

If the SYN bit is set to 1, the application knows that the host want to established a connection with him. Also used to synchronization the sequence numbers. Most Firewalls through away packets with SYN=1 if the host want to established a connection to a application which the is server not allowed (security reasons).

ACK-Flag: 1 bit. Control Bit.

Acknowledgment Bit.



TCP Header (6)

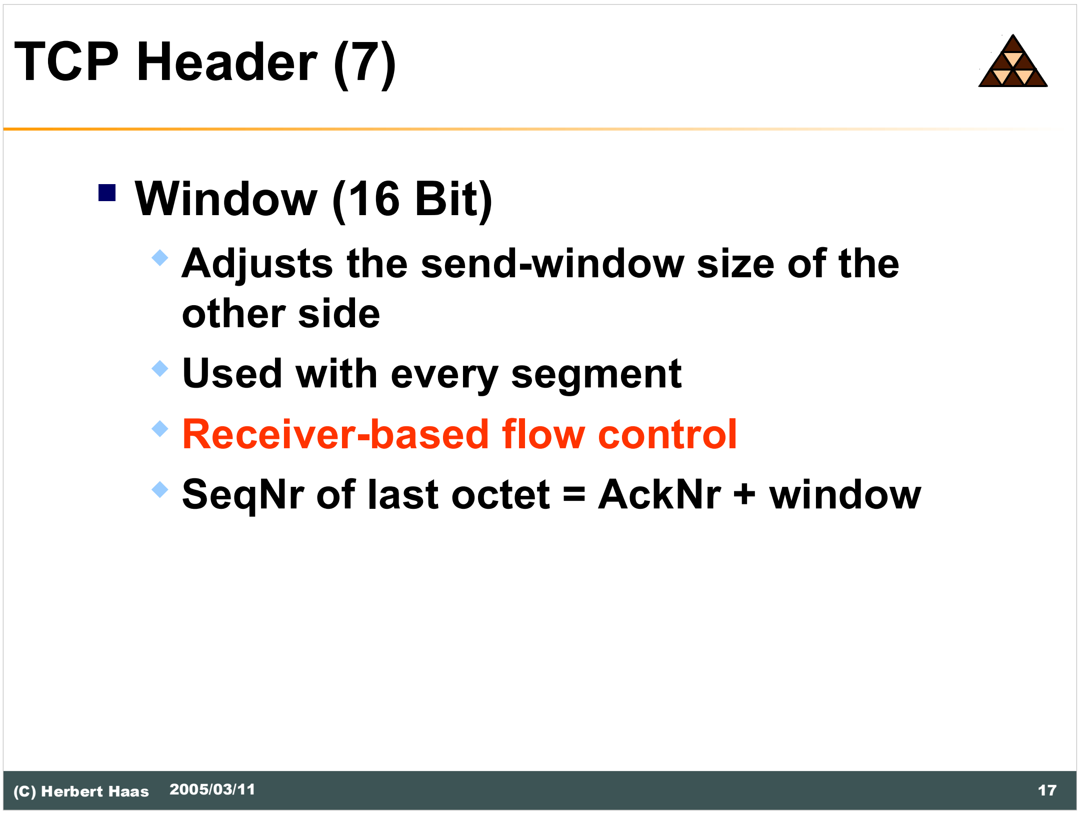
- FIN-Flag
  - Indicates that this segment is the last
  - Other side must also finish the conversation
- RST-Flag
  - Immediately kill the conversation
  - Used to refuse a connection-attempt

FIN-Flag: 1 bit. Control Bit.

The FIN-Flag is used in the “disconnect process”. It indicates that this segment is the last one. After the other side has also sent a segment with FIN=1, the connection is closed.

RST-Flag: 1 bit. Control Bit.

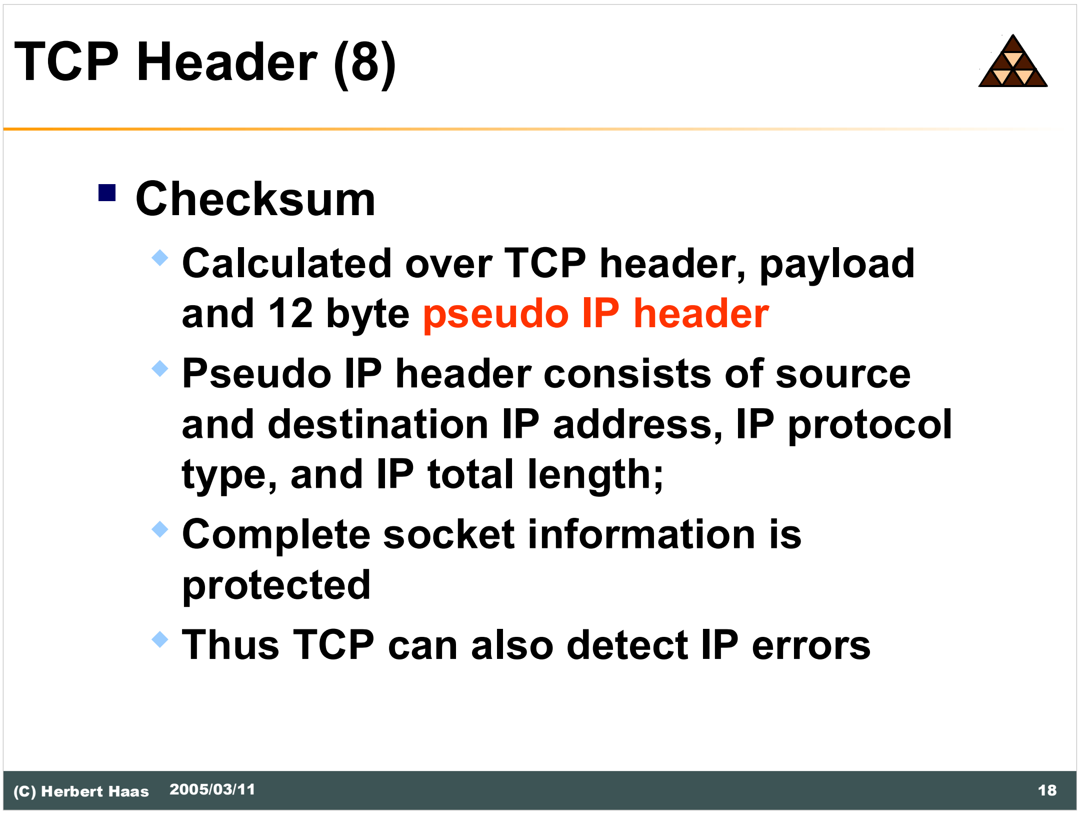
Resets the connection immediately.



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

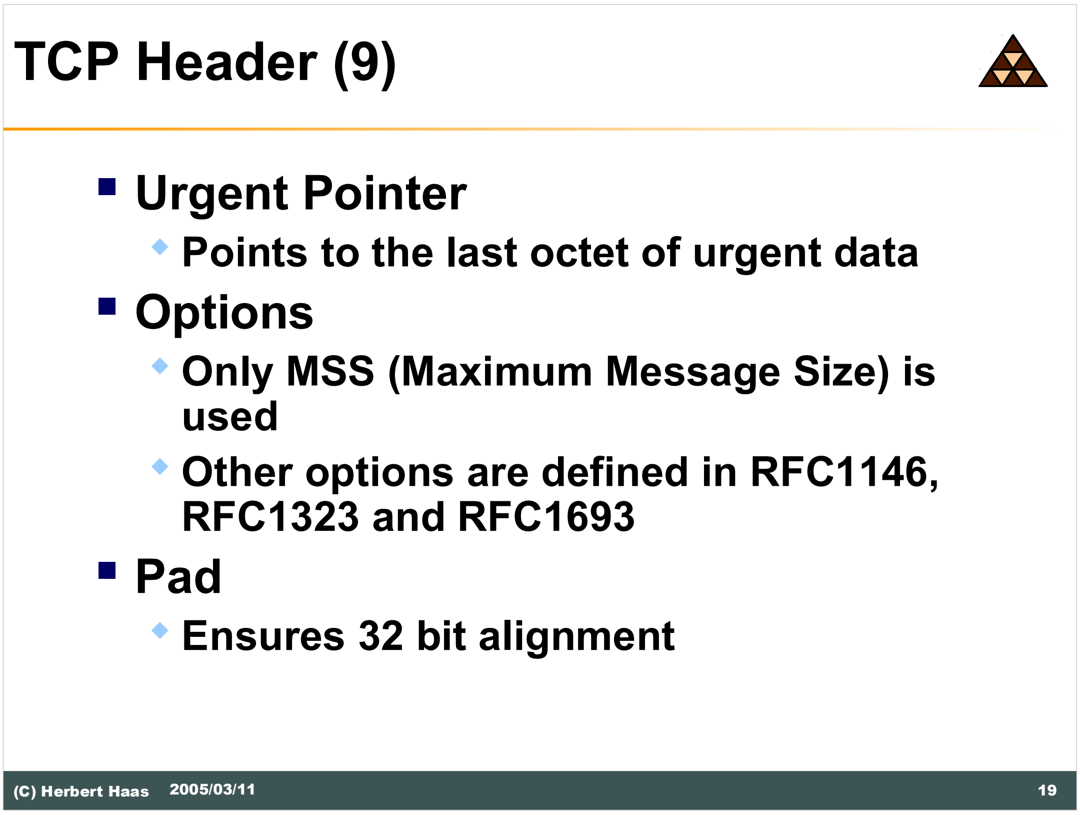
Window Size: 16 bit. The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept. See Slide 27.



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 Checksum: 16 bit. 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.



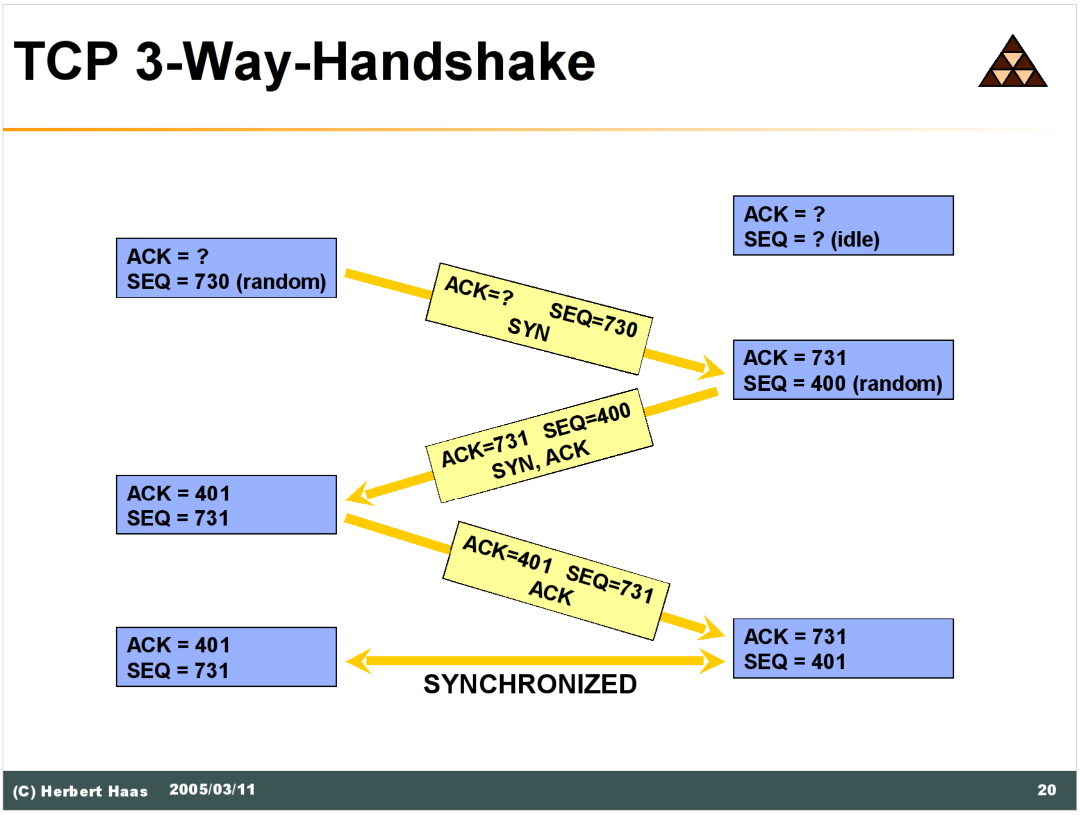
TCP Header (9)

- 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

Urgent Pointer: 16 bits. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

Options: Variable length. Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. Only the Maximum Message Size (MSS) is used. All options are included in the checksum.

Padding: Variable length. The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.



TCP 3-Way-Handshake

ACK = ?
SEQ = ? (idle)

ACK = ?
SEQ = 730 (random)

ACK=? SEQ=730
SYN

ACK = 731
SEQ = 400 (random)

ACK=731 SEQ=400
SYN, ACK

ACK = 401
SEQ = 731

ACK=401 SEQ=731
ACK

ACK = 401
SEQ = 731

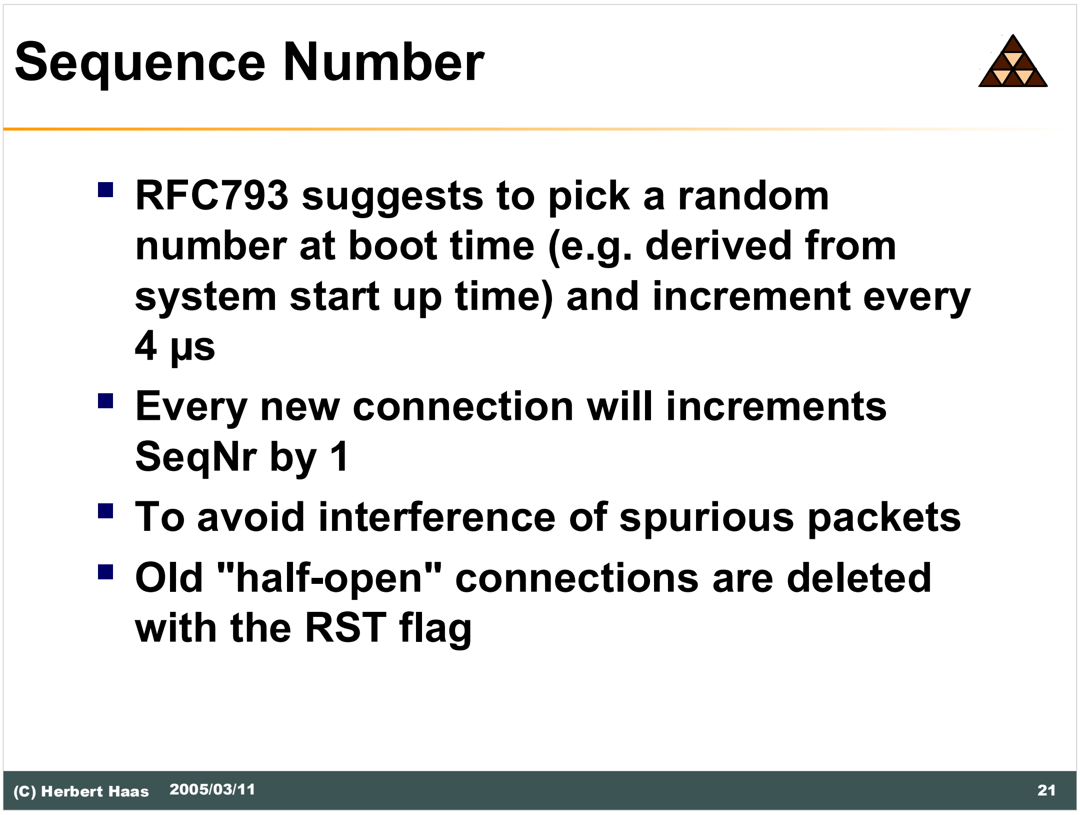
ACK = 731
SEQ = 401

SYNCHRONIZED

The diagram above shows the famous TCP 3-way handshake. The TCP 3-Way-Handshake is used to connect and synchronize two host with each other, that is, after the handshake procedure, both stations know the sequence numbers of each other.

The connection procedure (3-Way-Handshake) works with a simple principle. The host sends out a segment with SYN=1 (remember: if SYN=1 the application knows that the host want to established a connection) and the host also choose a random sequence number (SEQ). After the Server receives the segment correct, he acknowledgment (host-SEQ+1), also choose a random SEQ, and send back the segment with SYN=1. Remember the ACK-flag is always set, except in very first segment. Because the server sends back a segment with SYN=1 the host knows the connection is accepted. After the host sends a acknowledgement to the server the connection is established.

Note that a SYN consumes one sequence number! (After the 3-way handshake, only data bytes consume sequence numbers.)

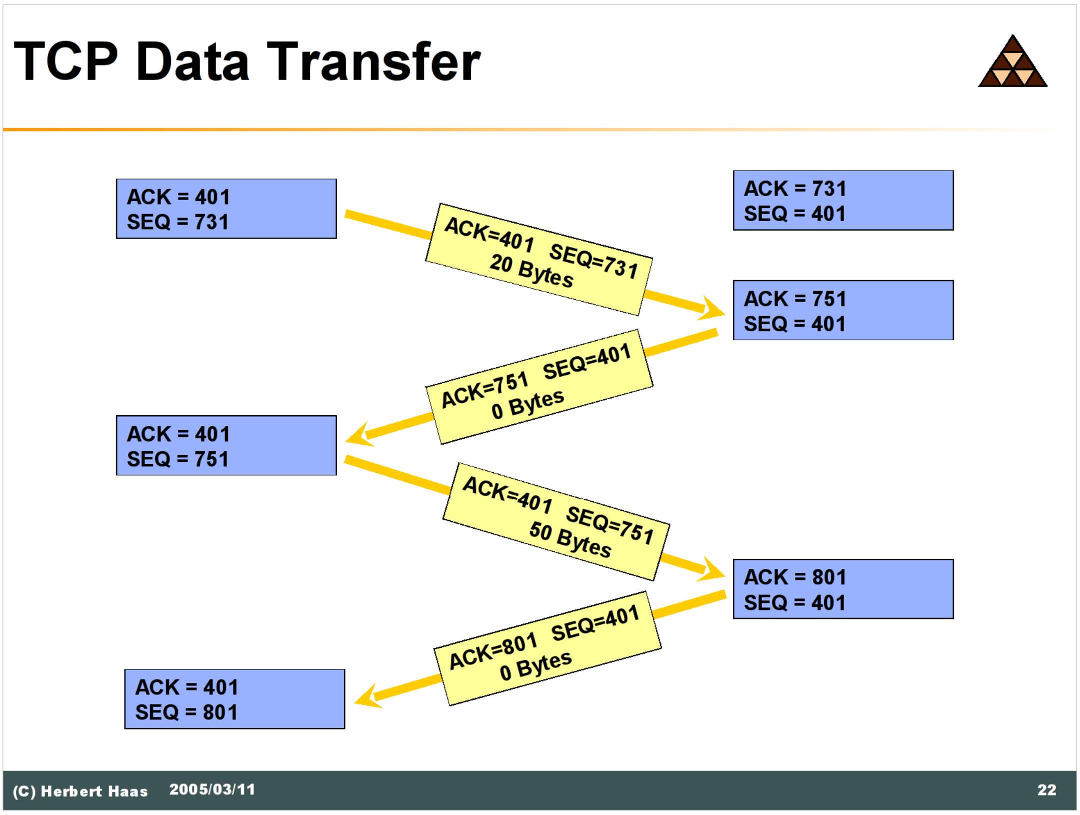


Sequence Number

- 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

RFC 793 suggests to pick a random starting sequence numbers and an explicit negotiation of starting sequence numbers to make a TCP connect immune against spurious packets.

Also disturbing segments (e.g. delayed TCP segments from old sessions etc.) and old "half-open" connections are deleted with the RST flag.



TCP Data Transfer

ACK = 401
SEQ = 731

ACK = 731
SEQ = 401

ACK=401 SEQ=731
20 Bytes

ACK = 751
SEQ = 401

ACK=751 SEQ=401
0 Bytes

ACK = 401
SEQ = 751

ACK=401 SEQ=751
50 Bytes

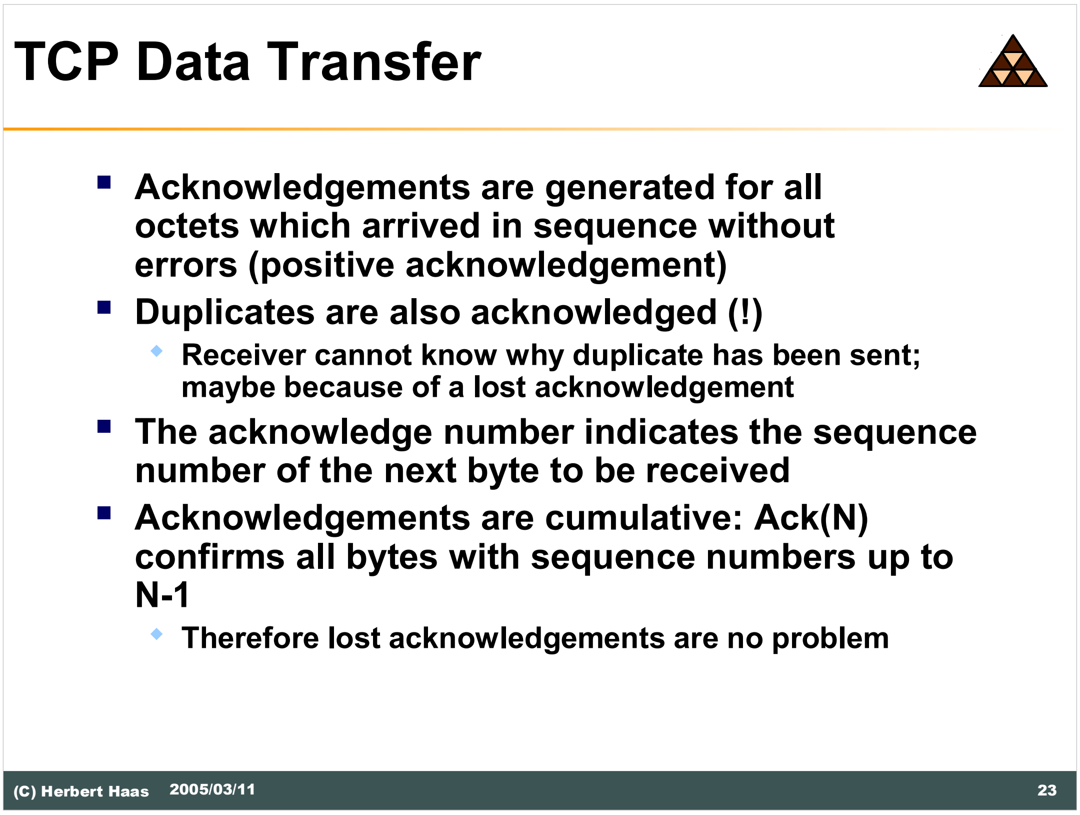
ACK = 801
SEQ = 401

ACK=801 SEQ=401
0 Bytes

ACK = 401
SEQ = 801

(C) Herbert Haas 2005/03/11 22

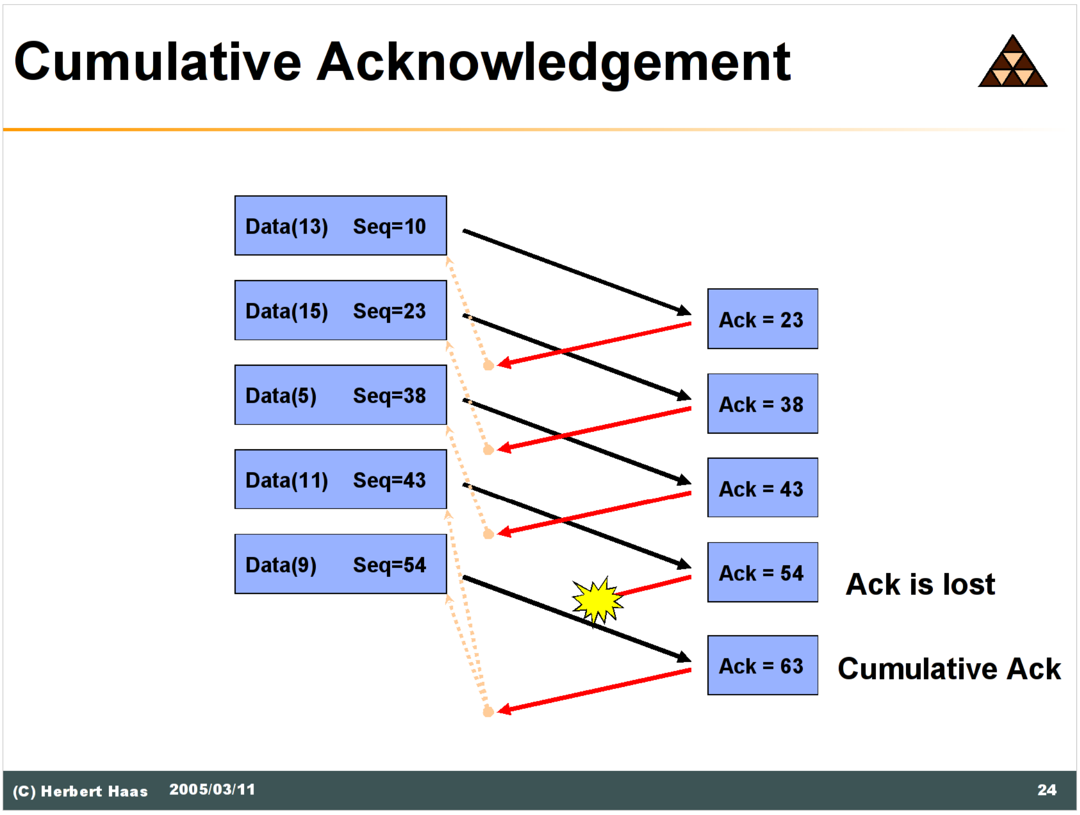
After the 3-way-handshake is finished the real data transfer is stared. A 20 Byte segment is sending to the server (ACK 401, SEQ 731). After the server receives the segment, he sets the ACK-flag to 751 (SEQ+20 Byte) and the SEQ to 401. Then he sends the segment pack (ACK 751, SEQ 401) to the host. After the host receives this segment he know that his 20 byte of date delivers correct (because he gets the ACK 751). The host continuous sending his data to the server.



