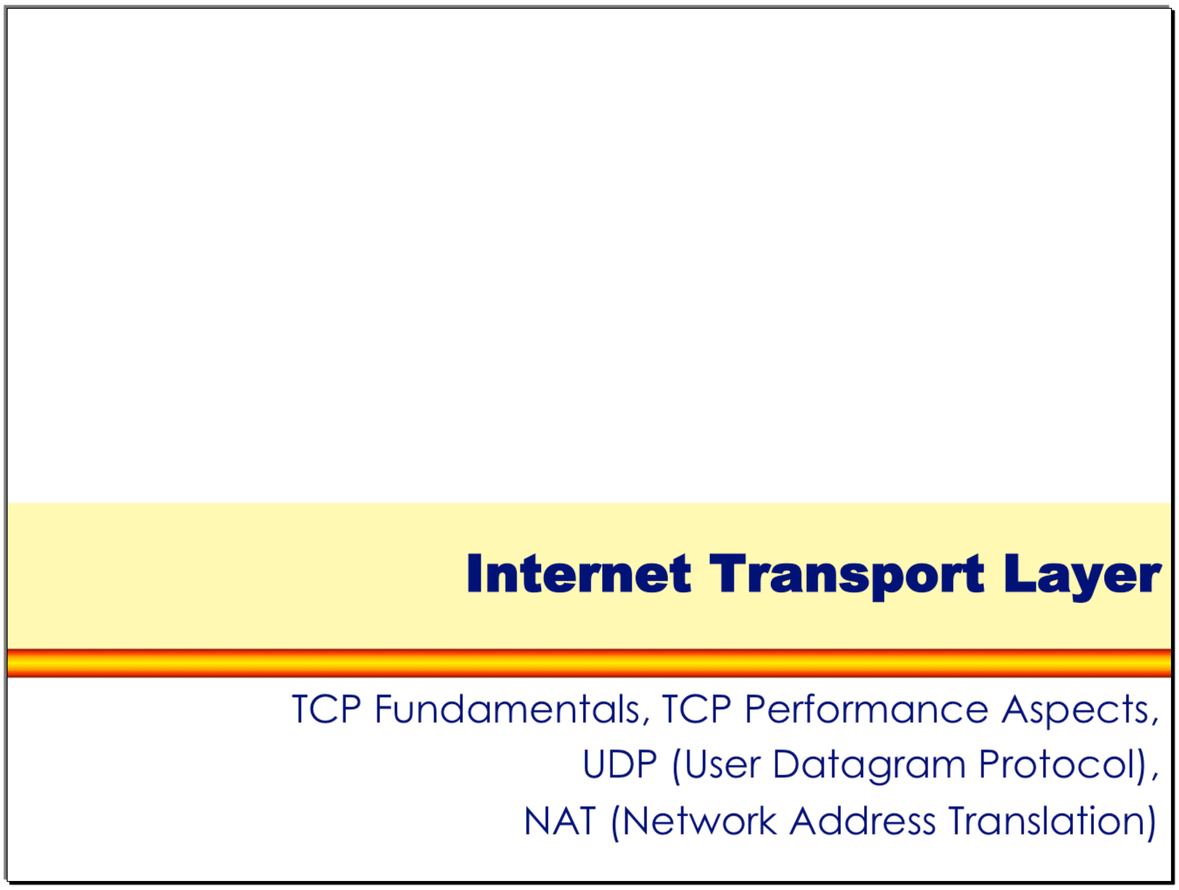
L11 - TCP, UDP and NAT (v6.0)

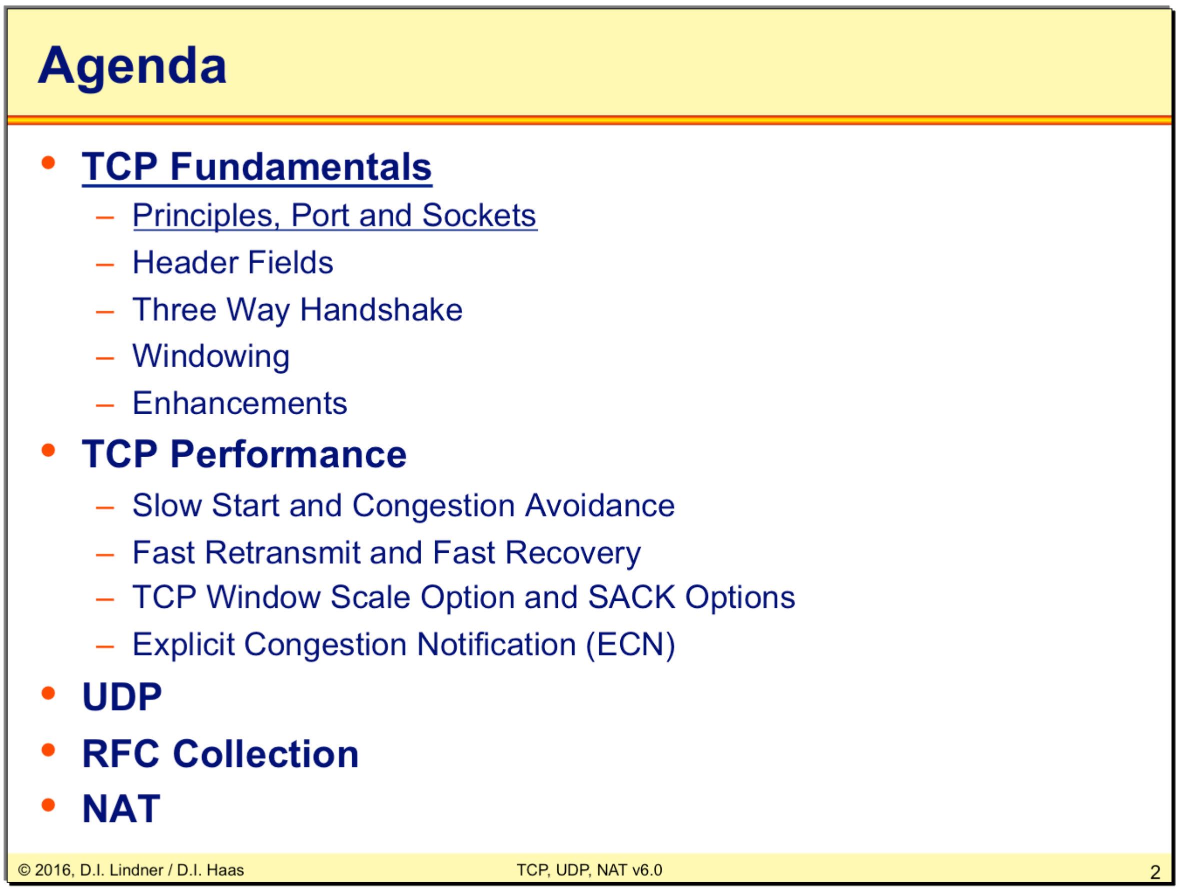


Internet Transport Layer

TCP Fundamentals, TCP Performance Aspects,
UDP (User Datagram Protocol),
NAT (Network Address Translation)