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

The acknowledge number is equal to the sequence number of the next octet to be received.



Cumulative Acknowledgement

Data(13) Seq=10

Data(15) Seq=23

Data(5) Seq=38

Data(11) Seq=43

Data(9) Seq=54

Ack = 23

Ack = 38

Ack = 43

Ack = 54

Ack = 63

Ack is lost

Cumulative Ack

Its not a problem for TCP when a acknowledgment get lost, because TCP is acknowledge all receiving data with every acknowledgement.

TCP Duplicates:

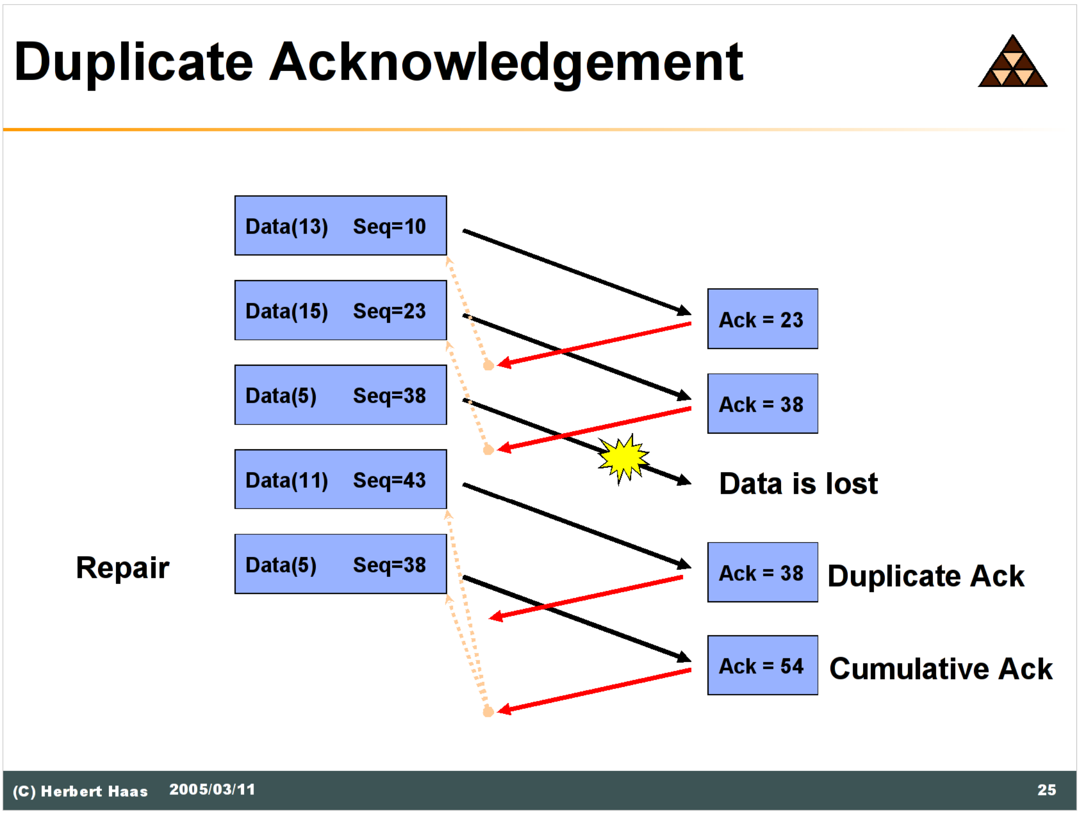
There are some reasons for retransmission if a TCP duplicate occur:

•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

The 32 bit sequence numbers provide enough "space" to tell duplicates from originals: 232 Octets with 2 Mbit/s means 9h for wrap around (compare to usual TTL = 64 seconds)



Duplicate Acknowledgement

Data(13) Seq=10

Data(15) Seq=23

Data(5) Seq=38

Data(11) Seq=43

Repair

Data(5) Seq=38

Ack = 23

Ack = 38

Data is lost

Ack = 38 Duplicate Ack

Ack = 54 Cumulative Ack

(C) Herbert Haas 2005/03/11 25

Its not a problem for TCP when a acknowledgment get lost, because TCP is acknowledge all receiving data with every acknowledgement.

TCP Duplicates:

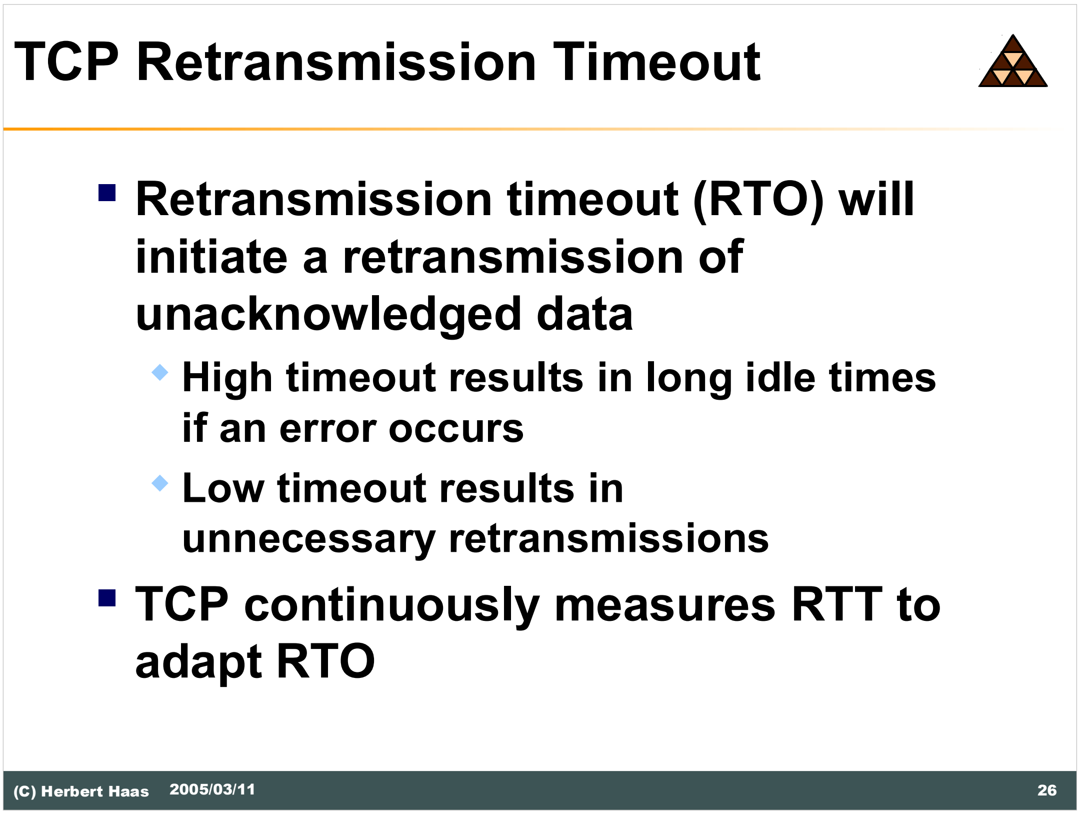
There are some reasons for retransmission if a TCP duplicate occur:

•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

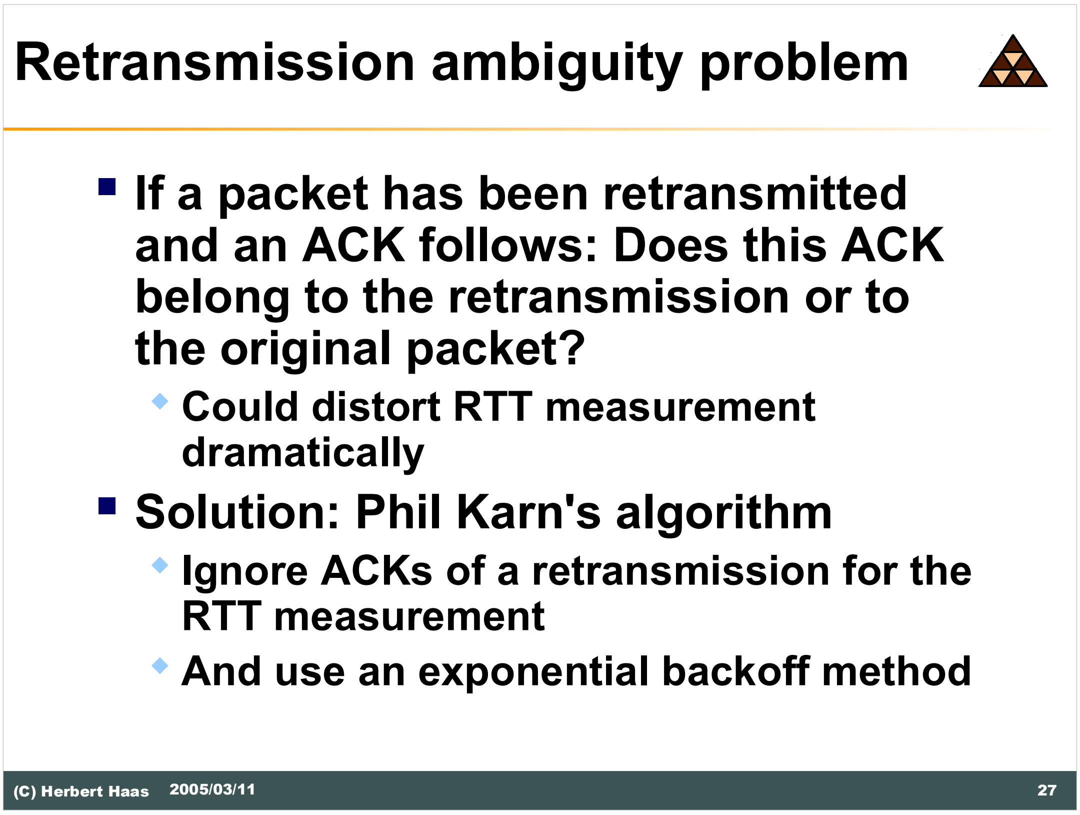
•Because original was delayed and timeout expired: no problem, segment is recognized as duplicate through the sequence number

The 32 bit sequence numbers provide enough "space" to tell duplicates from originals: 232 Octets with 2 Mbit/s means 9h for wrap around (compare to usual TTL = 64 seconds)



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

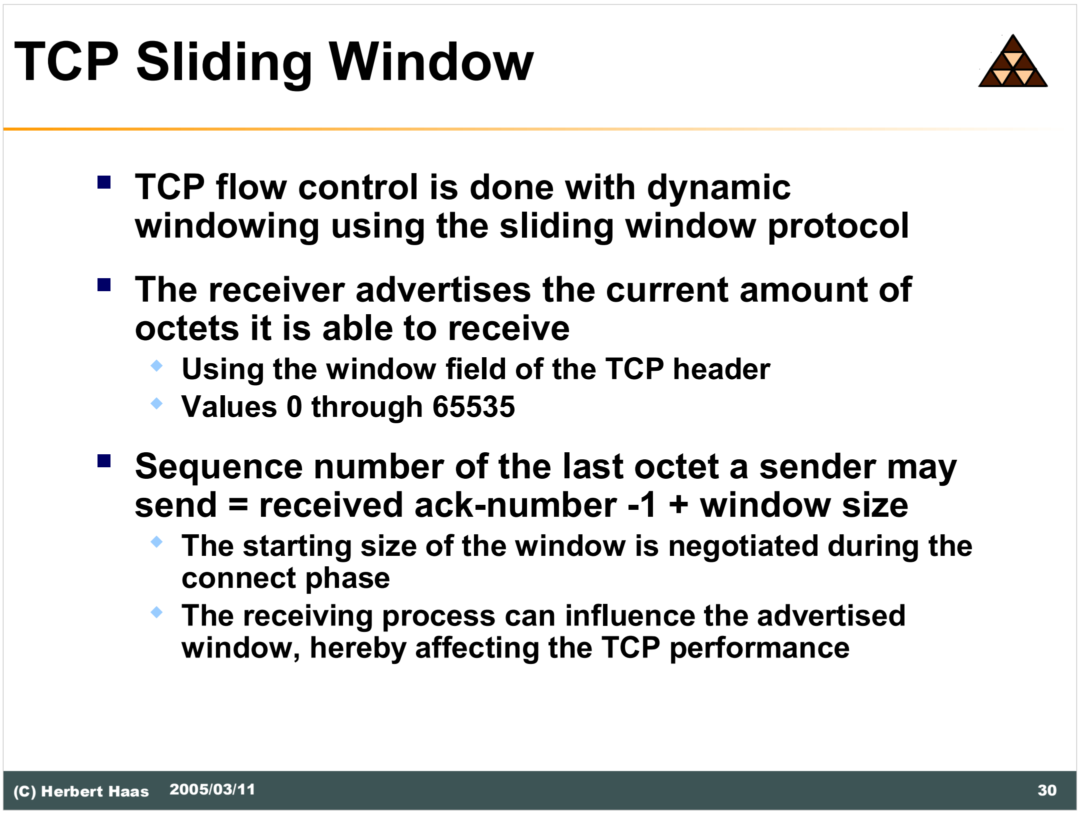
The exponential backoff algorithm means that the retransmission timeout is doubled every time the timer expires and the particular data segment was still not acknowledged. However, the backoff is truncated usually at 64 seconds.

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 $\beta=2$

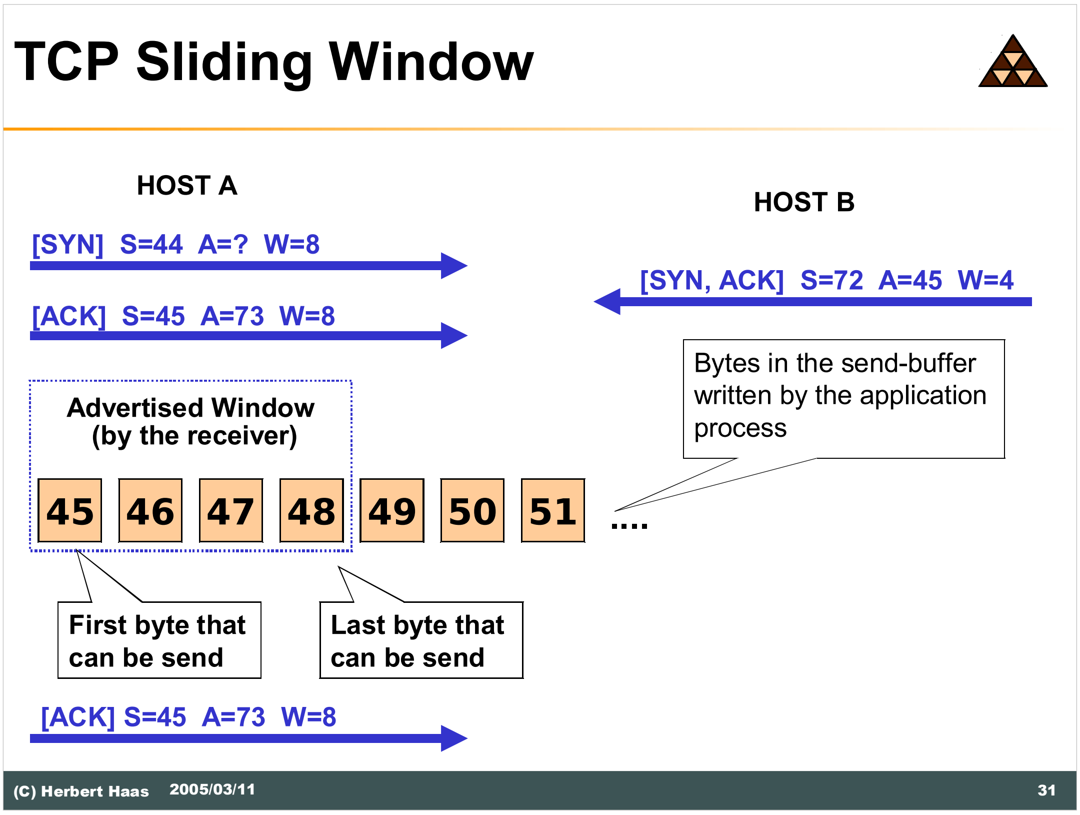
RTT Estimation (2/2)

- 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)
  - $A = A + g \cdot Err$
    - 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

HOST A

HOST B

[SYN] S=44 A=? W=8

[SYN, ACK] S=72 A=45 W=4

[ACK] S=45 A=73 W=8

Advertised Window
(by the receiver)

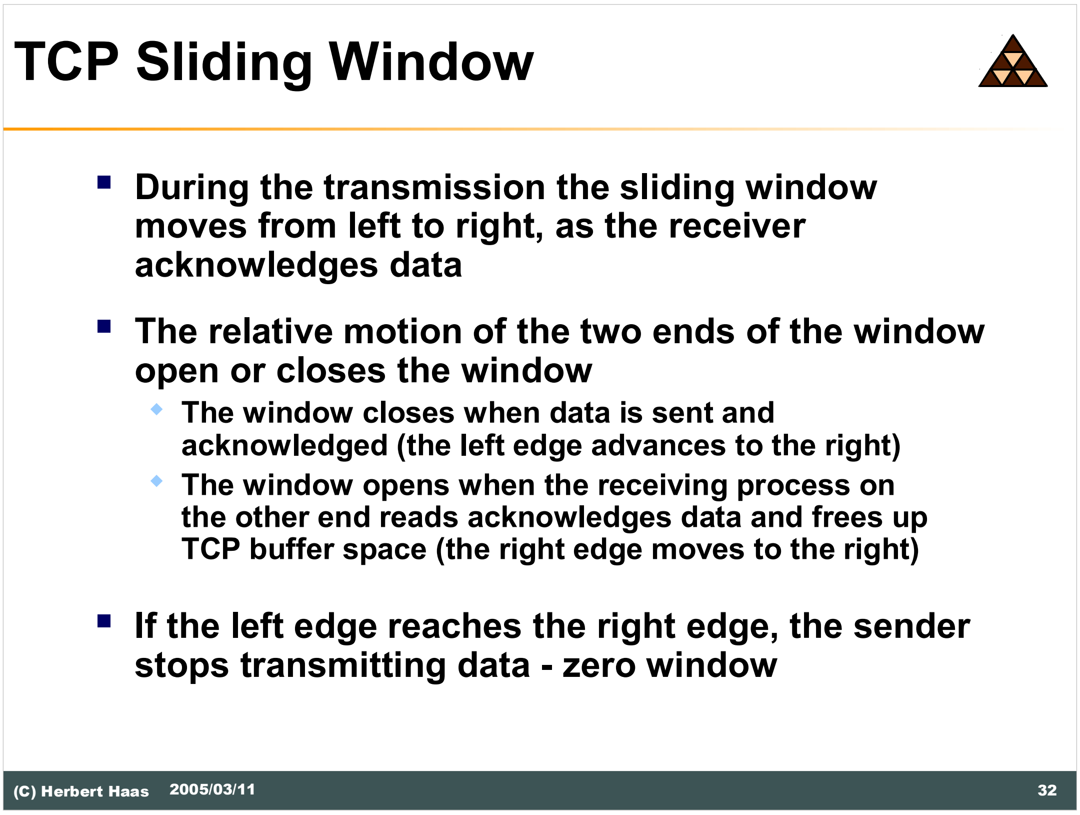
Bytes in the send-buffer written by the application process

45 46 47 48 49 50 51 ....

First byte that can be send

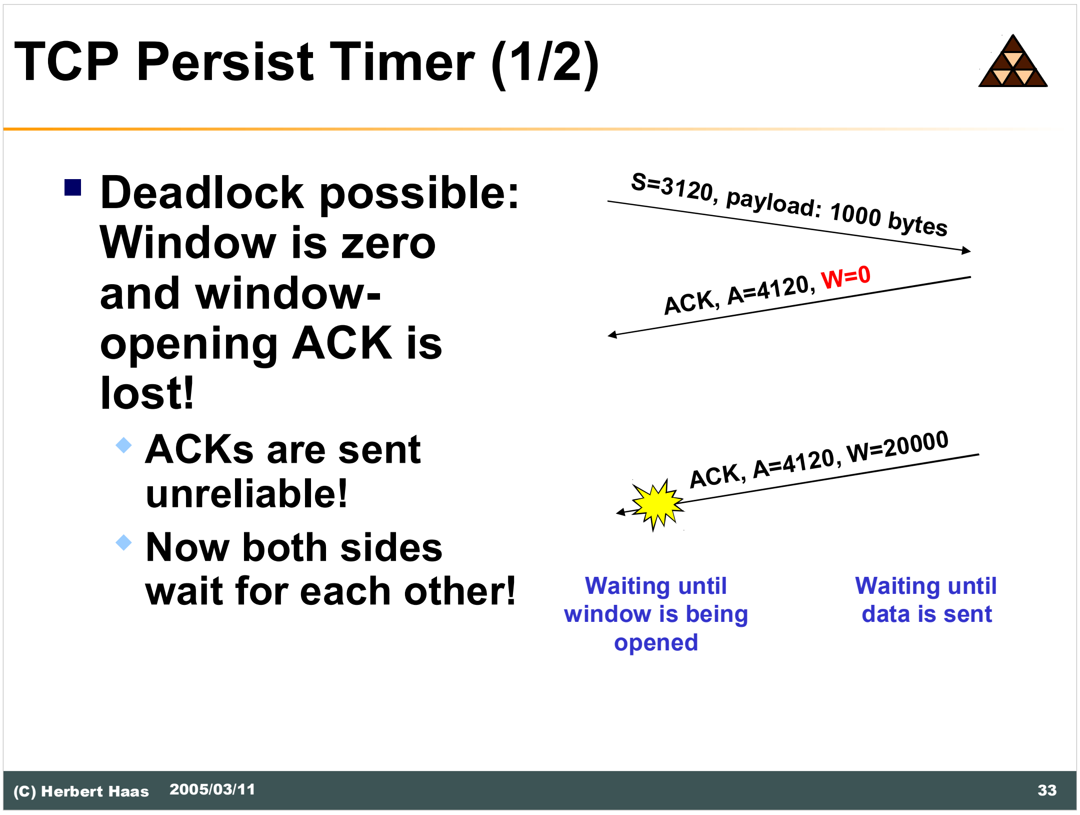
Last byte that can be send

[ACK] S=45 A=73 W=8



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



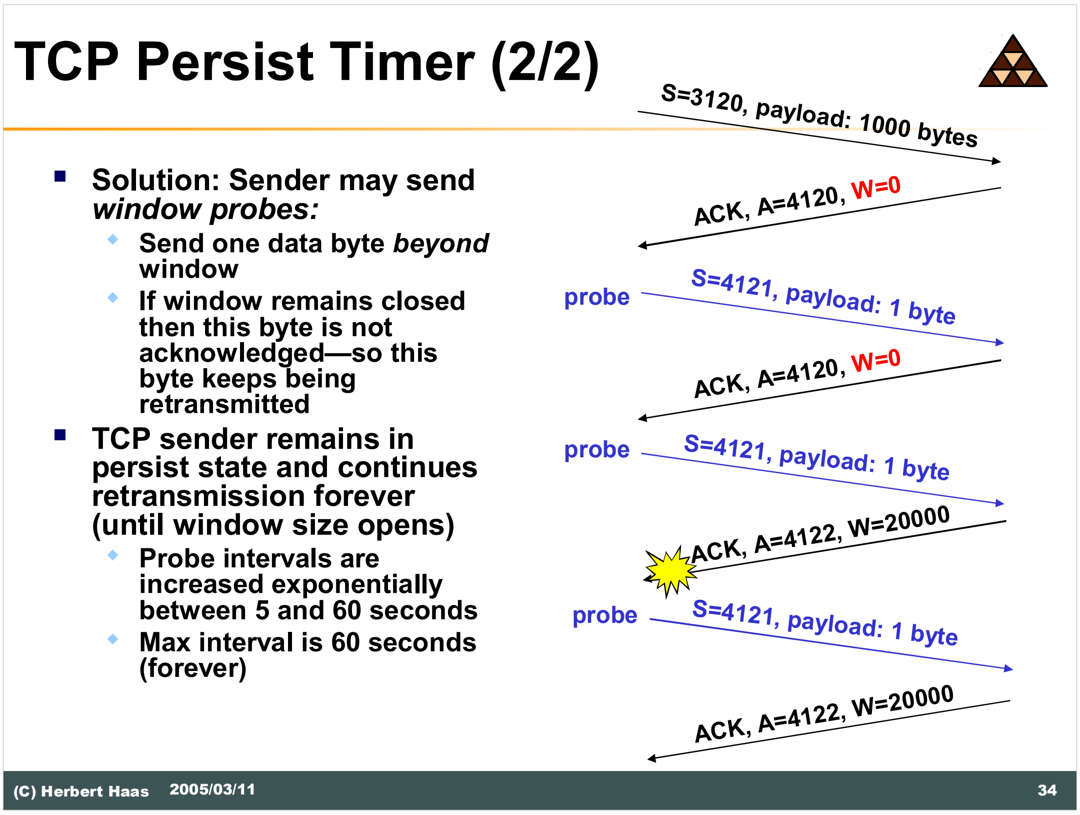
TCP Persist Timer (1/2)

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

S=3120, payload: 1000 bytes
ACK, A=4120, W=0
ACK, A=4120, W=20000
Waiting until window is being opened
Waiting until data is sent

Only if the ACK also contains data then the peer would retransmit it after timer expiration.

Window probes may be used to query receiver if window has been opened already.



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)

S=3120, payload: 1000 bytes

ACK, A=4120, W=0

probe S=4121, payload: 1 byte

ACK, A=4120, W=0

probe S=4121, payload: 1 byte

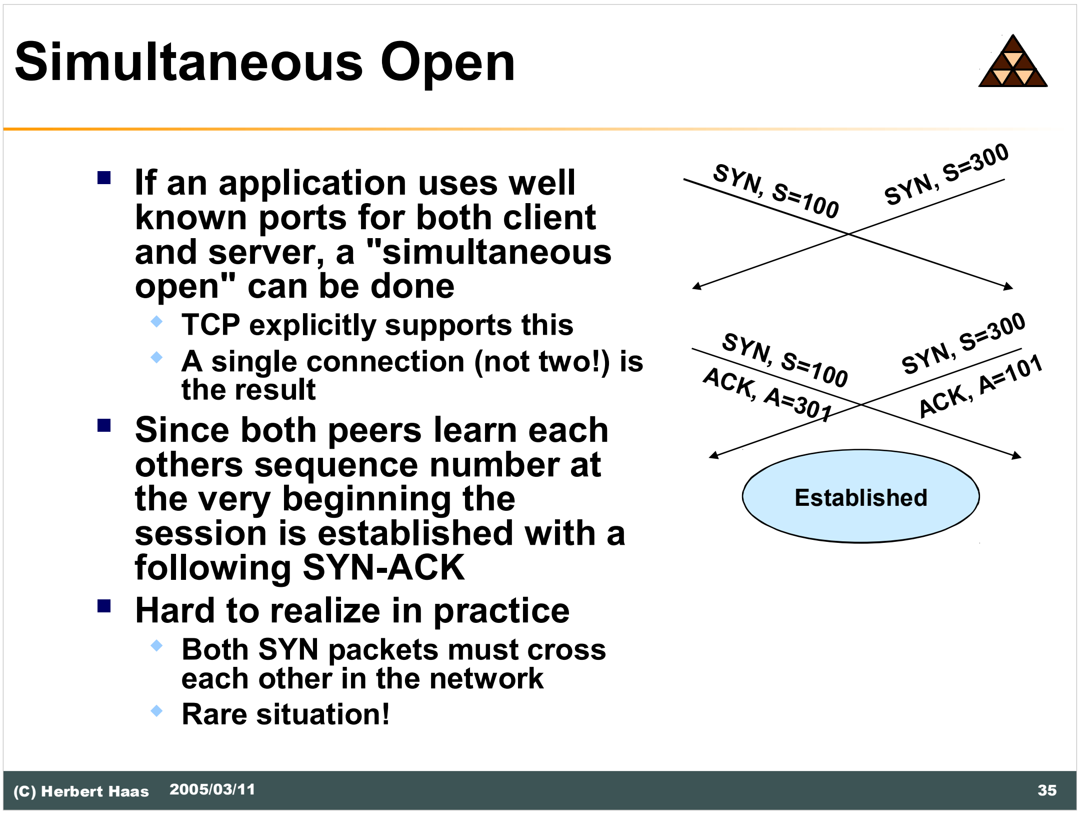
ACK, A=4122, W=20000

probe S=4121, payload: 1 byte

ACK, A=4122, W=20000

(C) Herbert Haas 2005/03/11 34

Since sender really has data to send the sender can use single bytes of the bytestream to be send for ACK probes. The window probing interval is increased similar as the normal retransmission interval following a truncated exponential backoff, but is always bounded between 5 and 60 seconds. If the peer does not open the window again the sender will transmit a window probe every 60 seconds.

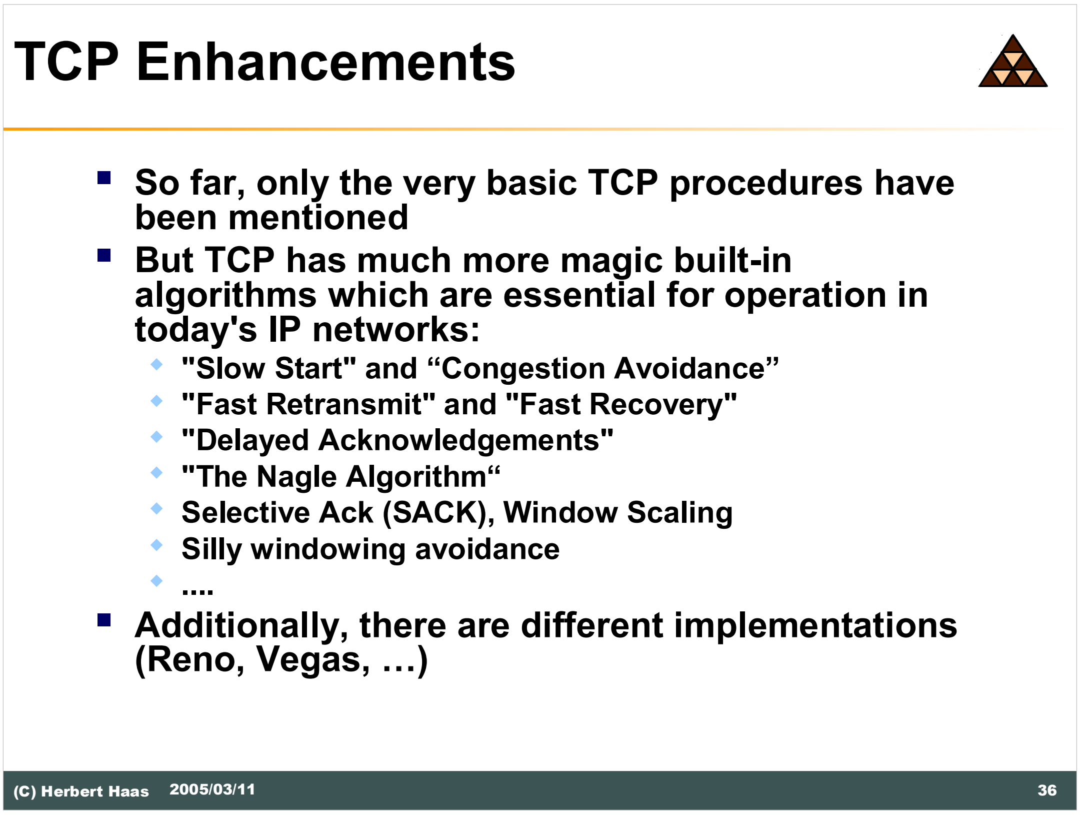


Simultaneous Open

- 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!

True OSI protocols would establish two separate connections but TCP would result in a single connection.

Note the different SQNR handling in the handshake!



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, …)

“Slow Start” and “Congestion avoidance” are mechanisms that control the segment rate (per RTT).

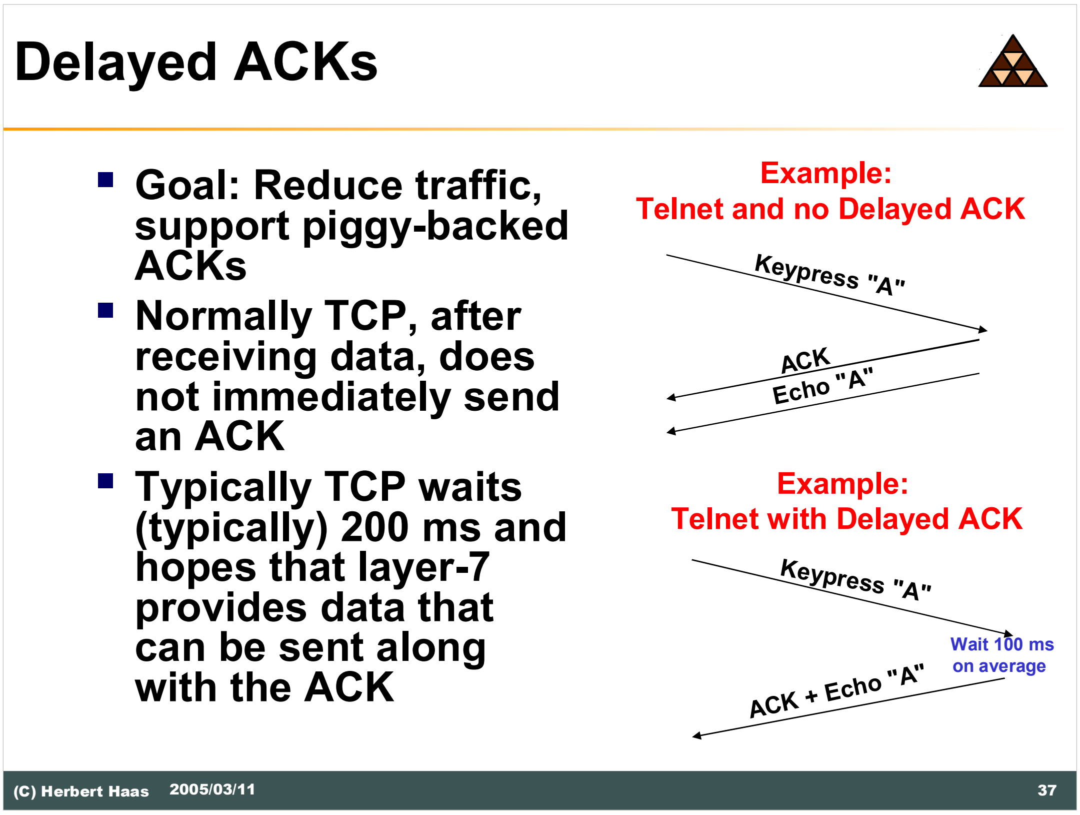
“Fast Retransmit” and “Fast Recovery” are mechanisms to avoid waiting for the timeout in case of retransmission and to avoid slow start after a fast retransmission.

Delayed Acknowledgements is typically used with applications like Telnet: Here each client-keystroke triggers a single packet with one byte payload and the server must response with both an echo plus a TCP acknowledgement. Note that also this server-echo must be acknowledged by the client. Therefore, layer-4 delays the acknowledgements because perhaps layer-7 might want to send some bytes also.

The Nagle algorithm tries to make WAN connections more efficient. We simply delay the segment transmission in order to collect more bytes from layer-7.

Selective Acks enhance the traditional positive-ack-mechanism and allows to selectively acknowledge some correctly received segments within a larger corrupted block.

Window Scaling deals with the problem of a jumping window in case the RTT-BW-product is greater than 65535 (the classical max window size). This TCP option allows to left-shift the window value (each bit-shift is like multiply by two).



Delayed ACKs

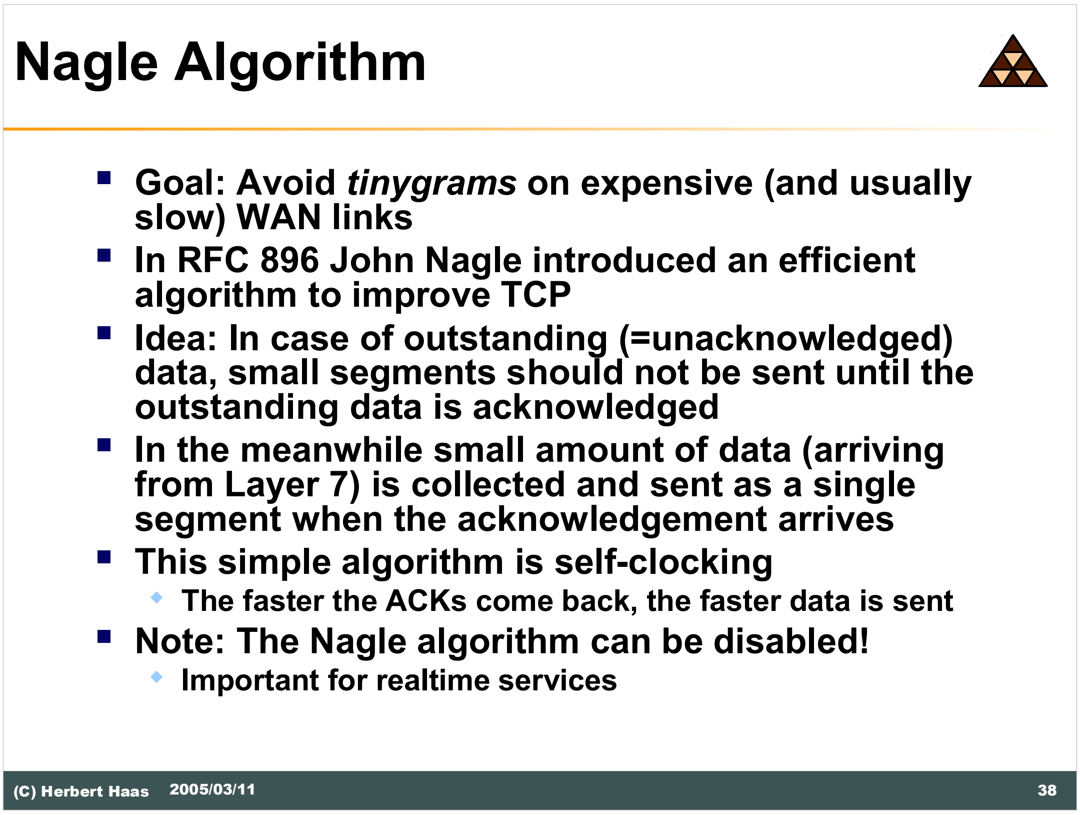
- Goal: Reduce traffic, support piggy-backed ACKs
- Normally TCP, after receiving data, does not immediately send an ACK
- Typically TCP waits (typically) 200 ms and hopes that layer-7 provides data that can be sent along with the ACK

Example:
Telnet and no Delayed ACK

Example:
Telnet with Delayed ACK

Actually the kernel maintains a 200 msec timer and every TCP session waits until this central timer expires before sending an ACK. If we are lucky the application has given us also some data to send, otherwise the ACK is sent without any payload. This is the reason, why we usually do not observe exact 200 msec delay between reception of a TCP packet and transmission of an ACK, rather the delay is something between 1 and 200 msec.

The Hosts Requirement RFC (1122) states that TCP should be implemented with Delayed ACK and that the delay must be less than 500 ms.



Nagle Algorithm

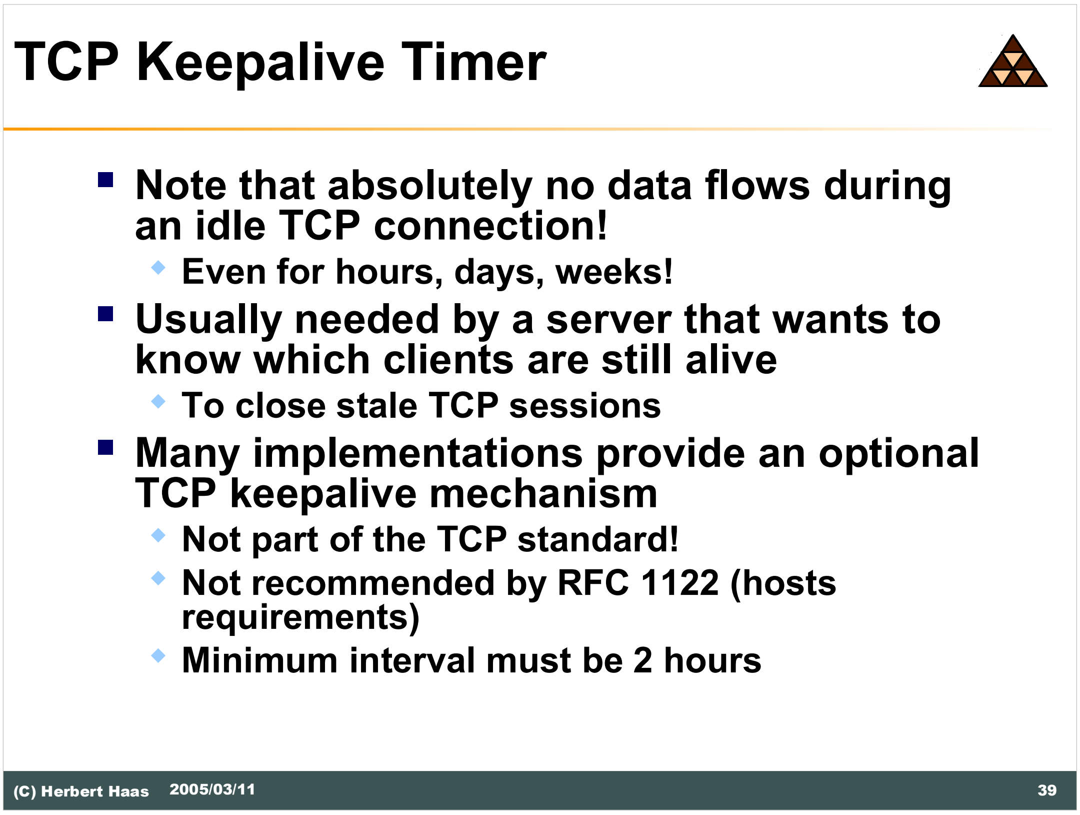
- Goal: Avoid *tinygrams* on expensive (and usually slow) WAN links
- In RFC 896 John Nagle introduced an efficient algorithm to improve TCP
- Idea: In case of outstanding (=unacknowledged) data, small segments should not be sent until the outstanding data is acknowledged
- In the meanwhile small amount 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

A tinygram is a very small packet, for example with a single byte payload. The total packet size would be 20 bytes IP, 20 bytes TCP plus 1 byte data (plus 18 bytes Ethernet). No problem on a LAN but lots of tinygrams may congest the (typically much) slower WAN links.

In this context, "small" means less than the segment size.

Note that the Nagle Algorithm can be disabled, which is important for certain realtime services. For example the X Window protocol disables the Nagle Algorithm so that e. g. realtime feedback of mouse movements can be communicated without delay.

The socket API provides the symbol TCP_NODELAY.

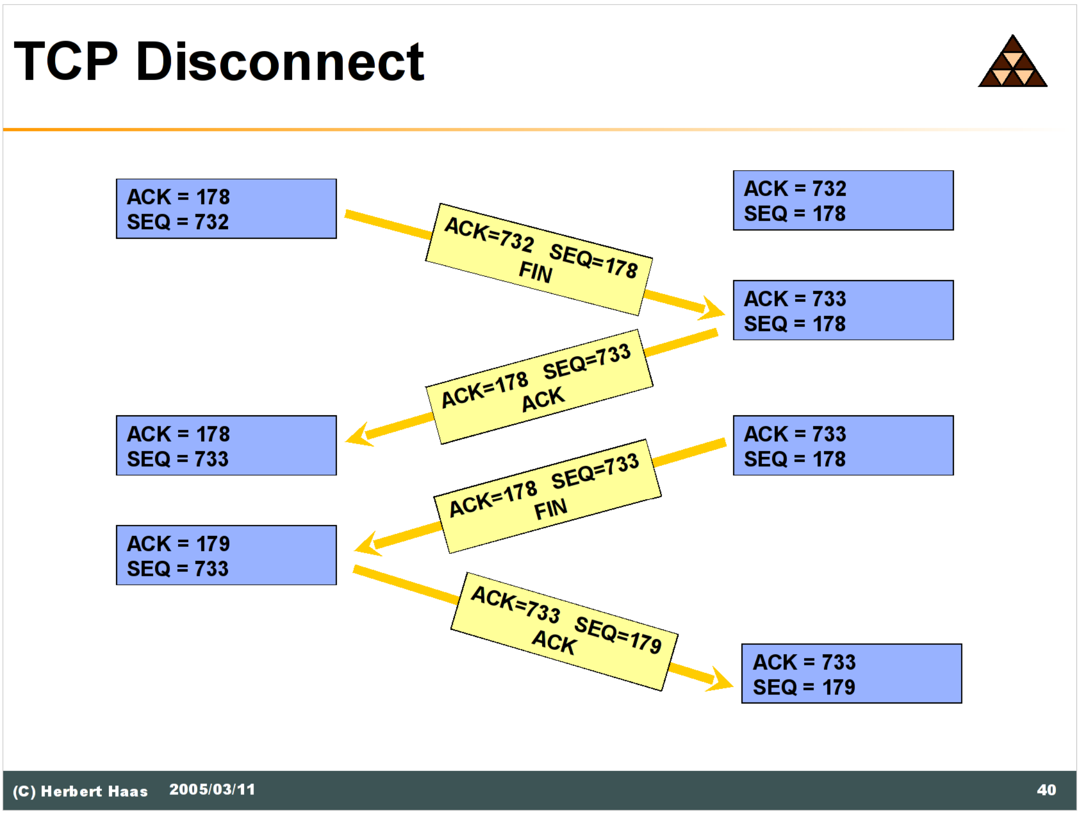


TCP Keepalive Timer

- 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

Sessions may remain up even for month without any data being sent.