Changes to version v5.1:

Slide 35:	SEQ and ACK numbers corrected for TCP Disconnect
Slide 158:	New RFC for NAT stated

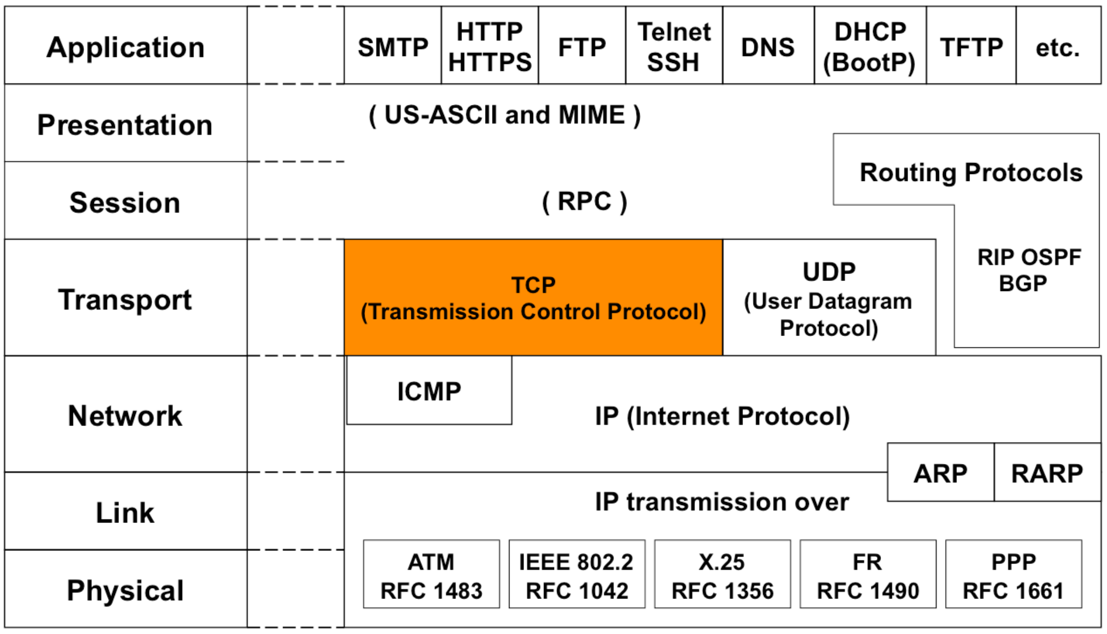


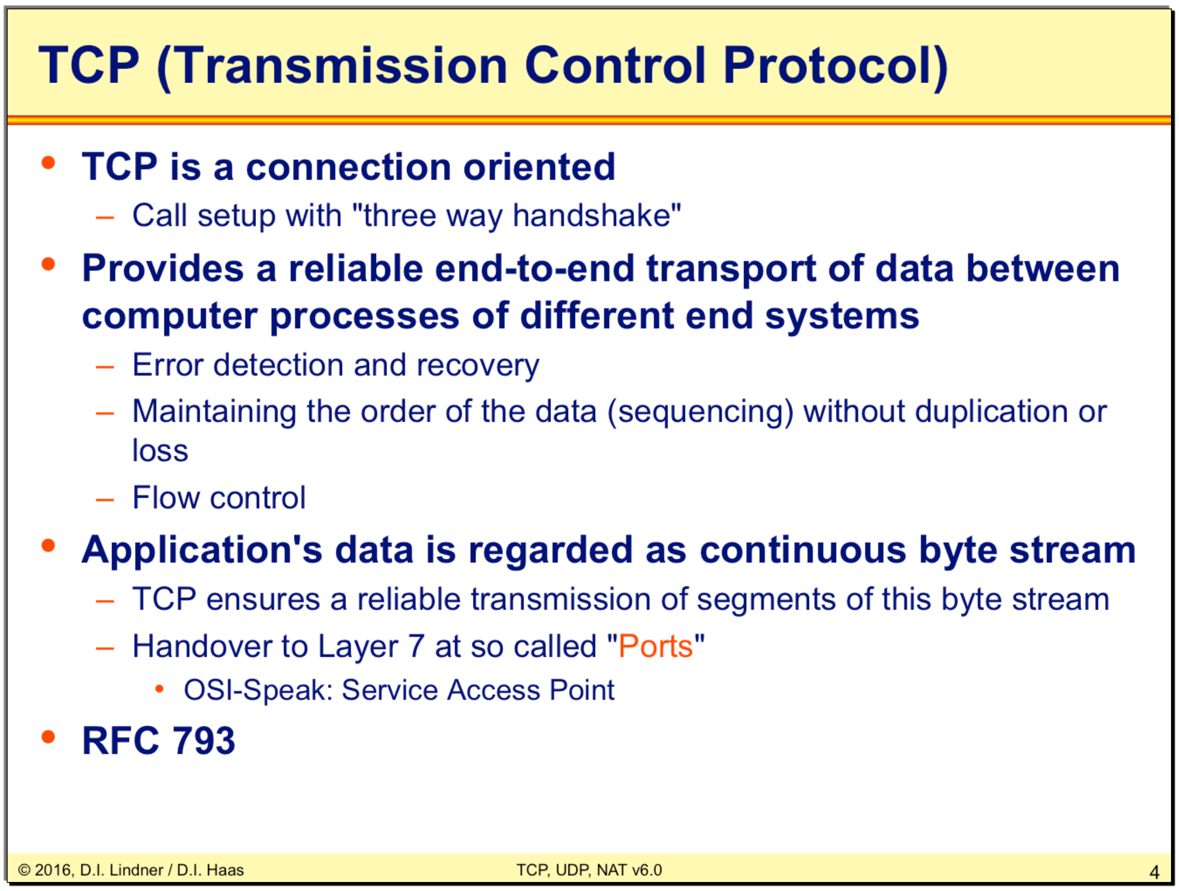
Agenda

TCP, UDP, NAT v6.0 2

L11 - TCP, UDP and NAT (v6.0)

TCP/IP Protocol Suite





TCP (Transmission Control Protocol)

- **TCP is a connection oriented**
  - Call setup with "three way handshake"
- **Provides a reliable end-to-end transport of data between computer processes of different end systems**
  - Error detection and recovery
  - Maintaining the order of the data (sequencing) without duplication or loss
  - Flow control
- **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 so called "Ports"
    - OSI-Speak: Service Access Point
- **RFC 793**

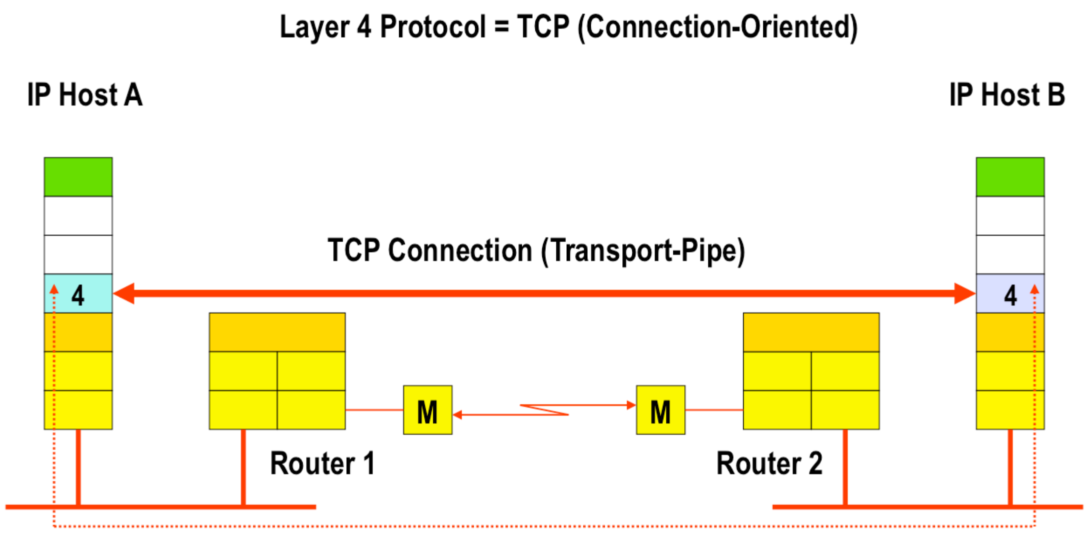
© 2016, D.I. Lindner / D.I. Haas   TCP, UDP, NAT v6.0   4

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

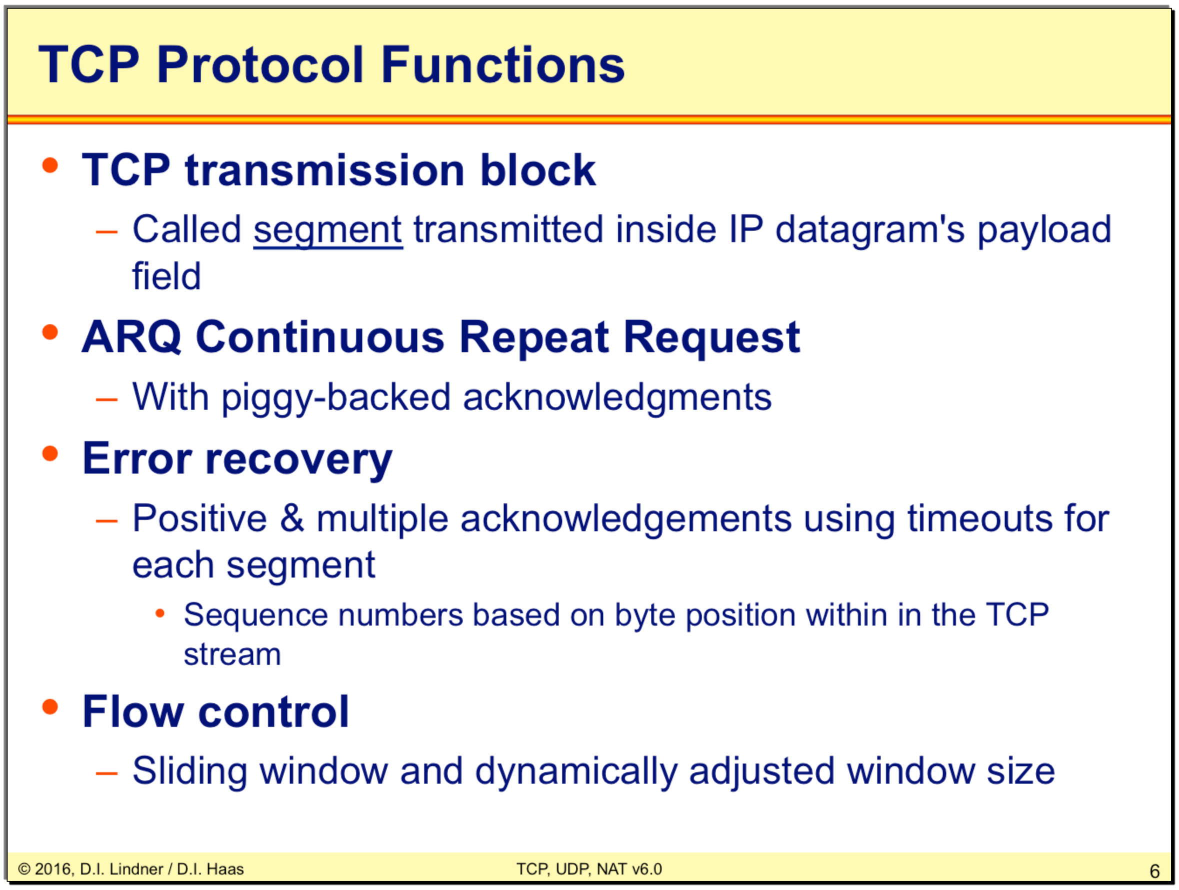
One important thing with TCP is the **Port-Number**, which will be discussed later in this chapter.