The Host Requirements RFC mentions three disadvantages: 1) Keepalives can cause perfectly good connections to be dropped during transient failures, 2) they consume unnecessary bandwidth, and 3) they cost money when the ISP charge at a per packet base. Furthermore many people think that keepalive mechnisms should be implemented at the application layer.



TCP Disconnect

ACK = 178
SEQ = 732

ACK = 732
SEQ = 178

ACK=732 SEQ=178
FIN

ACK = 733
SEQ = 178

ACK=178 SEQ=733
ACK

ACK = 178
SEQ = 733

ACK = 733
SEQ = 178

ACK=178 SEQ=733
FIN

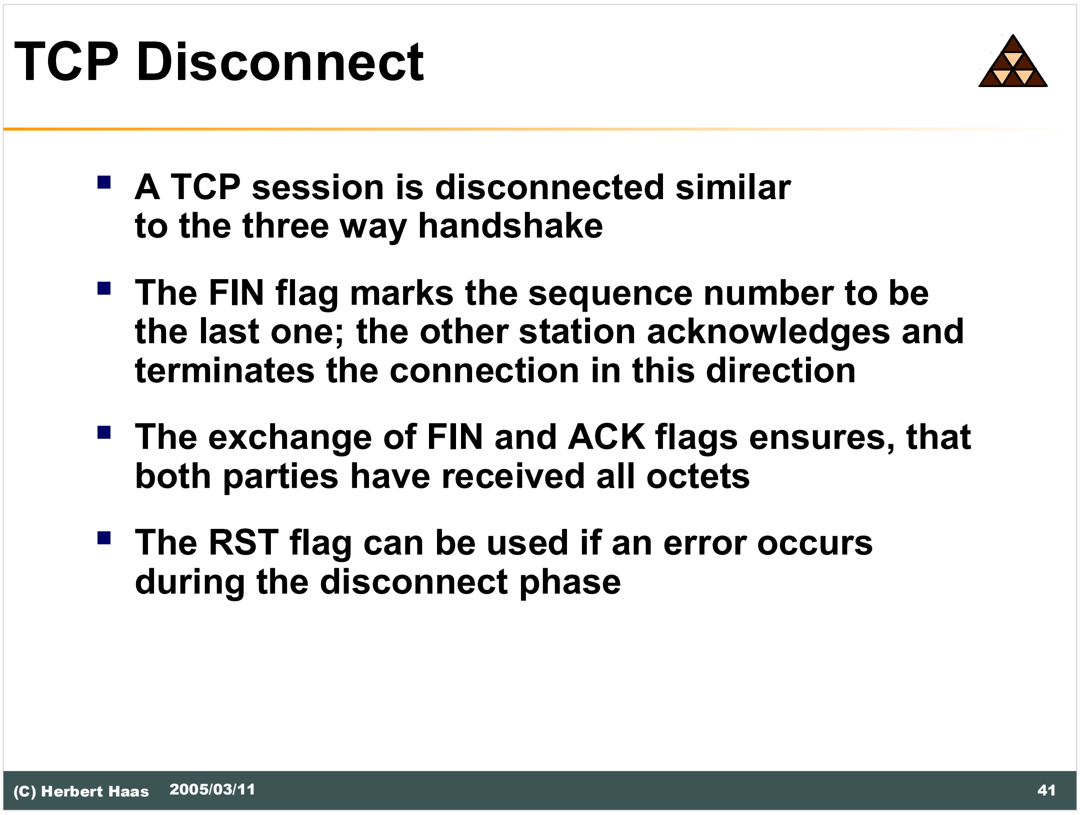
ACK = 179
SEQ = 733

ACK=733 SEQ=179
ACK

ACK = 733
SEQ = 179

(C) Herbert Haas 2005/03/11 40

The "ordered" disconnect process is also a handshake, slightly similar to the 3-Way-Handshake. The exchange of FIN and ACK flags ensures, that both parties have received all octets.

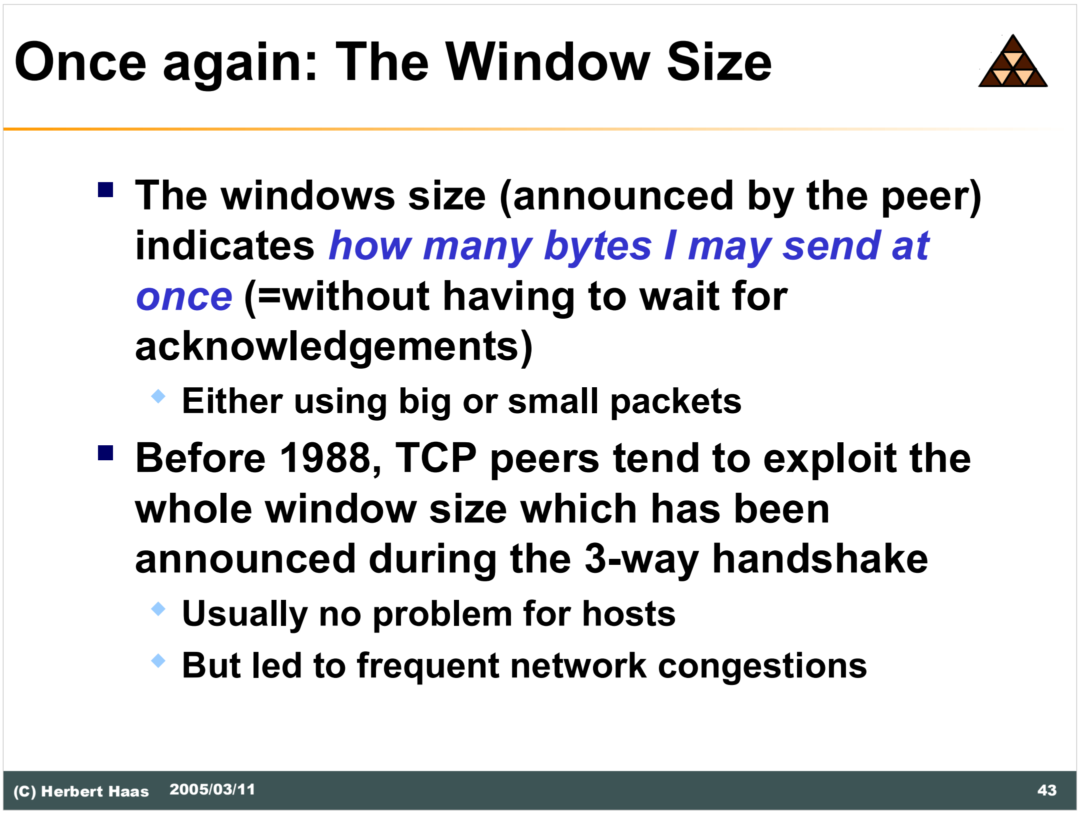


TCP Disconnect

- 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

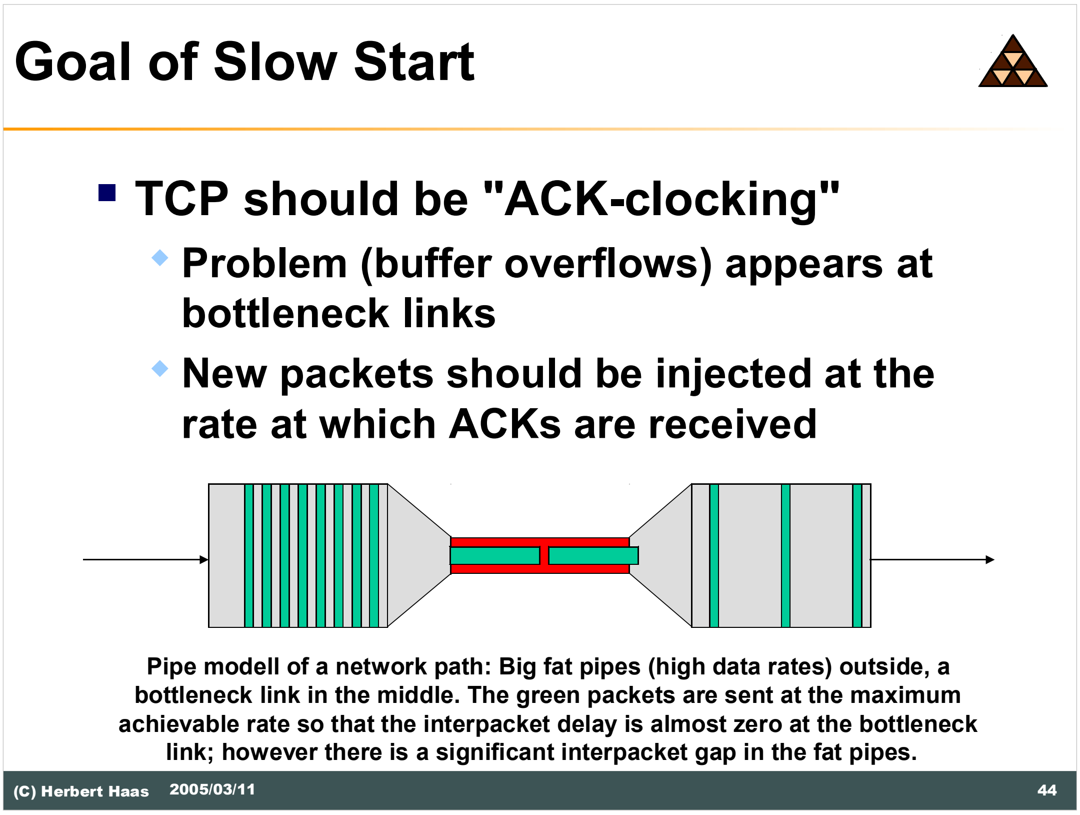
1. Slow Start & Congestion Avoidance
2. Random Early Discard
3. Explicit Congestion Notification



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

Note that hosts only need to deal with a single or a few TCP connections while network nodes such as routers and switches must transfer thousands, sometimes even millions of connections. Those nodes must queue packets and schedule them on outgoing interfaces (which might be slower than the inbound rates). If all TCP senders transmit at "maximum speed" – i. e. what is announced by the window – then network nodes may experience buffer overflows.

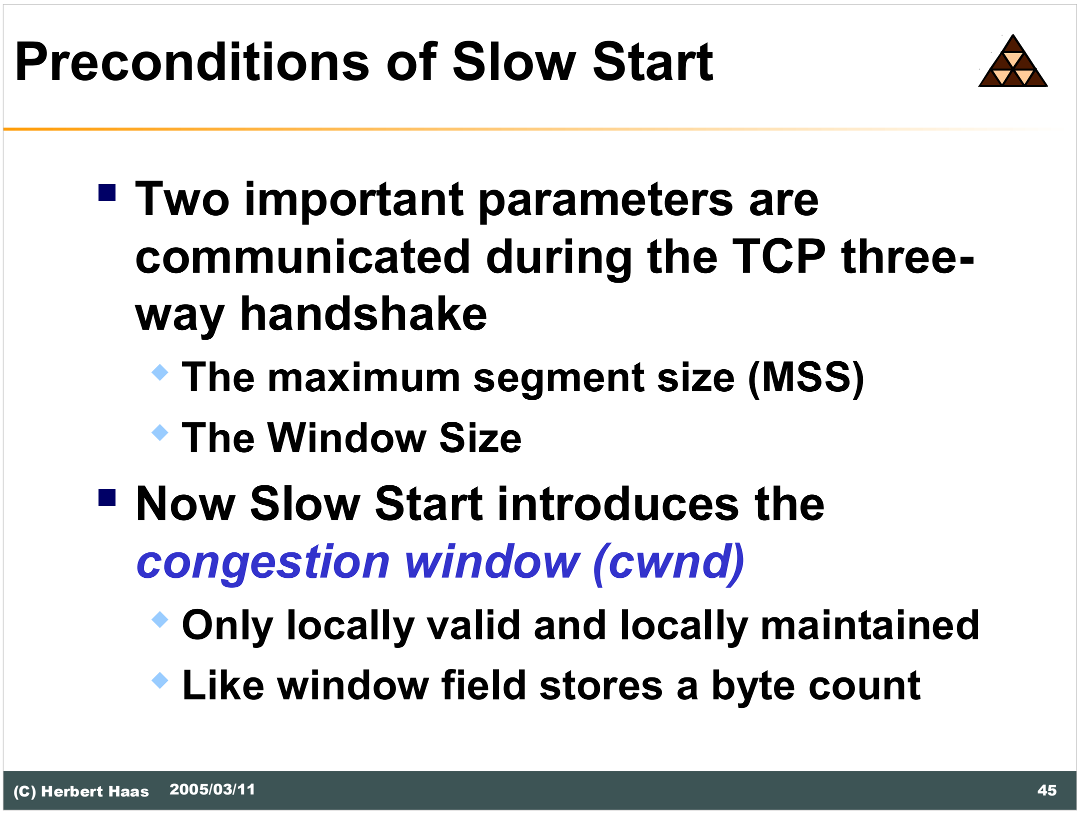


Goal of Slow Start

- 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.

Using TCP the depths of the queues are controlled by the ACK frequency, therefore TCP is called to be **ACK-clocked**. Only when an ACK is received the next segment is sent. Therefore TCP is self-regulating and the queue-depth is determined by the bottleneck: Every node runs exactly at the bottleneck link rate. If a higher rate would be used, then ACKs stay out and TCP would throttle its sending rate.



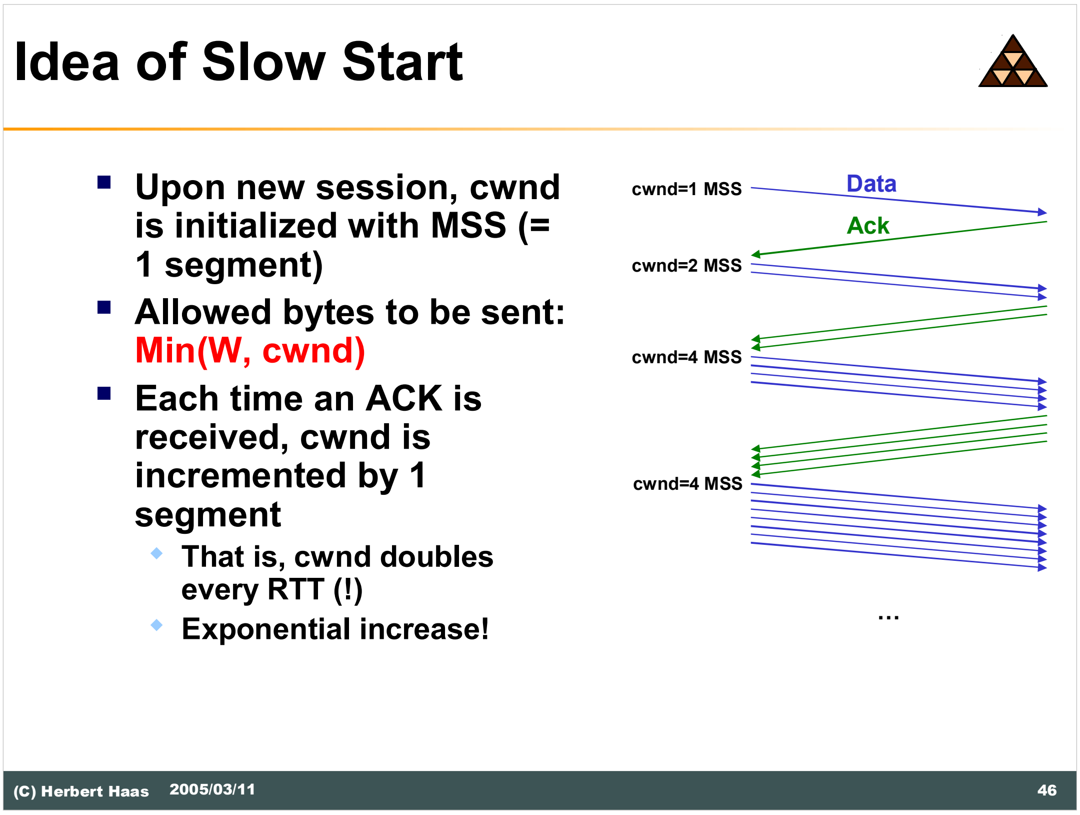
Preconditions of Slow Start

- 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

(C) Herbert Haas 2005/03/11 45

The MSS is typically around 1024 bytes or more but does NOT count the TCP/IP header overhead, so the true packet is 20+20 bytes larger. The MSS is not negotiated, rather each peer can announce ist acceptable MSS size and the other peer must obey. If no MSS option is communicated then the default of 536 bytes (i. e. 576 in total with IP and TCP header) is assumed.

Note: The MSS is only communicated in SYN-packets.

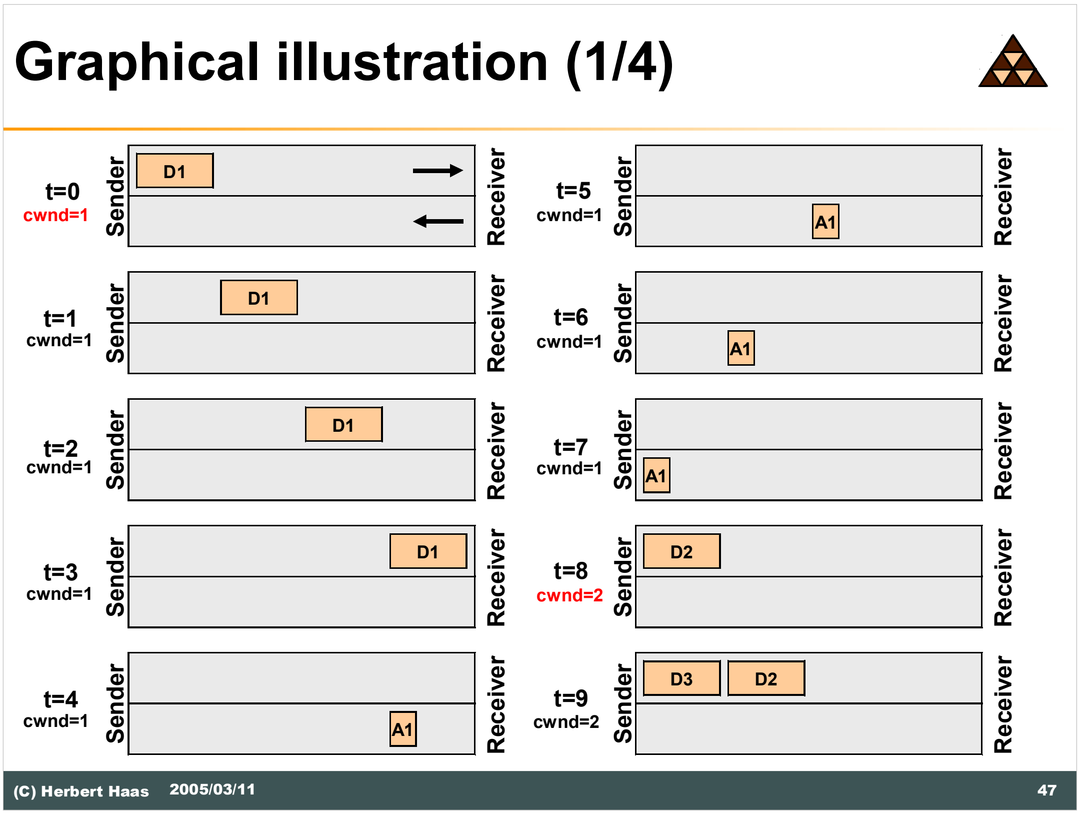


Idea of Slow Start

- 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!

Note that the sender may transmit up to the minimum of the congestion window (cwnd) and the advertized window (W).

The cwnd implements sender-imposed flow control, the advertized window allows for receiver-imposed flow control. But how does this mechanism deal with network congestion? Continue reading!

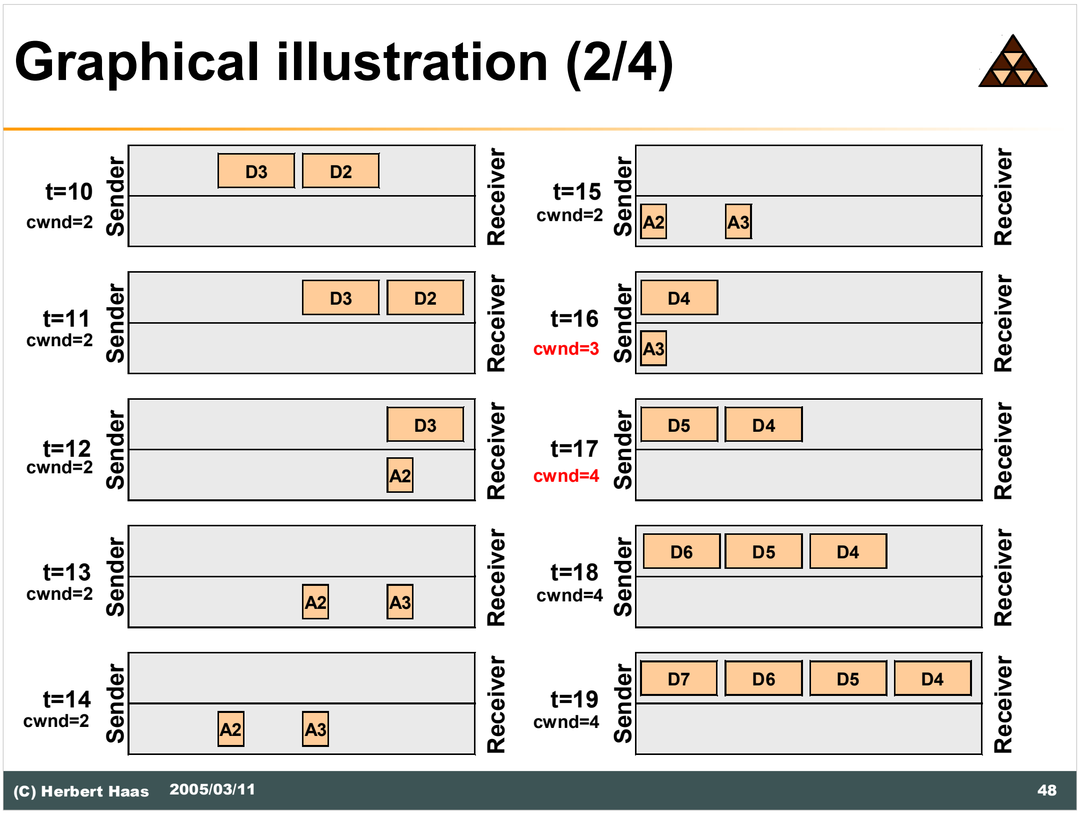


Graphical illustration (1/4)

(C) Herbert Haas 2005/03/11 47

The picture shows the two unidirectional channels between sender and receiver as pipe representation.