TCP and OSI Transport Layer 4


Layer 4 Protocol = TCP (Connection-Oriented)
IP Host A
IP Host B
TCP Connection (Transport-Pipe)
4
4
M
M
Router 1
Router 2

TCP, UDP, NAT v6.0 5

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. End systems see the network communication as reliable transport pipe (which could be compared with a virtual circuit already known from the network principles chapter) connecting them to each other.



TCP Protocol Functions

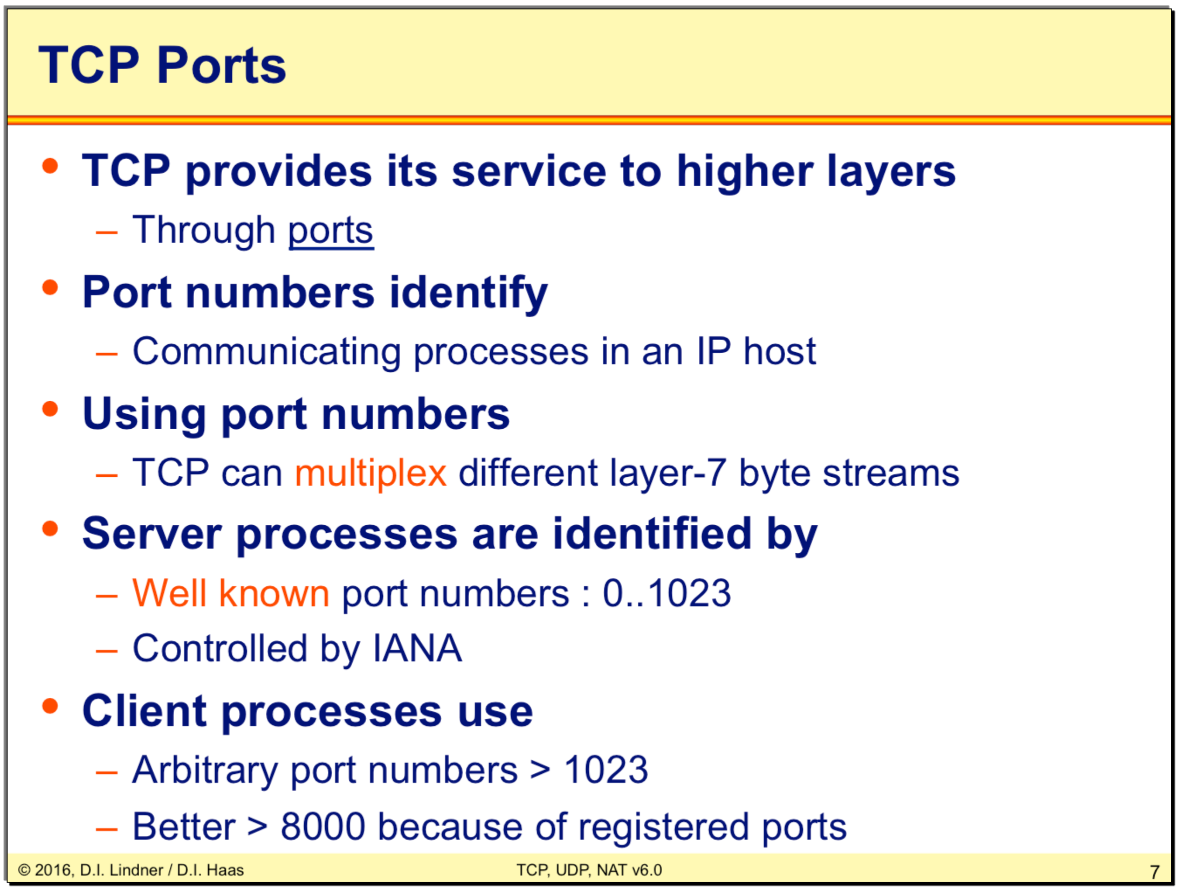
- **TCP transmission block**
  - Called segment transmitted inside IP datagram's payload field
- **ARQ Continuous Repeat Request**
  - With piggy-backed acknowledgments
- **Error recovery**
  - Positive & multiple acknowledgements using timeouts for each segment
    - Sequence numbers based on byte position within in the TCP stream
- **Flow control**
  - Sliding window and dynamically adjusted window size

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 6

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

In general, segments are encapsulated in single IP datagrams.

Maximum segment size depends on max. packet or frame size used by IP next hop link (fragmentation is possible)



TCP Ports

- TCP provides its service to higher layers
  - Through ports
- Port numbers identify
  - Communicating processes in an IP host
- Using port numbers
  - TCP 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

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 7

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.

Well known ports are reserved for common applications and services (like Telnet, WWW, FTP etc.) and are in the range from 0 to 1023. They 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 used by proprietary server applications They are not controlled by the IANA but only listed -> see RFC1700 for details.

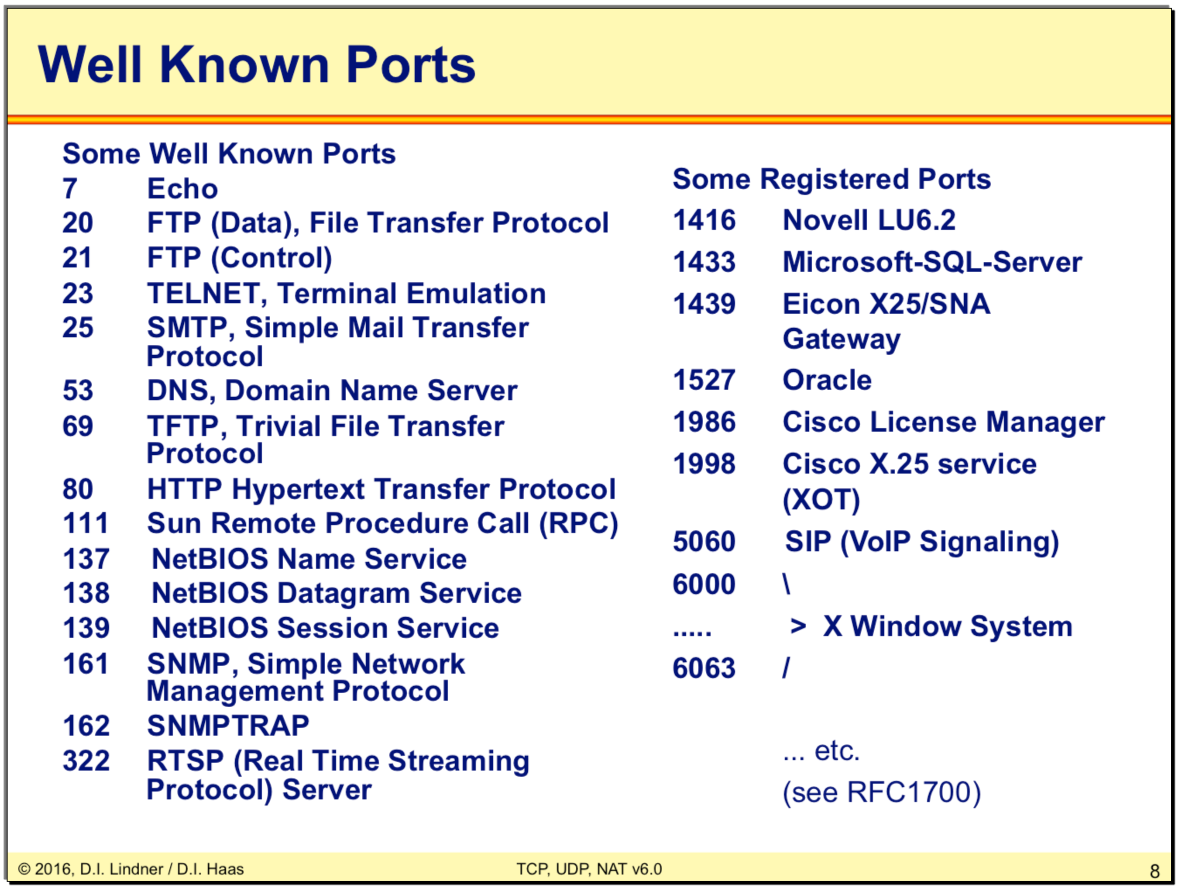
Remember: A TCP connection is always initiated from client to server.

Server applications listen on their well-known ports for incoming TCP connections. A well-known port of a server process is used as destination port of an outgoing TCP segment from the client.

Client applications chose a free port number (which is not already used by another outgoing TCP connection) as the source port of an outgoing TCP segment sent to the server.

Some services like FTP (File Transfer Protocol) or RPC (Remote Procedure Call) use dynamically assigned port numbers. Sun RPC (Remote Procedure Call) uses a portmapper

L11 - TCP, UDP and NAT (v6.0)



Well Known Ports

Some Well Known Ports

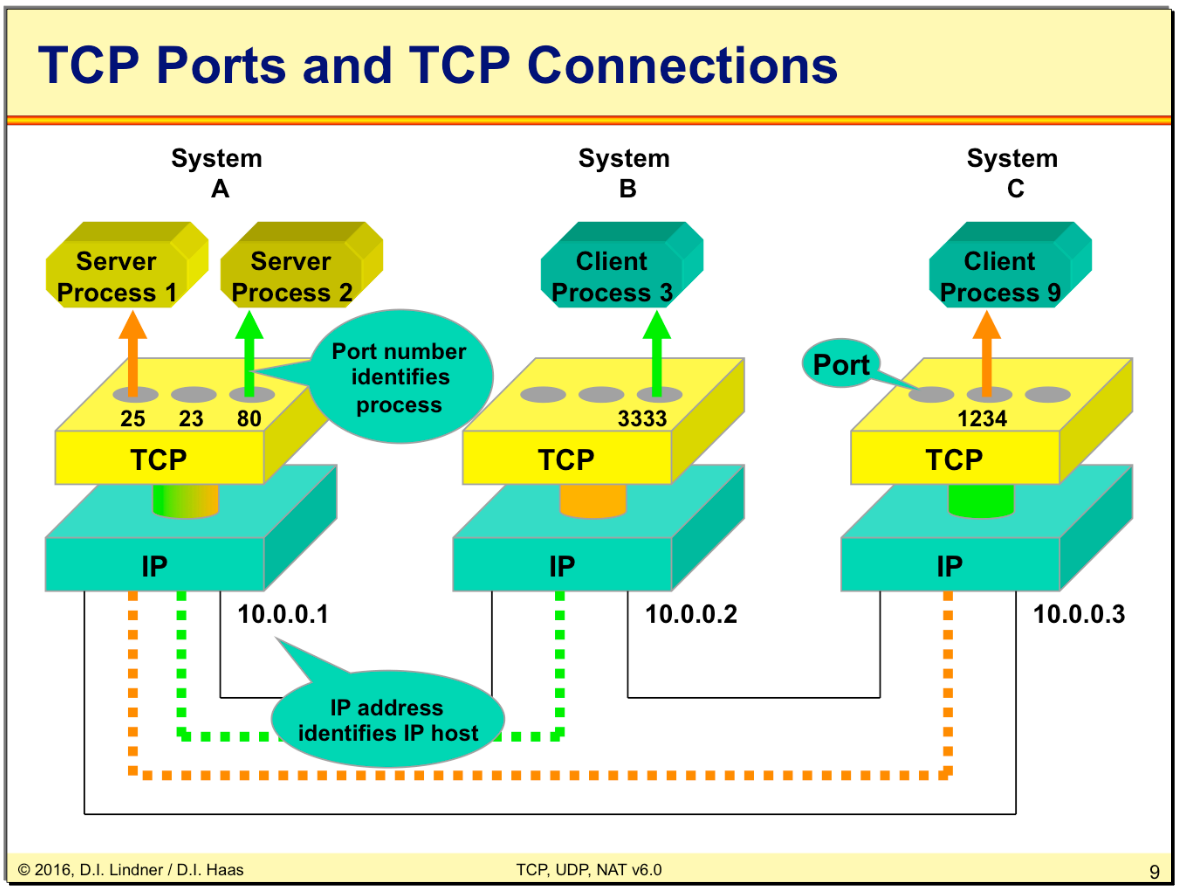
7	Echo
20	FTP (Data), File Transfer Protocol
21	FTP (Control)
23	TELNET, Terminal Emulation
25	SMTP, Simple Mail Transfer Protocol
53	DNS, Domain Name Server
69	TFTP, Trivial File Transfer Protocol
80	HTTP Hypertext Transfer Protocol
111	Sun Remote Procedure Call (RPC)
137	NetBIOS Name Service
138	NetBIOS 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 Manager
1998	Cisco X.25 service (XOT)
5060	SIP (VoIP Signaling)
6000	\
.....	> X Window System
6063	/