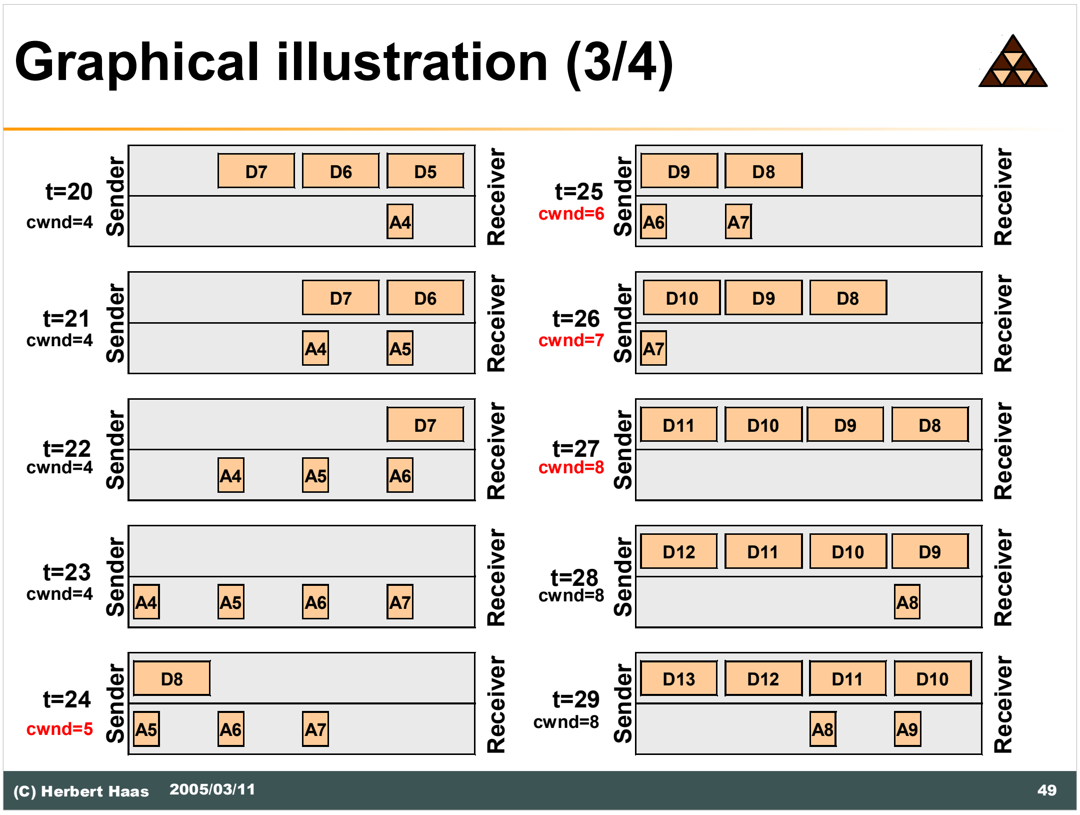
Observe how the cwnd is increased upon reception of ACKs.



Graphical illustration (2/4)

(C) Herbert Haas 2005/03/11 48

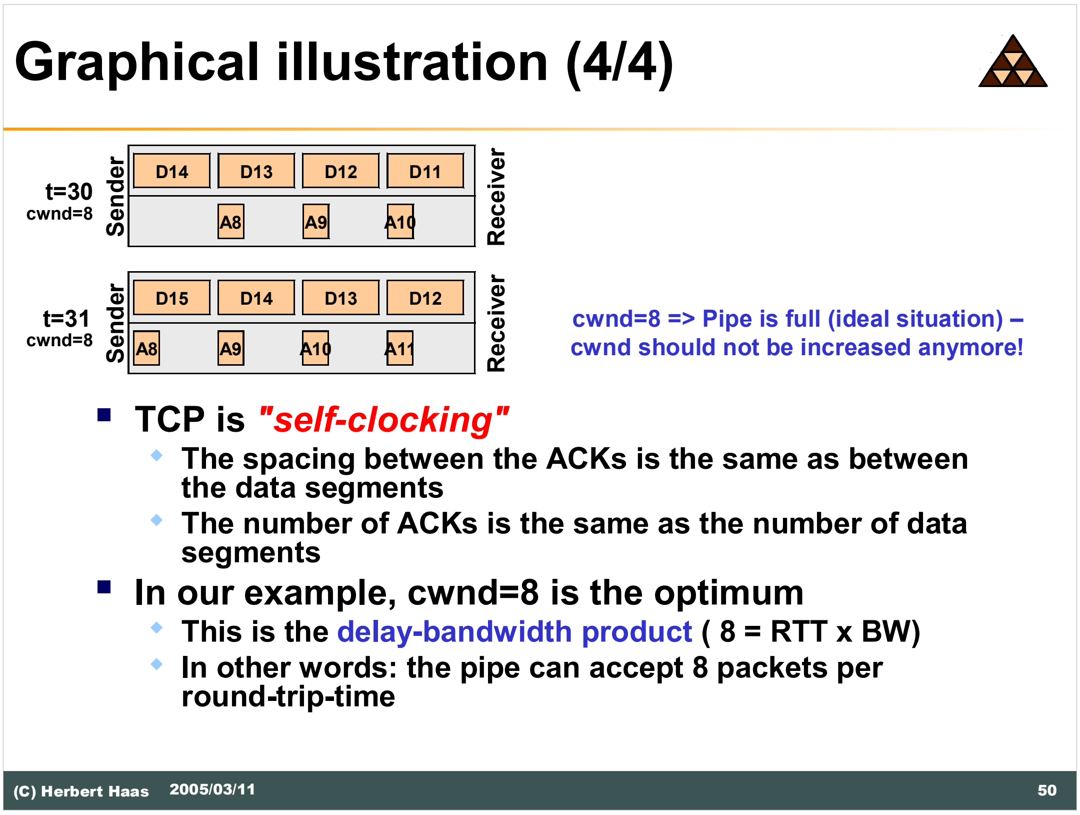
Observe the exponential growth of the data rate.



Graphical illustration (3/4)

(C) Herbert Haas 2005/03/11 49

We are approaching the limit soon…



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

At t=31, the pipe is ideally filled with packets; each time an ACK is received, another data packet is injected for transmission.

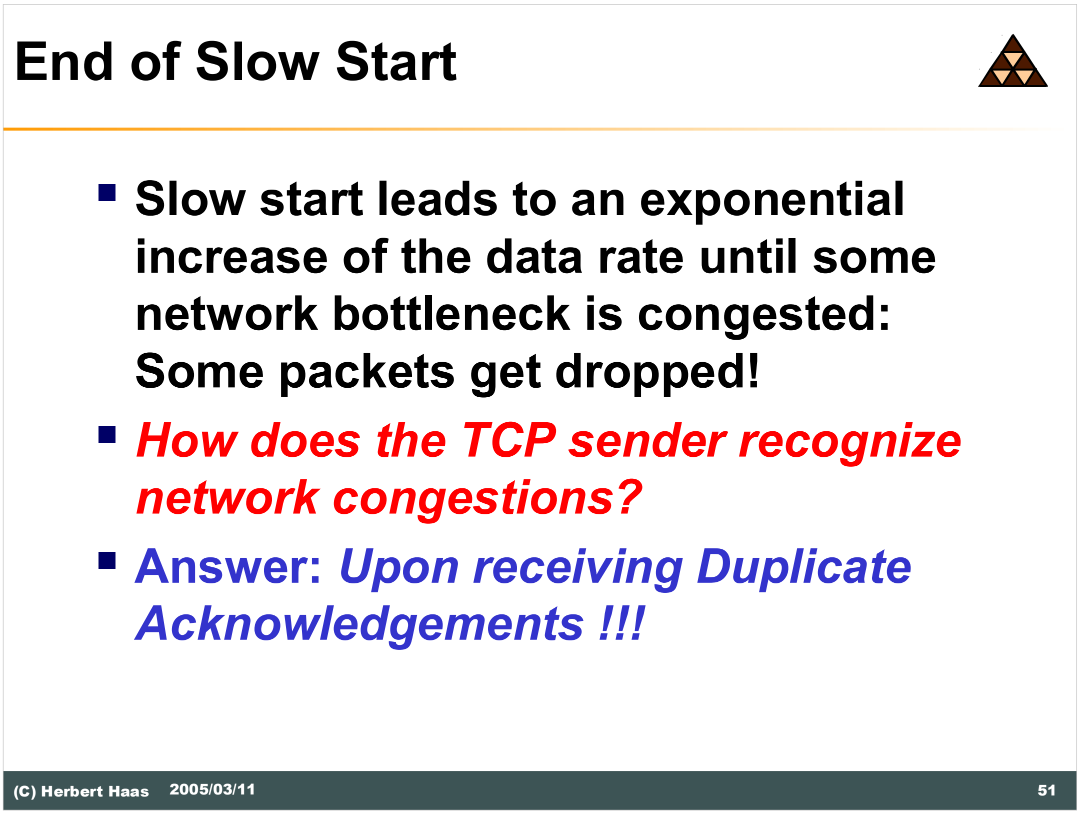
In our example cwnd=8 is the optimimum, corresponding to 8 packets that can be sent before waiting for an acknowledgement. This optimum is expressed via the famous delay-bandwidth product, i. e.

pipe capacity = RTT x BW ,

where the capacity is measured in bits, RTT in seconds, and the BW in bits/sec.

Our problem now is how to stop TCP from further increasing the cwnd… (continue reading).

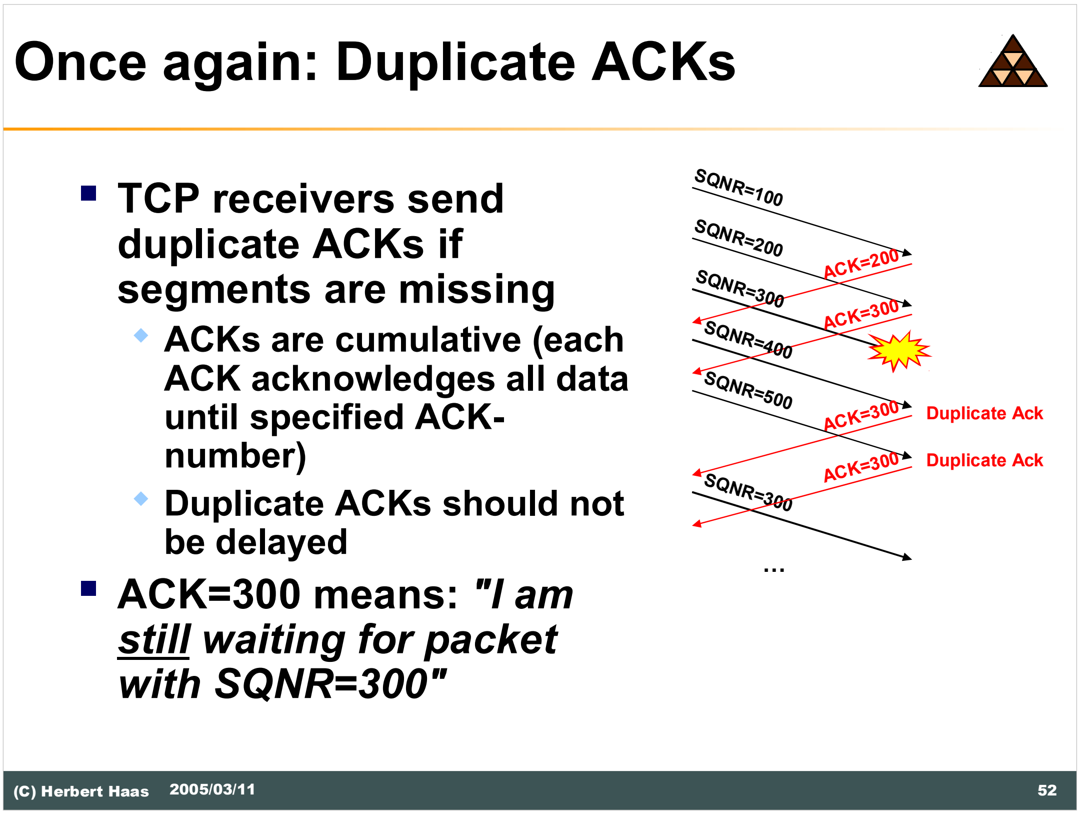
(BTW: Of course this illustration is not completely realistic because the spacing between the packets is distorted by many packet buffers along the path.)



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 !!!*

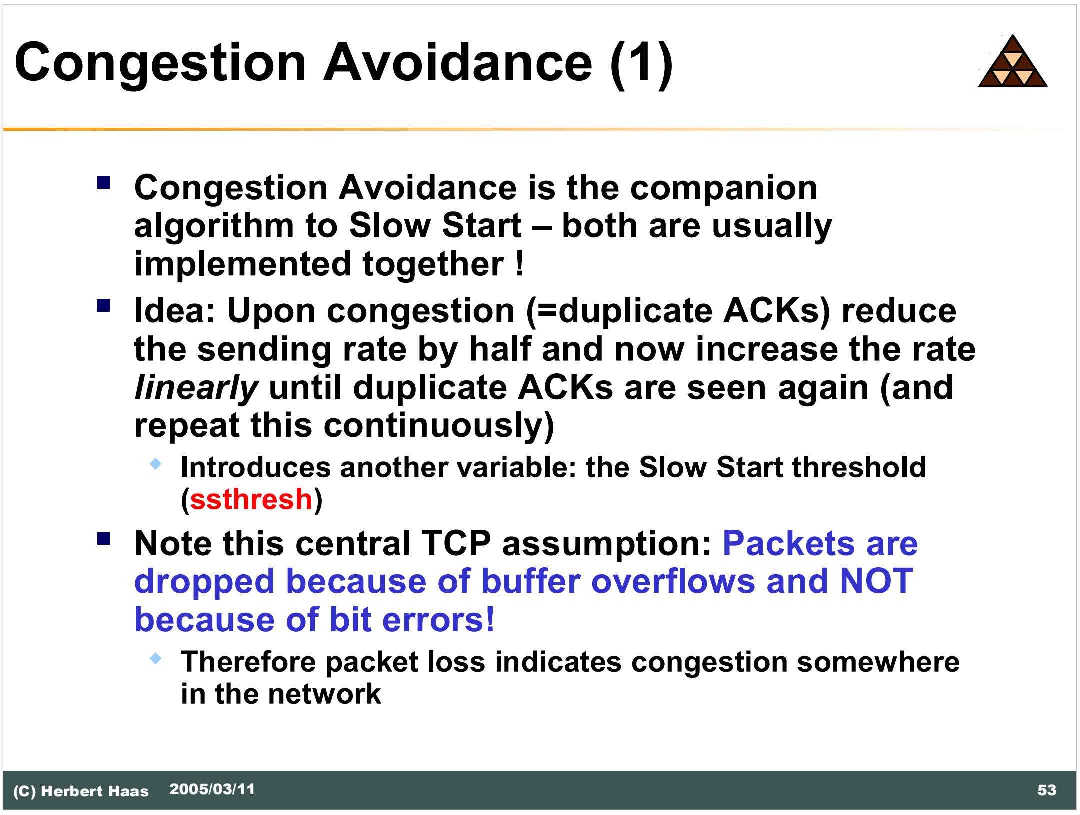
Slow start ends its exponential increase until duplicate acknowledgements are received.



Once again: Duplicate ACKs

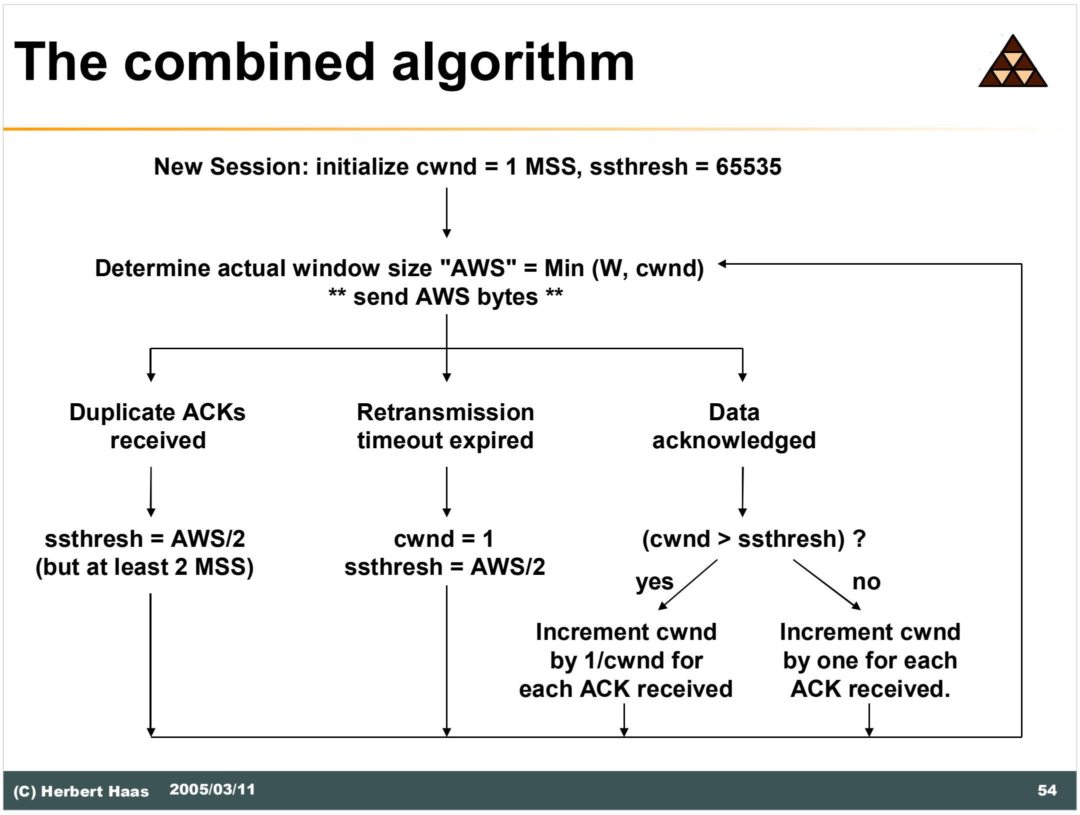
- TCP receivers send duplicate ACKs if segments are missing
  - ACKs are cumulative (each ACK acknowledges all data until specified ACK-number)
  - Duplicate ACKs should not be delayed
- ACK=300 means: *"I am <u>still</u> waiting for packet with SQNR=300"*

Duplicate ACKs should be sent immediately that is it should not be delayed.



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

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

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

Duplicate ACKs received

ssthresh = AWS/2
(but at least 2 MSS)

Retransmission timeout expired

cwnd = 1
ssthresh = AWS/2

Data acknowledged

(cwnd > ssthresh) ?

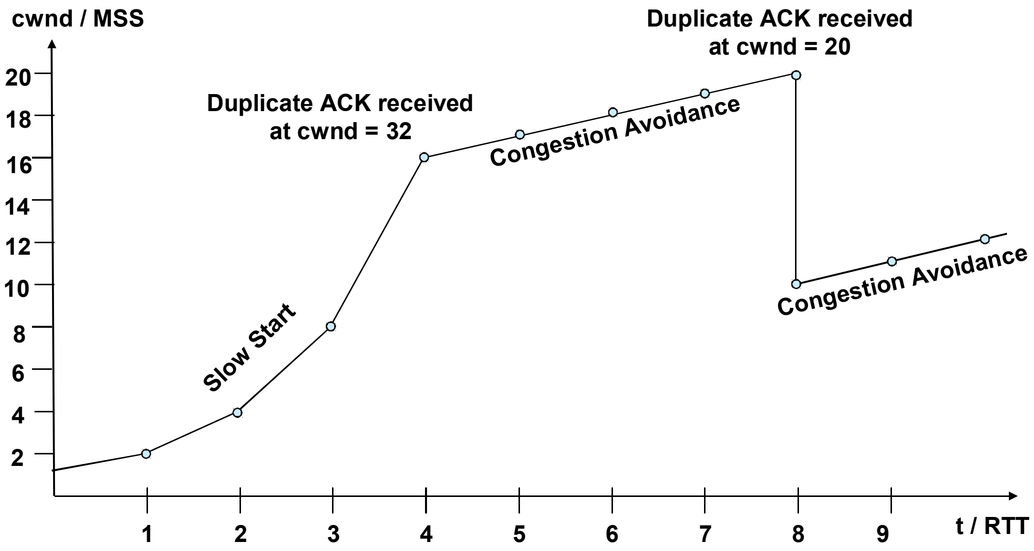
Increment cwnd by 1/cwnd for each ACK received

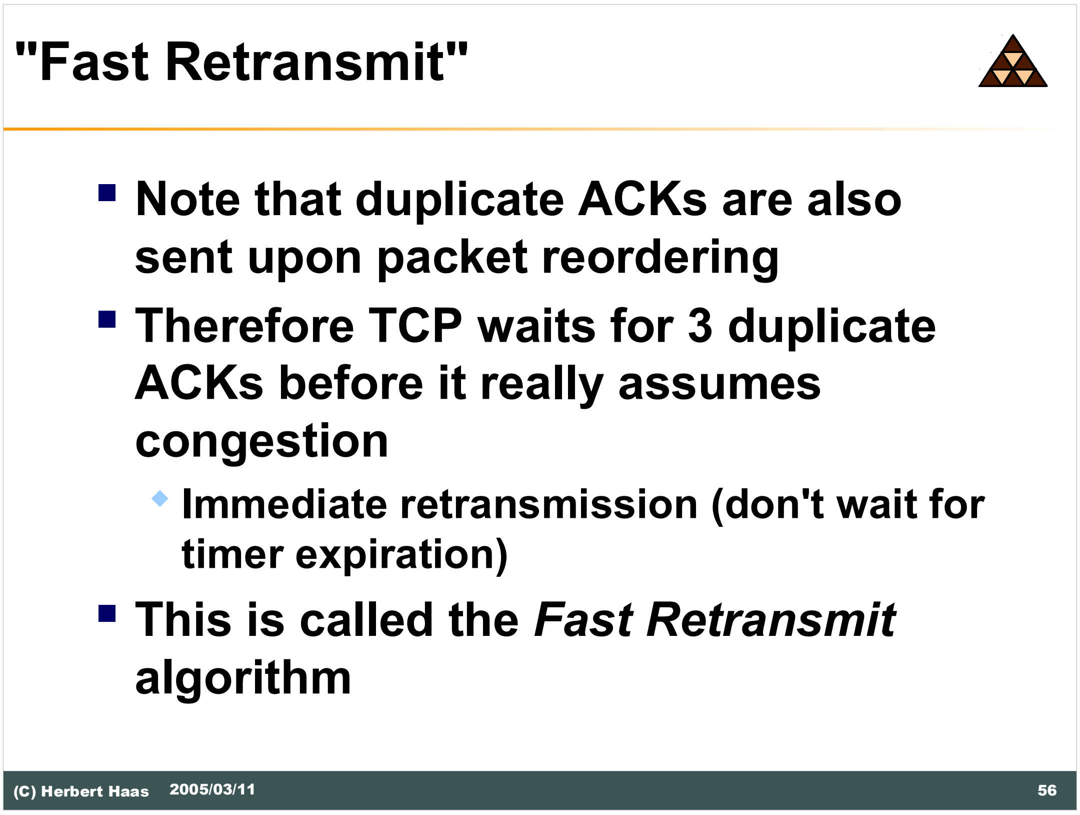
Increment cwnd by one for each ACK received.

Note that when slow start's exponential increase is only performed as long as cwnd is less or equal ssthresh. In this range, cwnd is increased by one with every received ACK. But if cwnd is greater than ssthresh, then cwnd is increased by 1/cwnd every received ACK. This means, cwnd is effectively increased by one every RTT.

Note that is not the complete algorithm. We must additionally discuss Fast Retransmit and Fast Recovetry—see next slides.

Slow Start and Congestion Avoidance

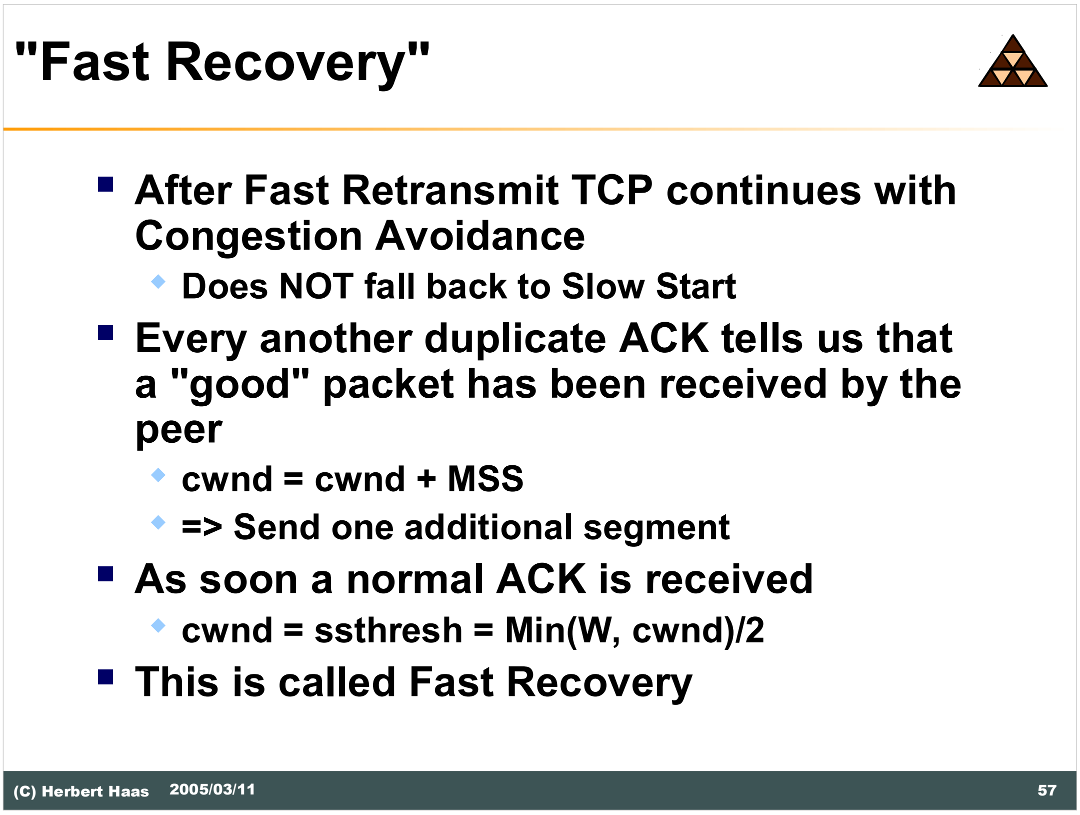




"Fast Retransmit"

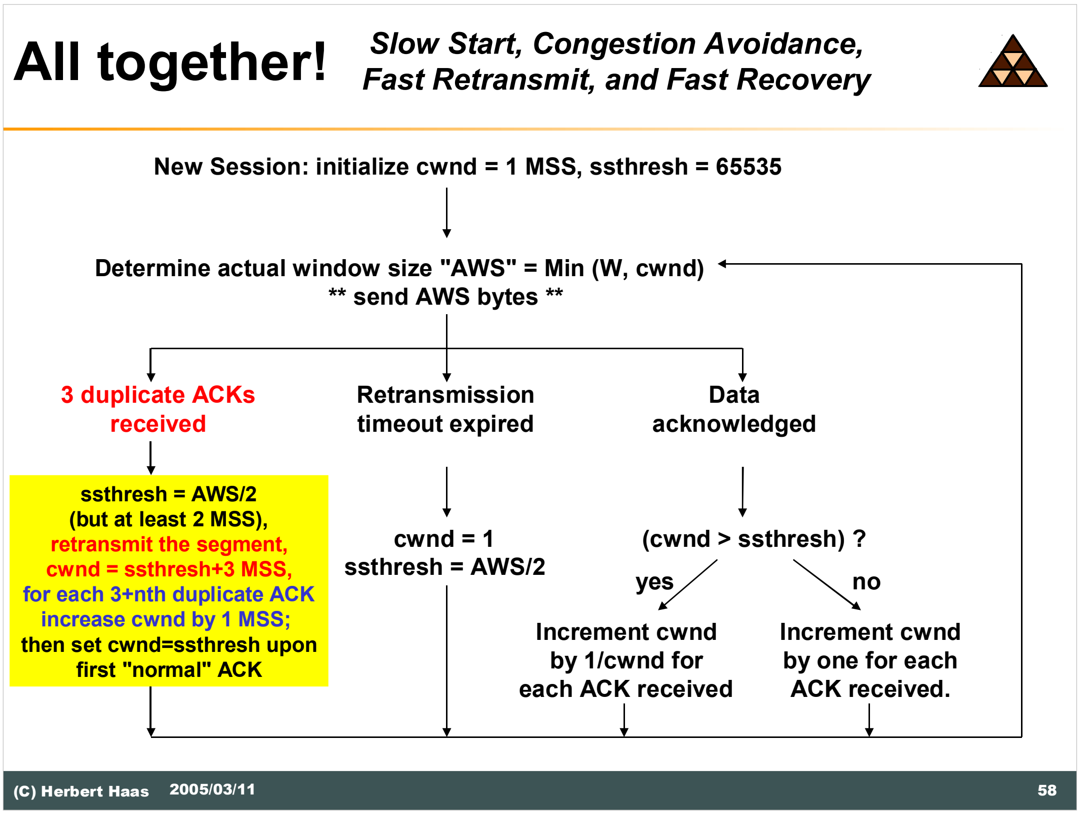
- Note that duplicate ACKs are also sent upon packet reordering
- Therefore TCP waits for 3 duplicate ACKs before it really assumes congestion
  - Immediate retransmission (don't wait for timer expiration)
- This is called the *Fast Retransmit* algorithm

Observations have shown that if three or more duplicate acks are sent then this is a strong indication for a lost packet. In this case Fast Retransmission is done, i. e. TCP does not wait until a packet's retransmission timer expires.



"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



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) ?

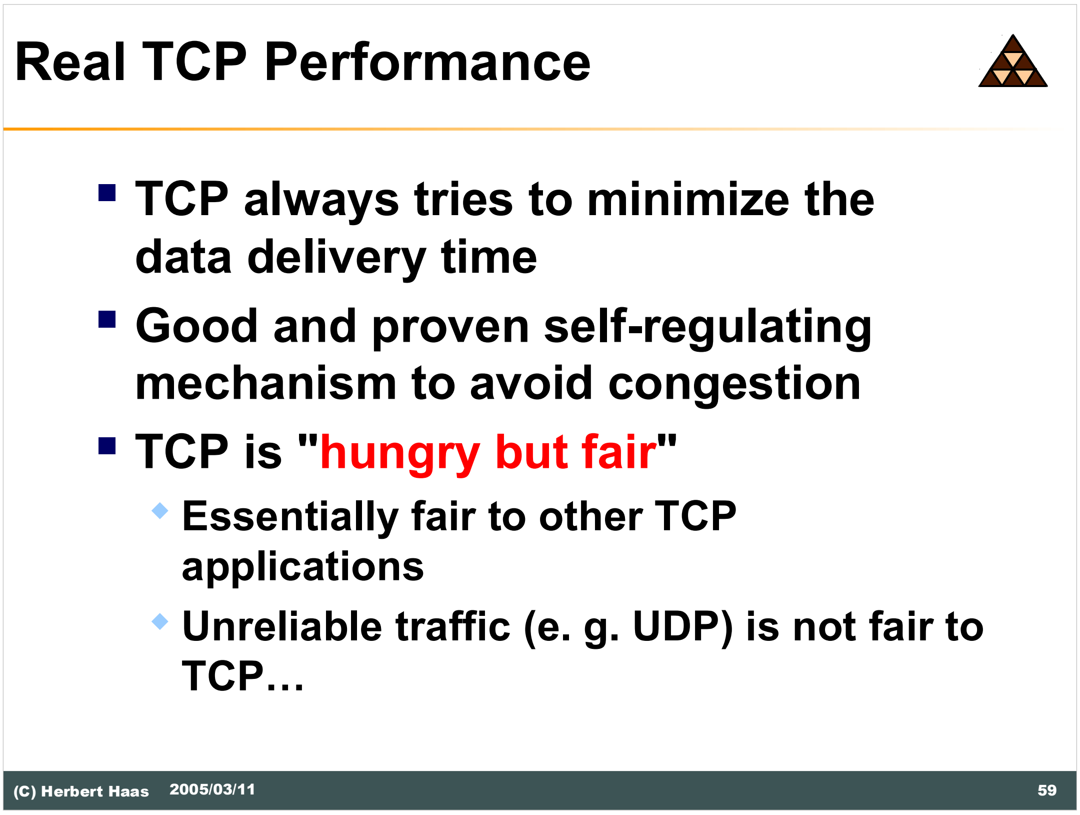
yes

Increment cwnd by 1/cwnd for each ACK received

no

Increment cwnd by one for each ACK received.

When one or two duplicate ACKs are received, TCP does not react because packet reorder is probable. Upon the third duplicate ACK TCP assumes that the segment (for which the duplicate ACK is meant) is really lost. TCP now immediately retransmit the packet (i. e. it does not wait for any timer expiration), sets ssthresh to min{W, cwnd}/2 and then cwnd three segment sizes greater than this ssthresh value. If TCP still receives duplicate ACKs then obviously good packets still arrive at the peer; and therefore TCP continous sending new segements—hereby incrementing cwnd by one segment size for every another duplicate ACK (this actually allows the transmission of another new segment). As soon as a normal (=not duplicate) ACK is received (=it acknowledges the retransmitted segment) cwnd is set to ssthresh (=continue with normal congestion avoidance).

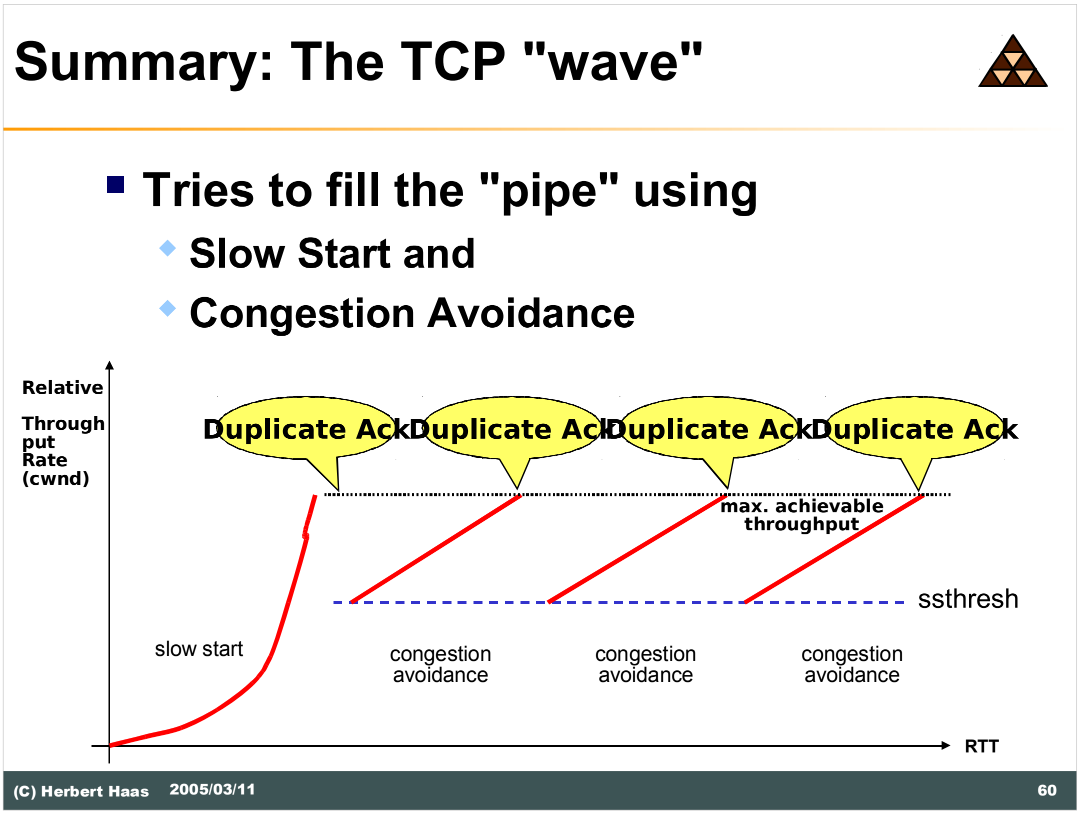


Real TCP Performance

- 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…

TCP has been designed for data traffic only. Error recovery does not make sense for voice and video streams. TCP checks the current maximum bandwidth and tries to utilize all of it. In case of congestion situations TCP will reduce the sending rate dramatically and explores again the network's capabilities. Because of this behavior TCP is called "hungry but fair".

The problem with this behavior is the consequence for all other types of traffic: TCP might grasp all it can get and nothing is left for the rest.



Summary: The TCP "wave"

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

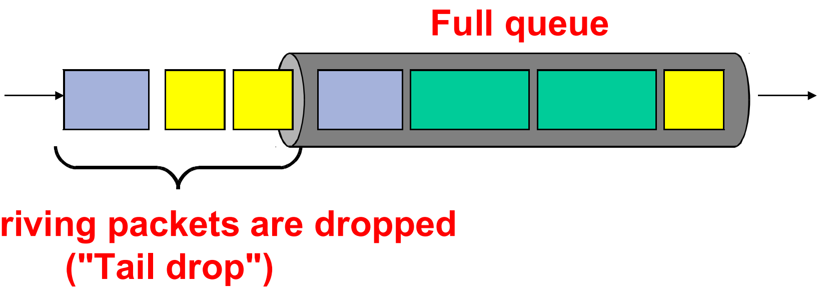
The diagram above shows the typical TCP behavior of one flow. There are two important algorithms involved with TCP congestion control: "**Slow Start**" increases the sending rate exponentially beginning with a very low sending rate (typically 1-2 segments per RTT). When the limit of the network is reached, that is, when duplicate acknowledgement occur, then "**Congestion Avoidance**" reduces the sending rate by 50 percent and then it is increased only linearly.

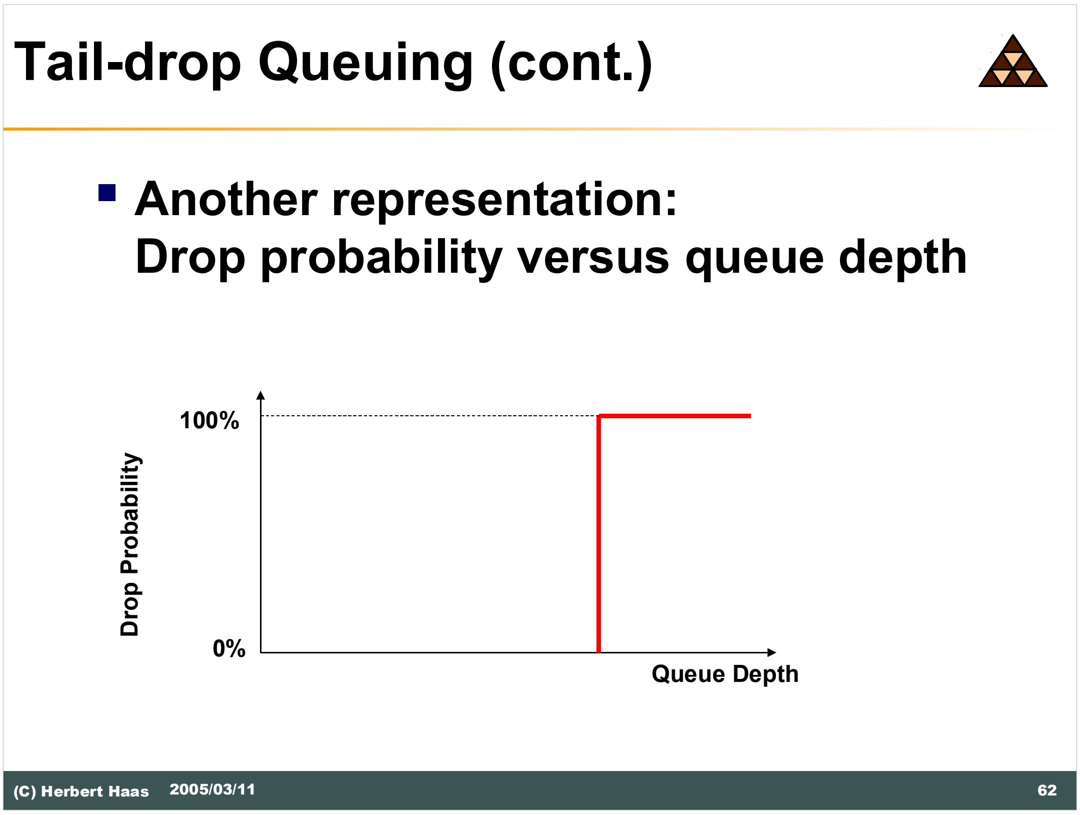
The rule is: On receiving a duplicate ACK, congestion avoidance is performed. On receiving no ACK at all, slow start is performed again, beginning at zero sending rate.

Note that this is only a quick and rough explanation of the two algorithms—the details are a bit more complicated. Furthermore, different TCP implementations utilize these algorithm differently.

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**

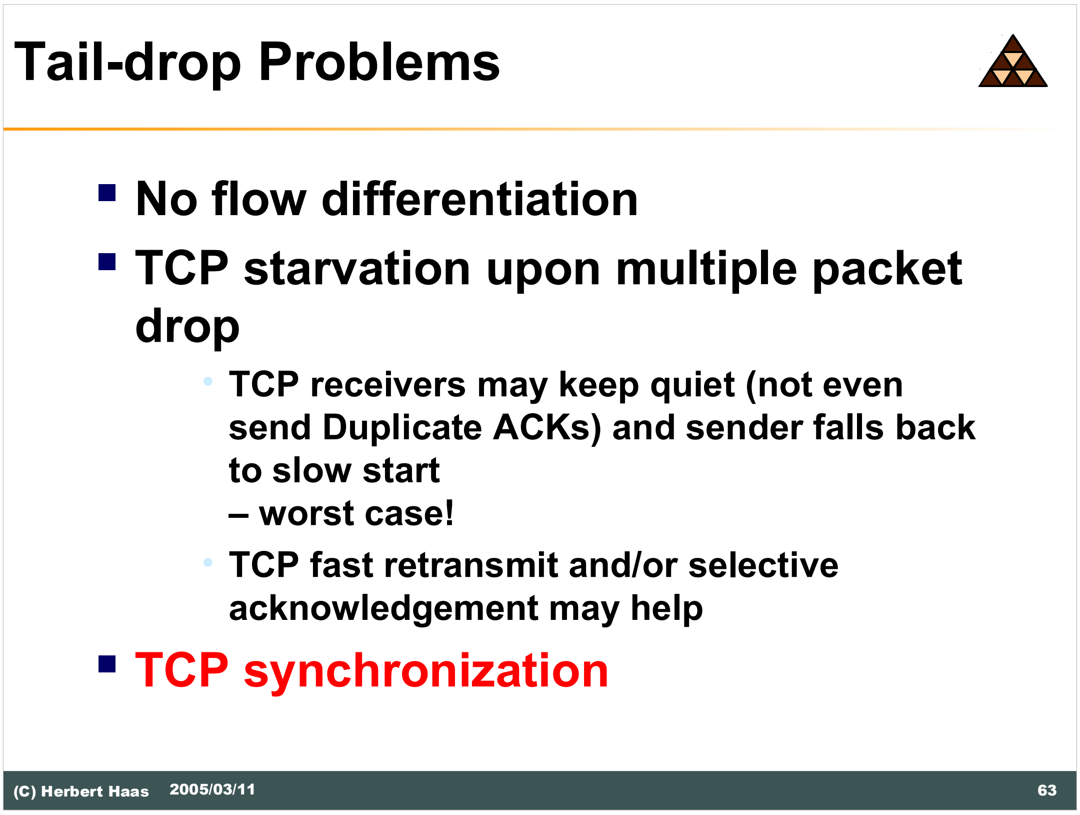




Tail-drop Queuing (cont.)

- Another representation:
  Drop probability versus queue depth

The "queue depth" denotes the amount of packets waiting in the queue for being forwarded. (It is NOT the size of the whole queue.)

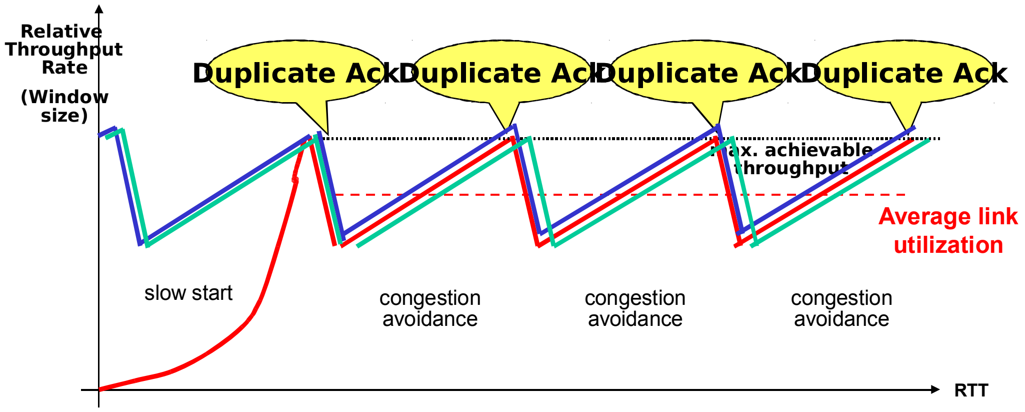


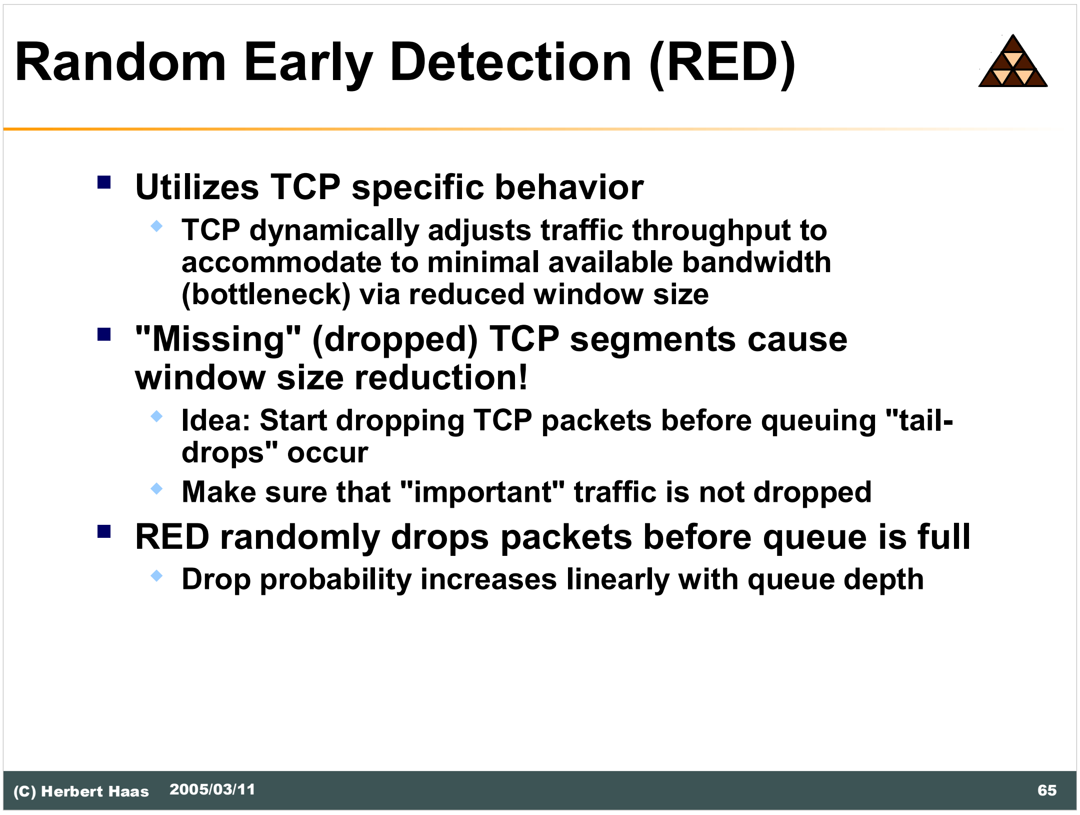
Tail-drop Problems

- 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

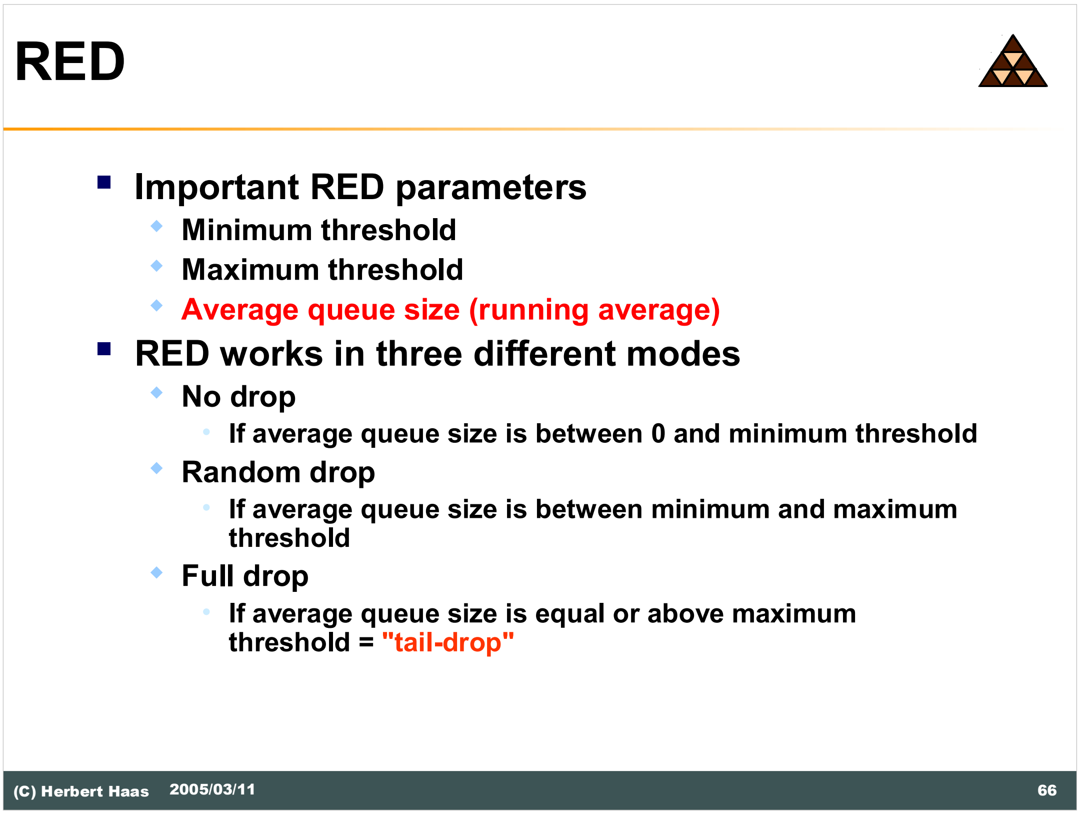
- Tail-drop drops many packets of different sessions at the same time
- All these sessions experience duplicate ACKs and perform synchronized congestion avoidance