... etc.
(see RFC1700)

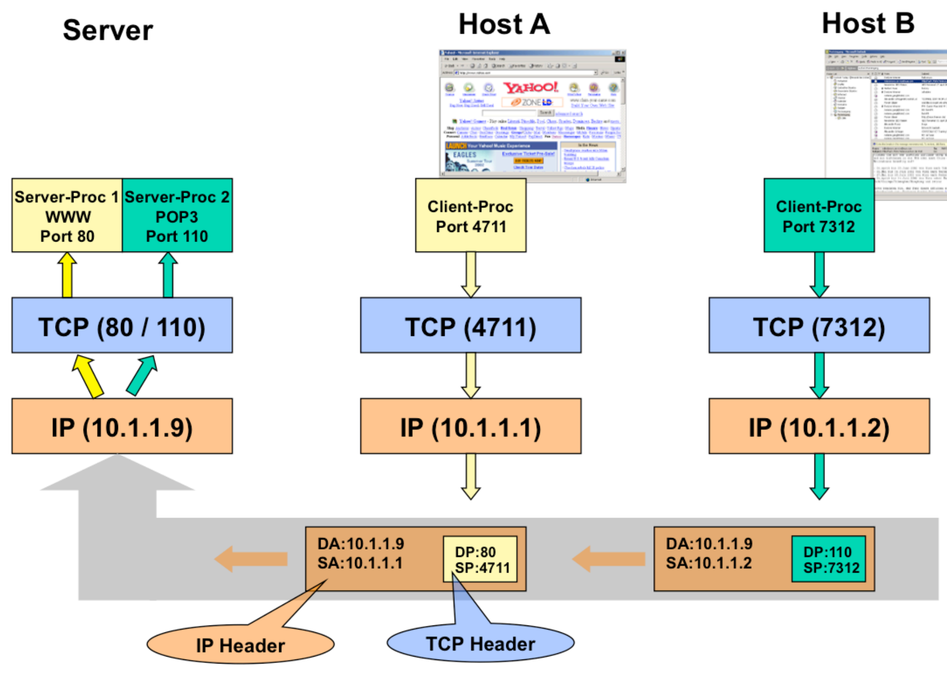
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 8



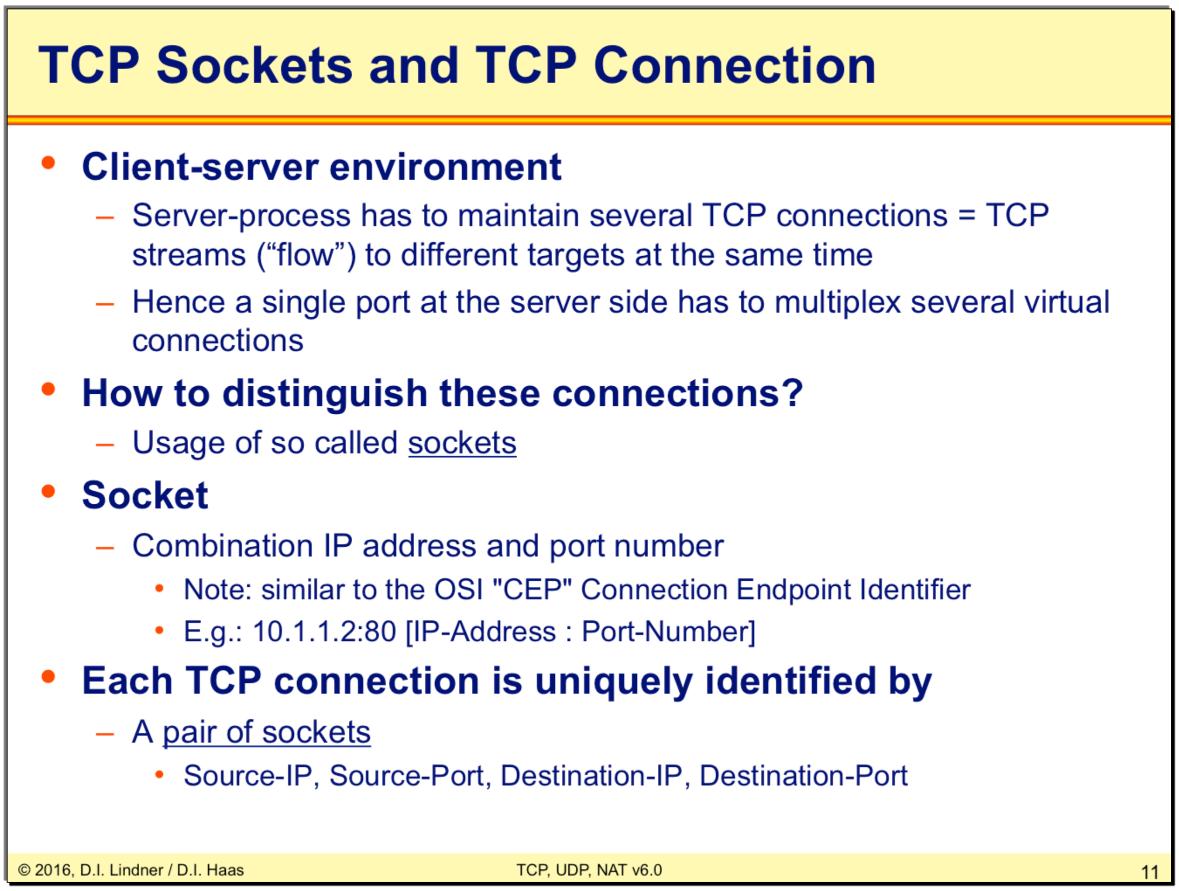
TCP Ports and TCP Connections

The TCP software functions like a multiplexer and demultiplexer for several TCP connections:
Port 25 on system A: process 1, system A <---------> port 1234, process 9, system C
Port 80 on system A: process 2, system A <---------> port 3333, process 3, system B

Example 1: TCP Port



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, well known port 110) and the client application uses the source port (SP) 7312. The TCP segment is send to the server with a destination-port (DP) of 110. Now the server knows host B and B makes a mail-check over POP3.

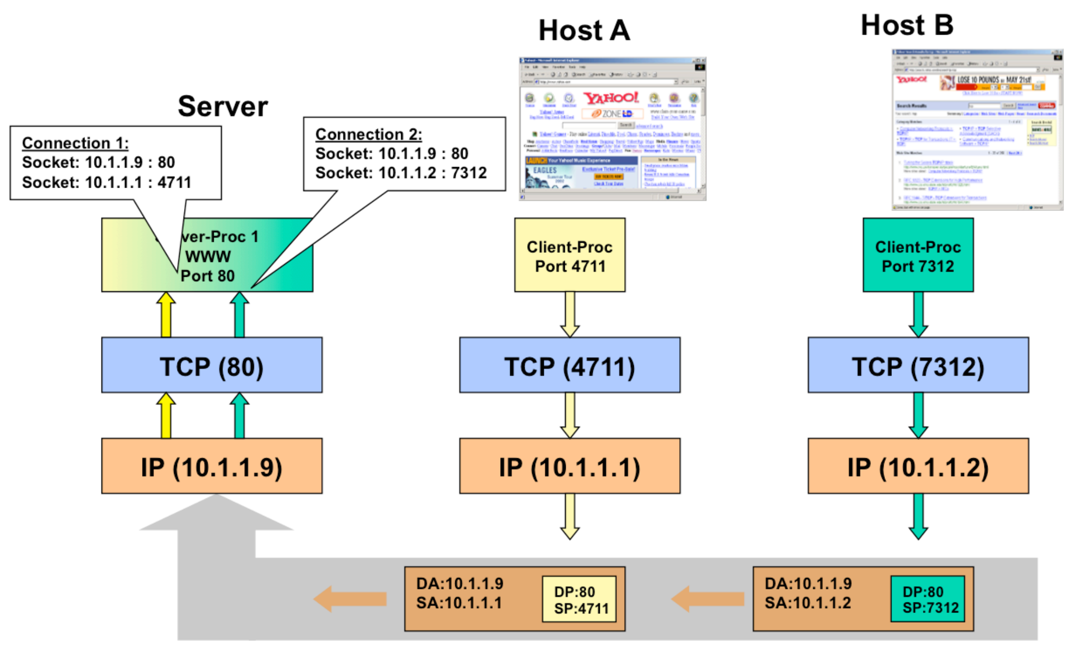


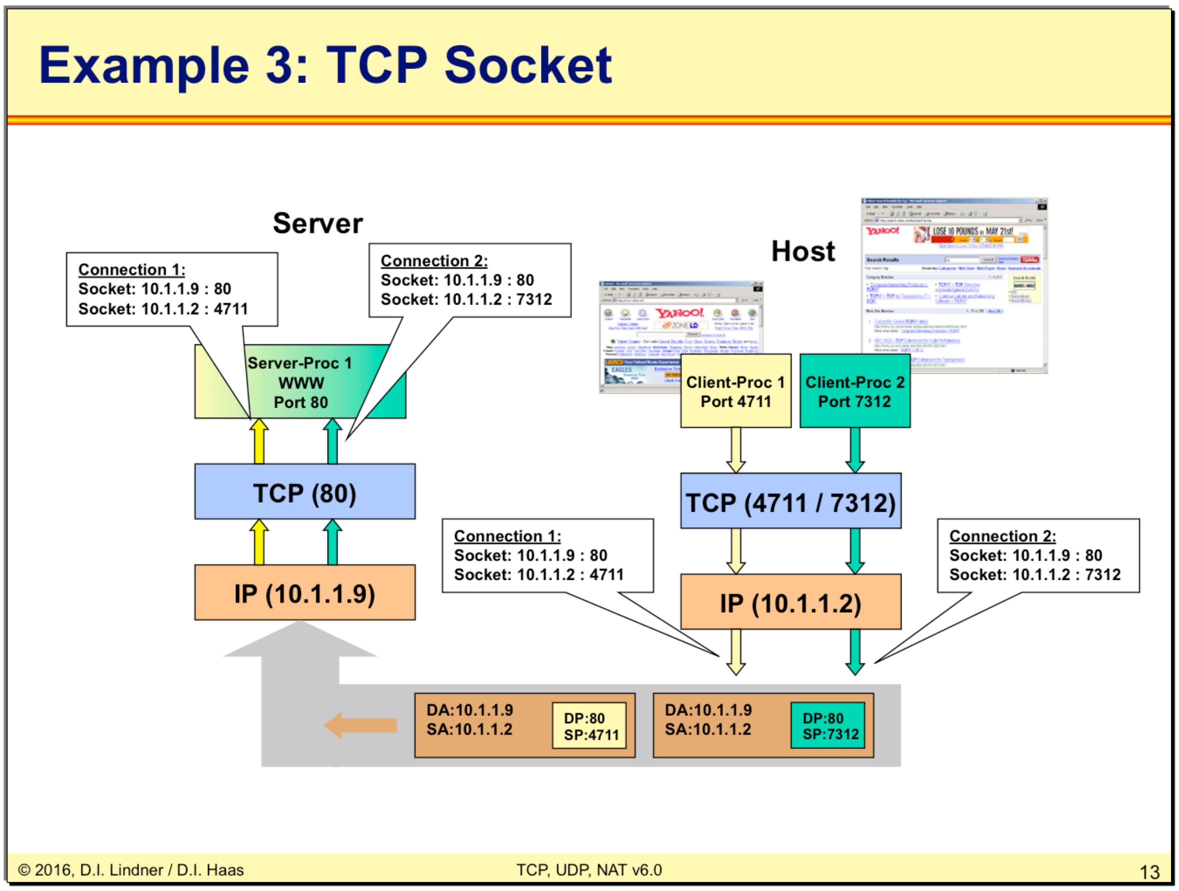
TCP Sockets and TCP Connection

- **Client-server environment**
  - Server-process has to maintain several TCP connections = TCP streams (“flow”) to different targets at the same time
  - Hence a single port at the server side has to multiplex several virtual connections
- **How to distinguish these connections?**
  - Usage of so called sockets
- **Socket**
  - Combination IP address and port number
    - Note: similar to the OSI "CEP" Connection Endpoint Identifier
    - E.g.: 10.1.1.2:80 [IP-Address : Port-Number]
- **Each TCP connection is uniquely identified by**
  - A pair of sockets
    - Source-IP, Source-Port, Destination-IP, Destination-Port

Server process multiplexes incoming streams with same destination port numbers according source IP address.