Random Early Detection (RED)

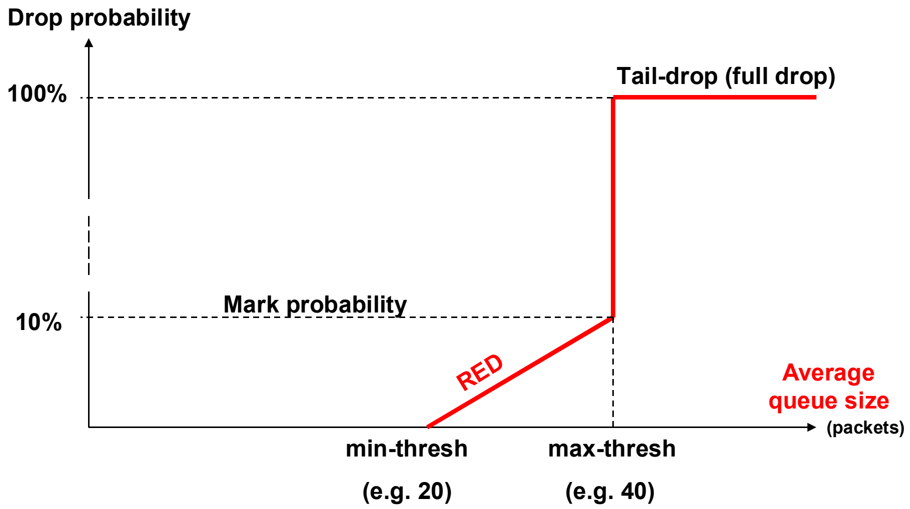
- Utilizes TCP specific behavior
  - TCP dynamically adjusts traffic throughput to accommodate to minimal available bandwidth (bottleneck) via reduced window size
- "Missing" (dropped) TCP segments cause window size reduction!
  - Idea: Start dropping TCP packets before queuing "tail-drops" occur
  -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



RED

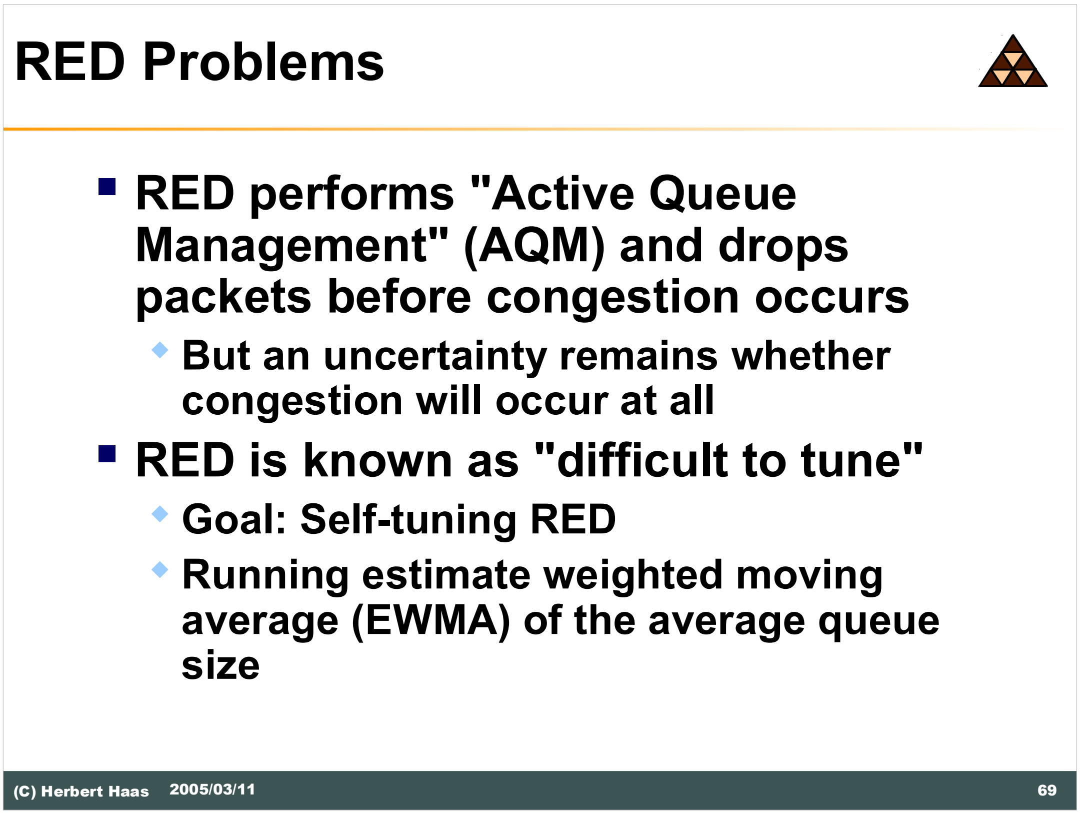
- Important RED parameters
  - Minimum threshold
  - Maximum threshold
  - Average queue size (running average)
- RED works in three different modes
  - No drop
    - If average queue size is between 0 and minimum threshold
  - Random drop
    - If average queue size is between minimum and maximum threshold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

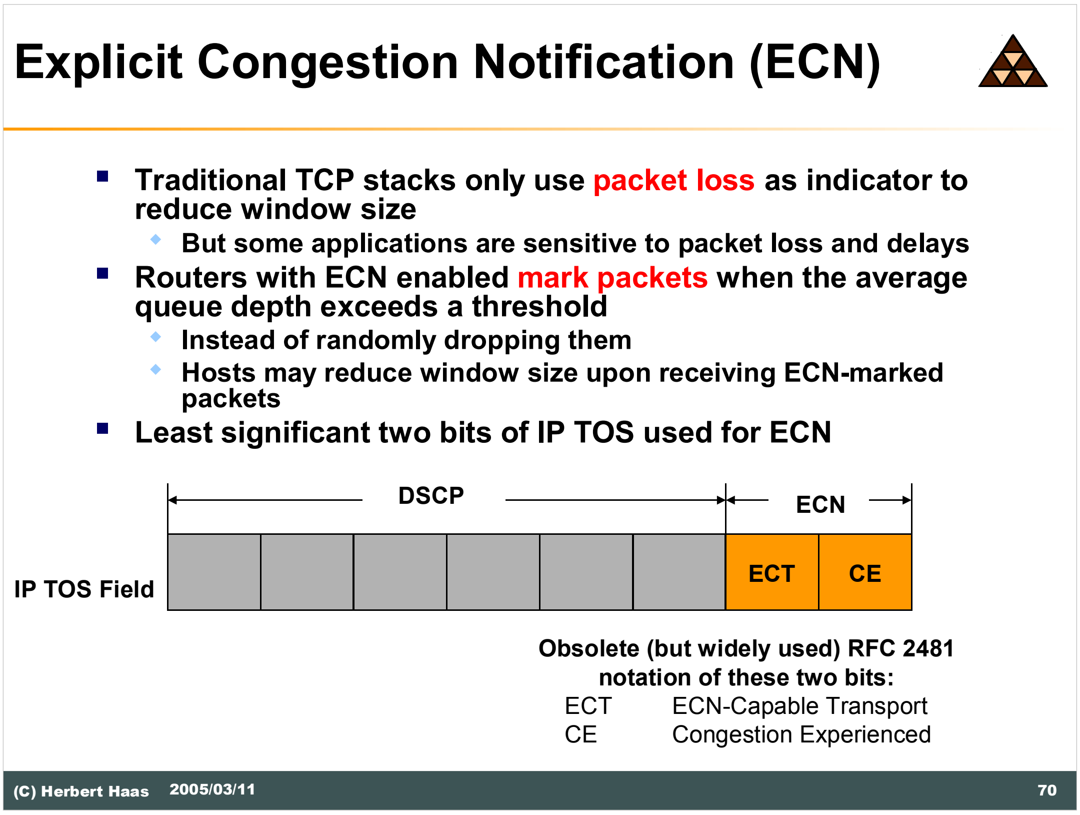
(C) Herbert Haas 2005/03/11 69

Many TCP streams in a network tend to synchronize each other in terms of intensity. That is, all TCP users recognize congestion simultaneously and would restart the slow-start process (sending at a very low rate). At this moment the network is not utilized. After a short time, all users would reach the maximum sending rate and network congestion occurs. At this time all buffers are full. Again all TCP users will stop and nearly stop sending again. This cycle continues infinitely and is called the TCP wave effect. The main disadvantage is the relatively low utilization of the network.

Random Early Discard (RED) is a method to de-synchronize the TCP streams by simply drop packets of a queue randomly. RED starts when a given queue depth is reached and is applied more aggressively when the queue depth increases.

RED causes the TCP receivers to send duplicate ACKs which in turn causes the TCP senders to perform congestion avoidance. The trick is that this happens randomly, so not all TCP applications are affected equally at the same time.

Although the principle of RED is fairly simply it is known to be difficult to tune. A lot of research has been done to find out optimal rules for RED tuning.



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

DSCP

ECN

IP TOS Field

ECT

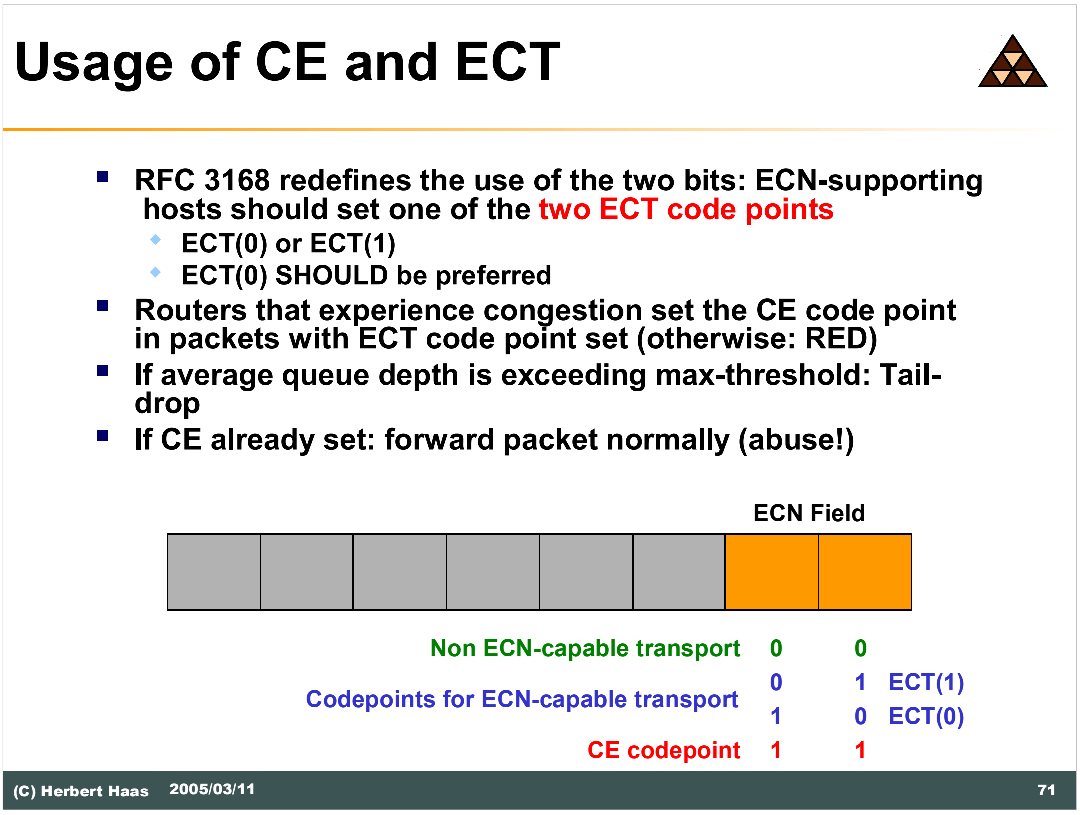
CE

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

ECT	ECN-Capable Transport
CE	Congestion Experienced

RFC 3168 - The Addition of Explicit Congestion Notification (ECN) to IP

The RFC 2481 originally identified the two bits: "The ECN-Capable Transport (ECT) bit would be set by the data sender to indicate that the end-points of the transport protocol are ECN-capable. The CE bit would be set by the router to indicate   congestion to the end nodes. Routers that have a packet arriving at a full queue would drop the packet, just as they do now."



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	

Why are two ECT codepoints used? As short answer: This has several reasons and supports multiple implementations, e. g. to differentiate between different sets of hosts etc.

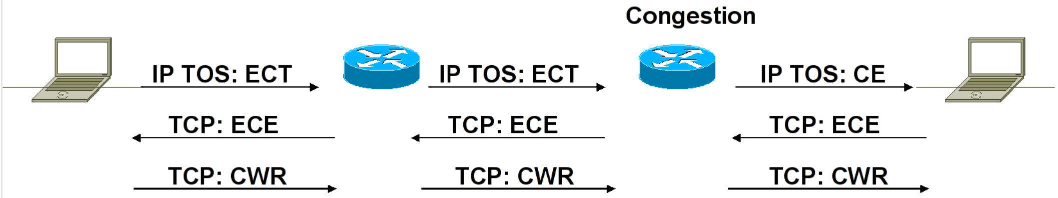
But the most important reason is to provide a mechanism so that a host (or a router) can check whether the network (or the host, respectively) indeed supports ECN. ECN has been introduced in the mid-1990s and the inventors wanted to increase the pressure for hists and routers to migrate. On the other hand non-ECN hosts could simply set the ECT-bit (see previous slide) and claimed to support ECN: Upon congestion the router would not drop the packet but only mark it. While ECN-capable host would reduce their TCP window, ECN-faking hosts would still remain at their transmission rate. Now the two ECT Codepoints could be used as Cookie which allows a host to detect whether a router erases the ECT or ECN bit. Also it can be tested whether the other side uses ECN.

If you do not fully understand this please read the RFCs and search in the WWW – there a lots of debates about that.

By the way: The bit combination 01 indeed stands for ECT(1) and not ECN(0). This is no typo.

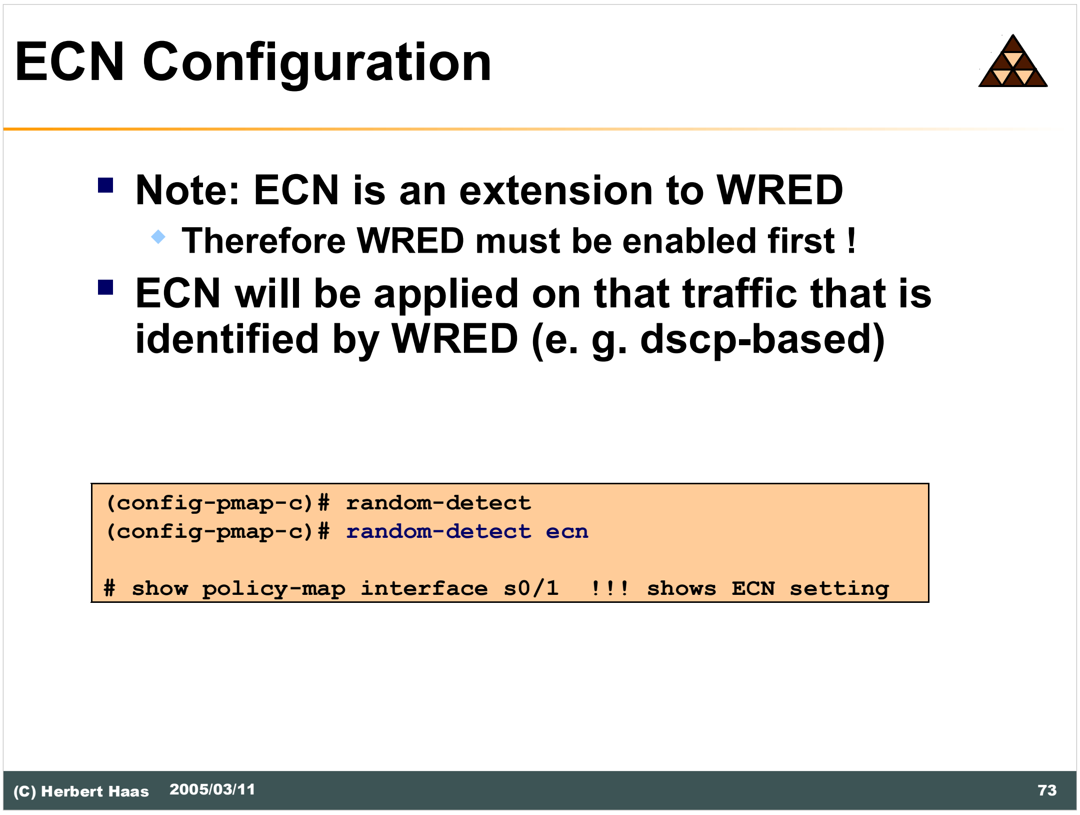
CWR and ECE

- 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

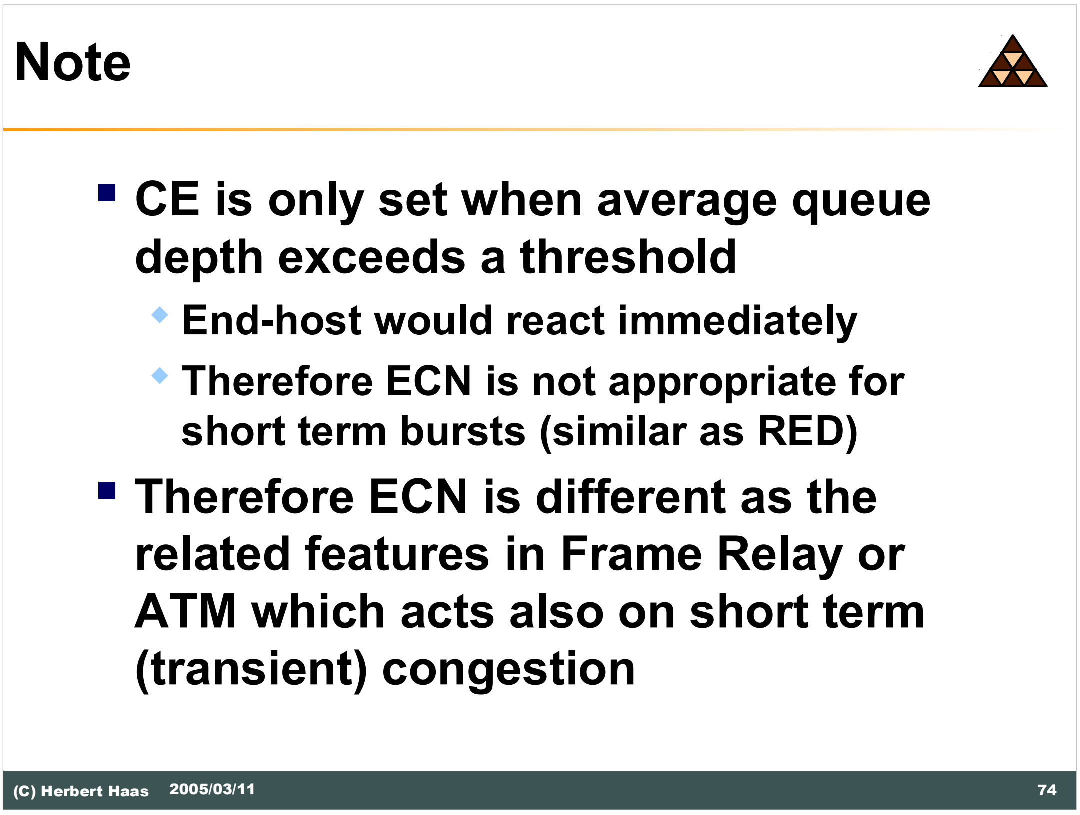


ECN Configuration

- 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)

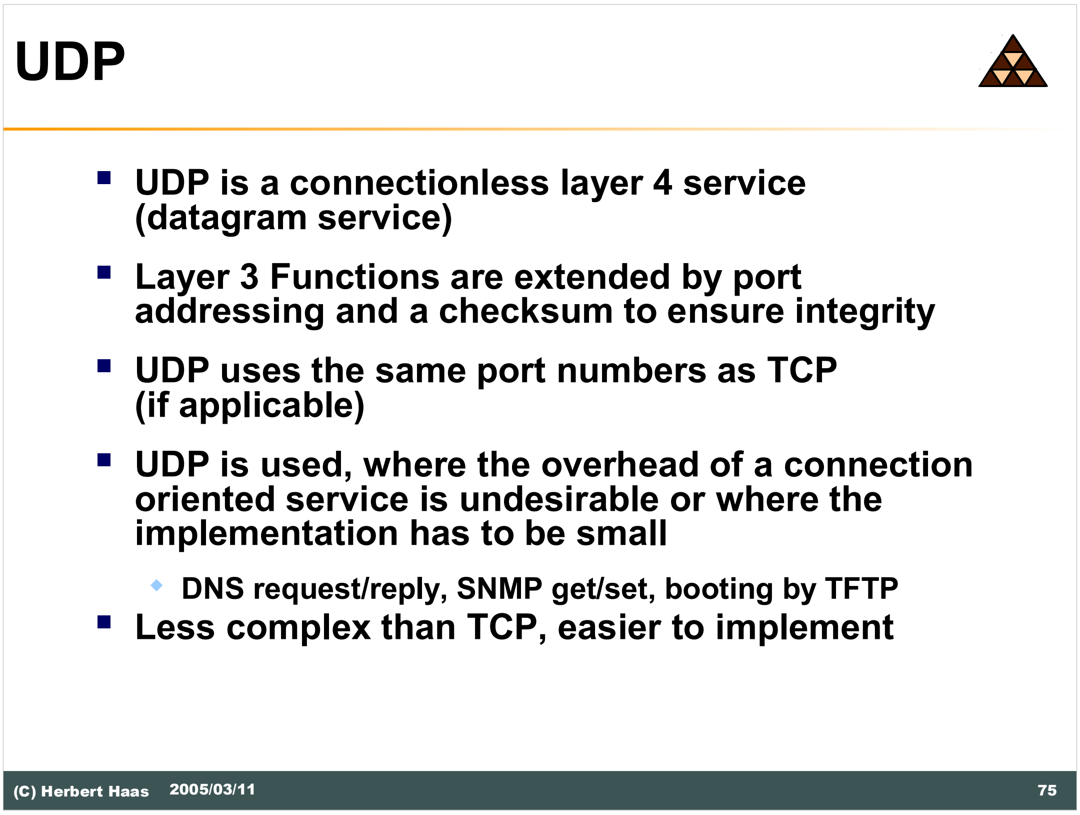
```
(config-pmap-c)# random-detect
(config-pmap-c)# random-detect ecn

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

If ECN is enabled, ECN can be used whether Weighted Random Early Detection (WRED) is based on the IP precedence value or the differentiated services code point (DSCP) value.



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

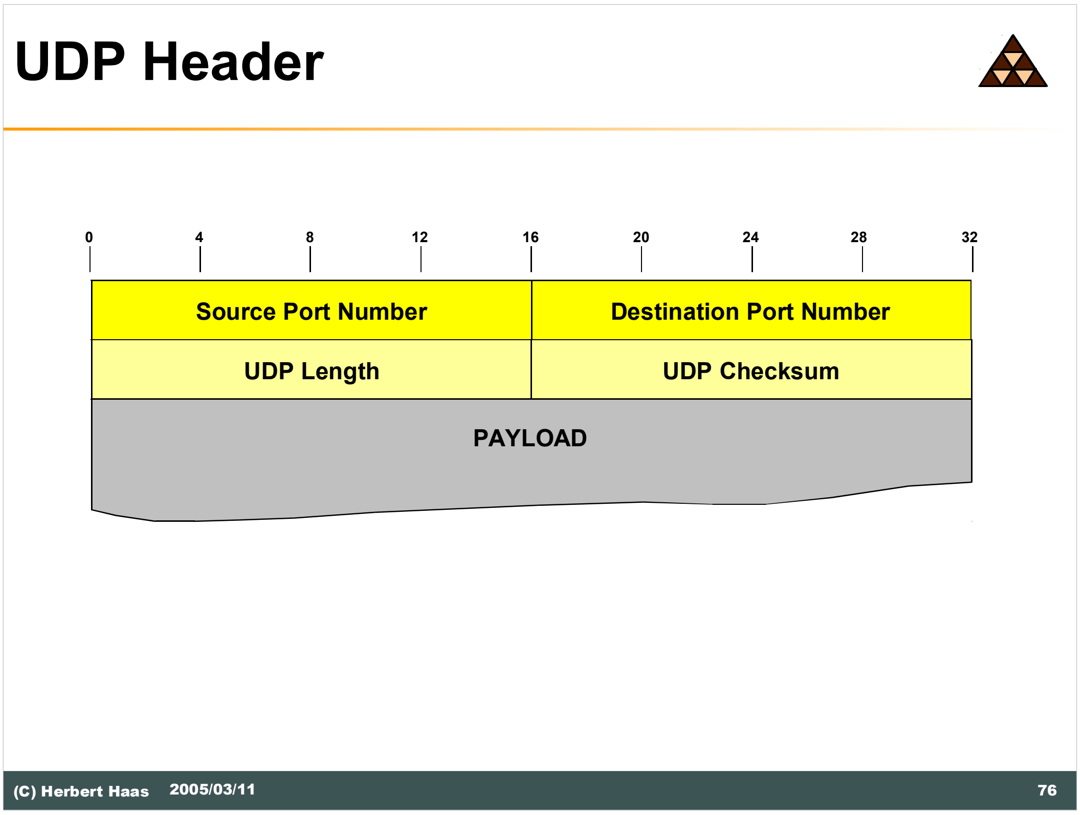


UDP

- 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 is connectionless and supports no error recovery or flow control. Therefore an UDP-stack is extremely lightweight compared to TCP.

Typically applications that do not require error recovery but rely on speed use UDP, such as multimedia protocols.



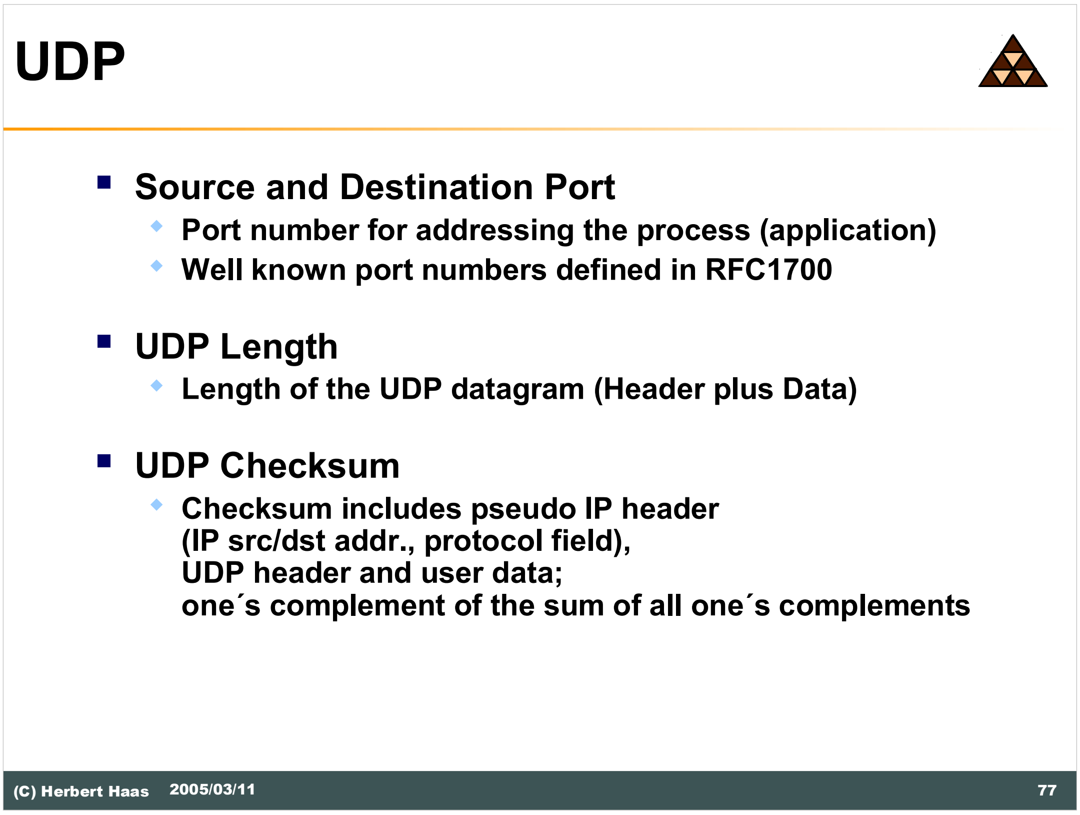
UDP Header

Source Port Number	Destination Port Number
UDP Length	UDP Checksum

PAYLOAD

The picture above shows the 8 byte UDP header. Note that the Checksum is often not calculated, so UDP basically carries only the port numbers.

I personally think that the length field is just for fun (or to align with 4 octets). The IP header already contains the total packet length.

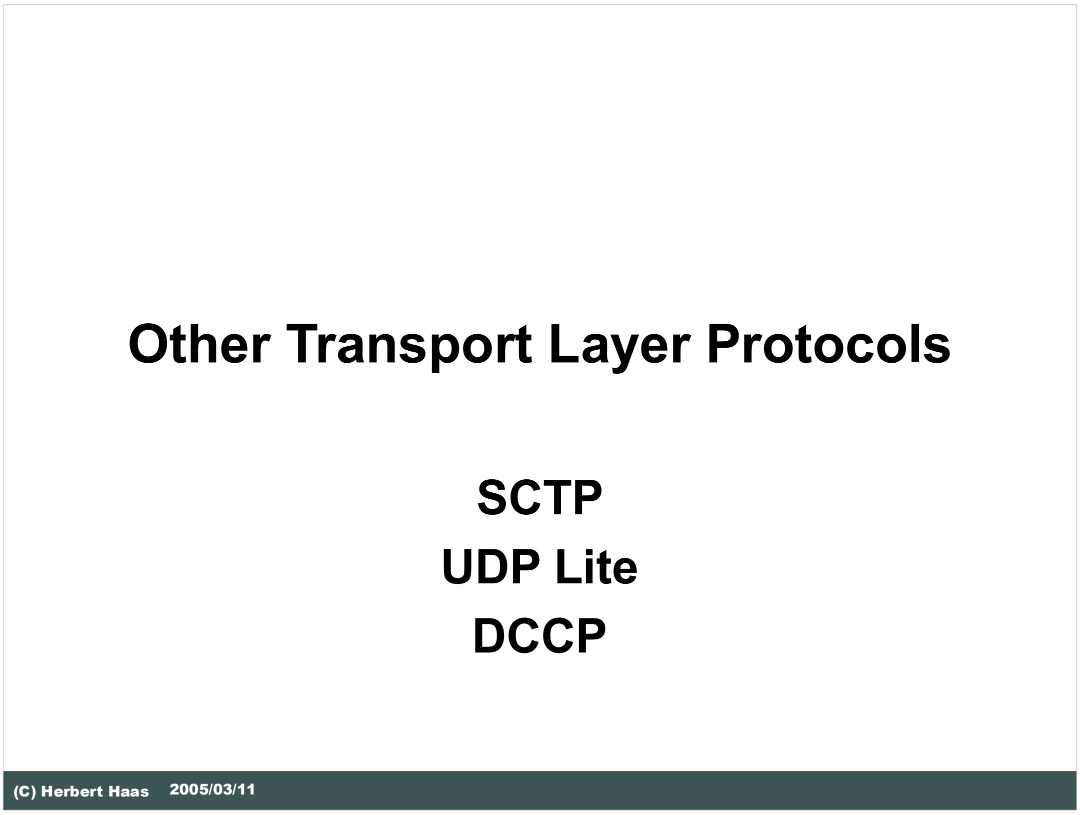


UDP

- **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

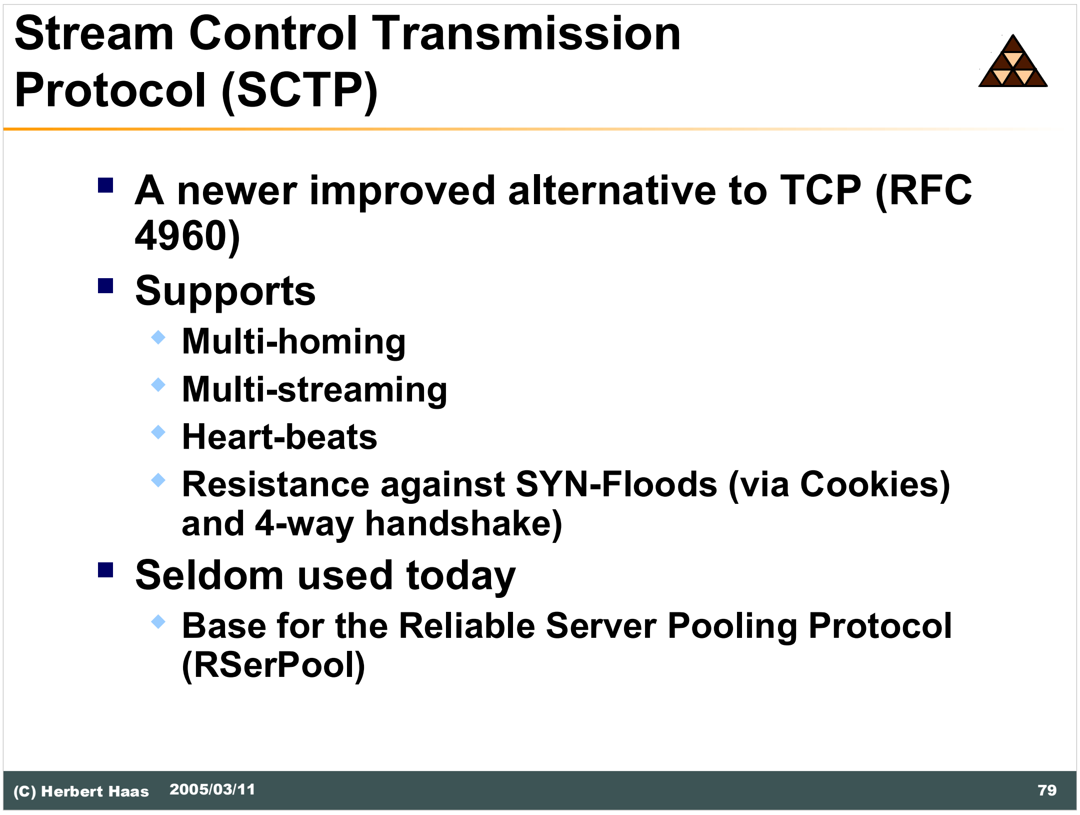
(C) Herbert Haas 2005/03/11 77

Compared to the TCP Header, the UDP is very small (8 byte to 20 byte) because UDP makes no error recovery or flow control.



Other Transport Layer Protocols

SCTP
UDP Lite
DCCP

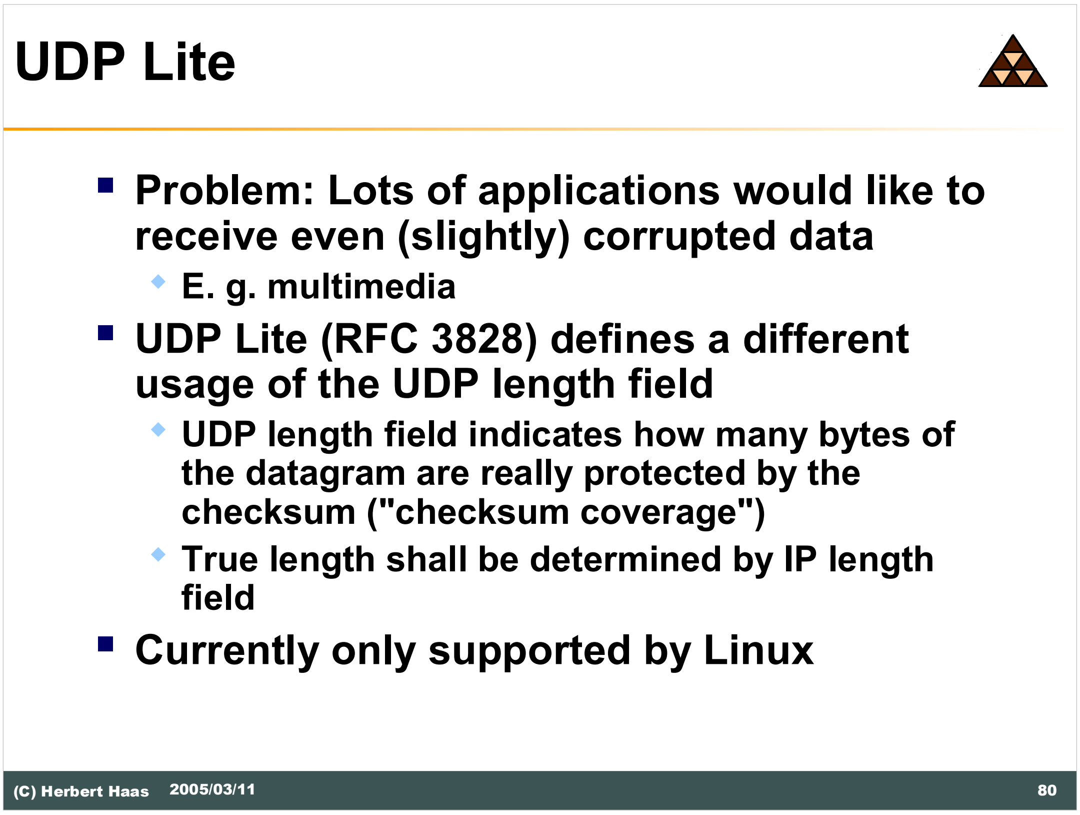


Stream Control Transmission Protocol (SCTP)

- 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)

Invented around 2000 it has not found wide acceptance today although there is a growing community behind it.

Multi-homing means that endpoints may consist of more than one IP address, i. e. a session may involve multiple interfaces per host.



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

(C) Herbert Haas 2005/03/11 80

Why rejecting big UDP datagrams when 99% of the payload is still useful?

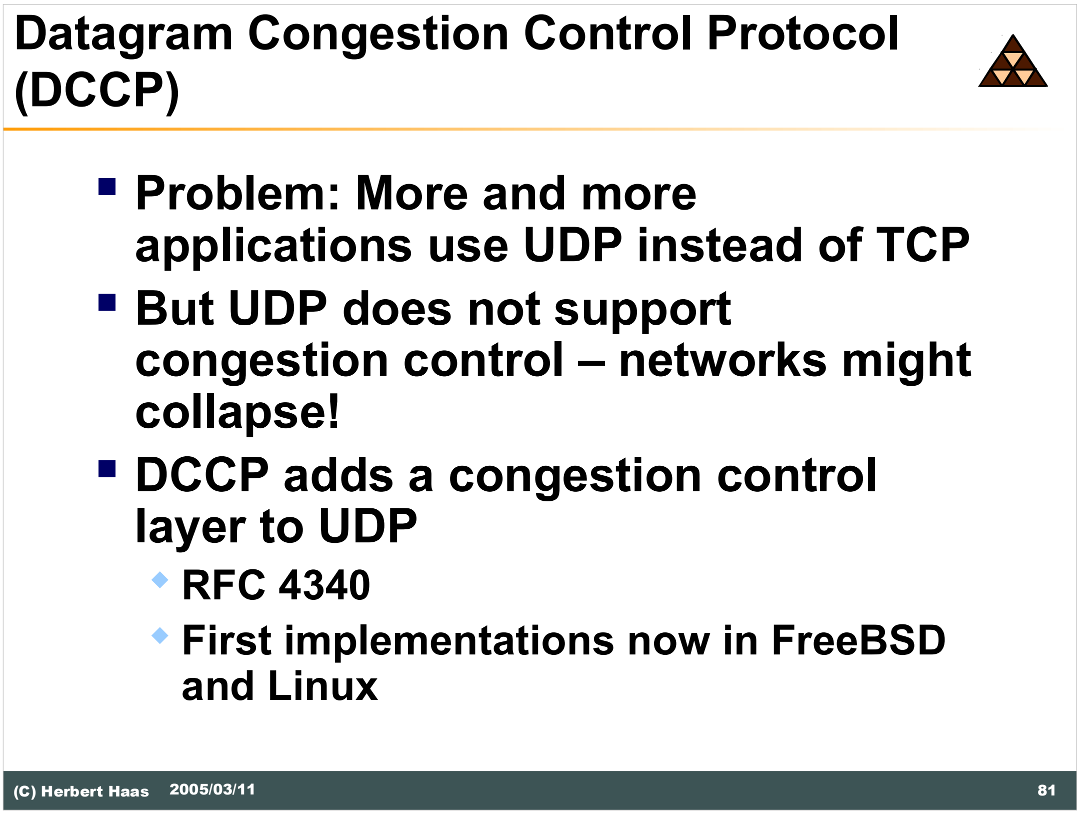
As stated in RFC 3828:

This new protocol is based on three observations: First, there is a class of applications that benefit from having damaged data delivered rather than discarded by the network. A number of codecs for voice and video fall into this class (e.g., the AMR speech codec [RFC-3267], the Internet Low Bit Rate Codec [ILBRC], and error resilient H.263+ [ITU-H.263], H.264 [ITU-H.264; H.264], and MPEG-4 [ISO-14496] video codecs). These codecs may be designed to cope better with errors in the payload than with loss of entire packets.

Second, all links that support IP transmission should use a strong link layer integrity check (e.g., CRC-32 [RFC-3819]), and this MUST be used by default for IP traffic. When the under-lying link supports it, certain types of traffic (e.g., UDP-Lite) may benefit from a different link behavior that permits partially damaged IP packets to be forwarded when requested [RFC-3819]. Several radio technologies (e.g., [3GPP]) support this link behavior when operating at a point where cost and delay are sufficiently low. If error-prone links are aware of the error sensitive portion of a packet, it is also possible for the physical link to provide greater protection to reduce the probability of corruption of these error sensitive bytes (e.g., the use of unequal Forward Error Correction).

A length field of zero means the whole UDP datagram is covered by the checksum. At least the header must be protected, that is the length field is either 0 or at least 8. It is required that the IP-pseudoheader is always part of the checksum computation.

UDP Lite is supported by Linux since kernel 2.6.20.



Datagram Congestion Control Protocol (DCCP)

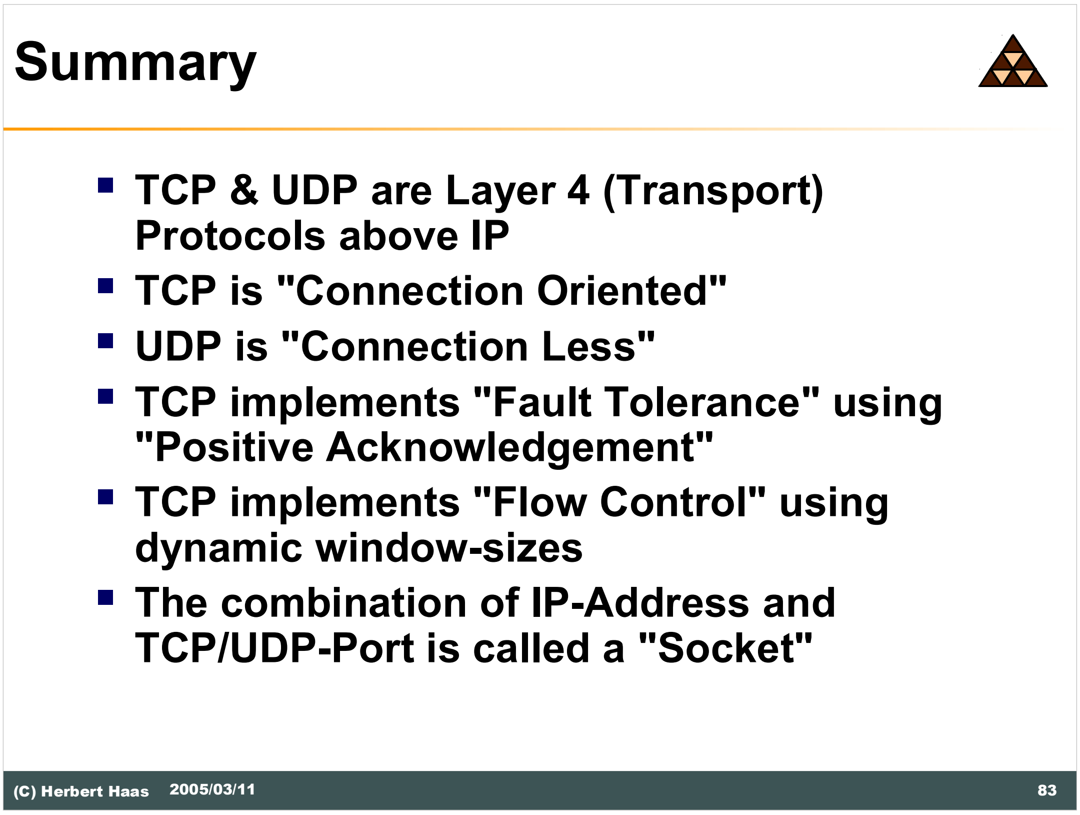
- Problem: More and more applications use UDP instead of TCP
- But UDP does not support congestion control – networks might collapse!
- DCCP adds a congestion control layer to UDP
  - RFC 4340
  - First implementations now in FreeBSD and Linux

DCCP (cont.)

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

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

<table>
<tr><td colspan="5">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="3">Data Offset</td><td>CcVal</td><td>CsCov</td><td>Checksum</td></tr>
<tr><td>Res</td><td>Packet Type</td><td>X = 1</td><td colspan="2">Reserved</td><td>Sequence Number (high bits)</td></tr>
<tr><td colspan="6">Sequence Number (low bits)</td></tr>
<tr><td colspan="5">Reserved</td><td>Acknowledge Number (high bits)</td></tr>
<tr><td colspan="6">Acknowledge Number (low bits)</td></tr>
<tr><td colspan="6">Options and Padding</td></tr>
<tr><td colspan="6">Application Data</td></tr>
</table>



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"