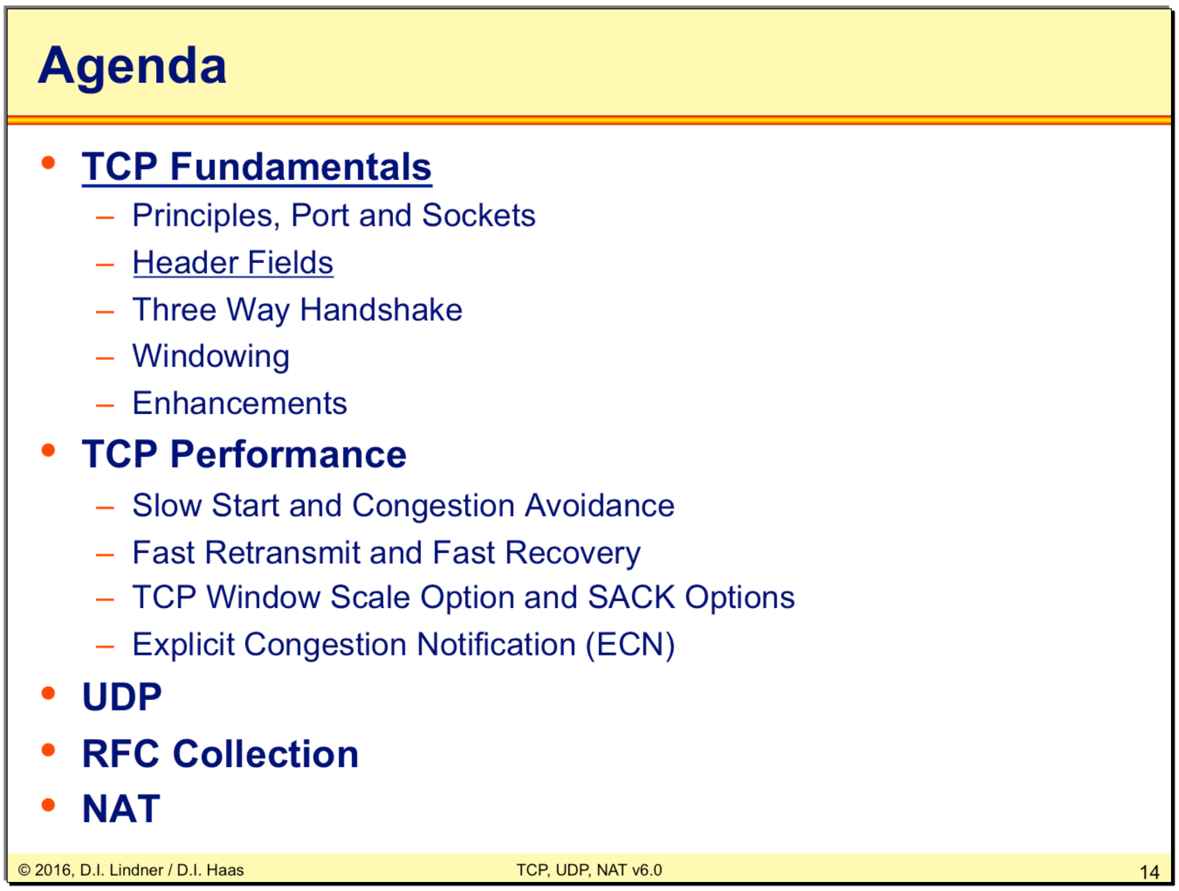
Example 2: TCP Socket





Example 3: TCP Socket

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.

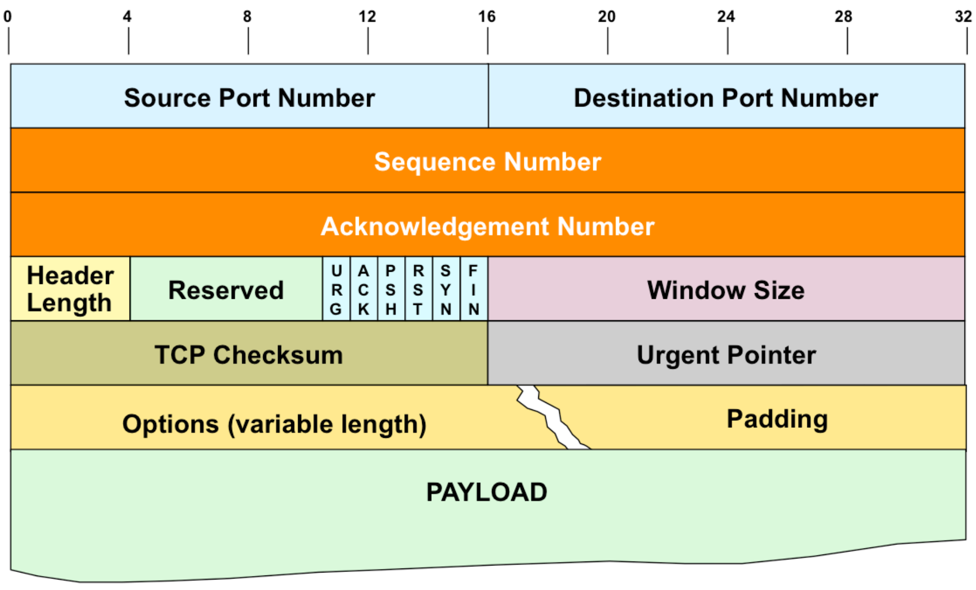


Agenda

- TCP Fundamentals
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- TCP Performance
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- UDP
- RFC Collection
- NAT

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 14

TCP Header



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.

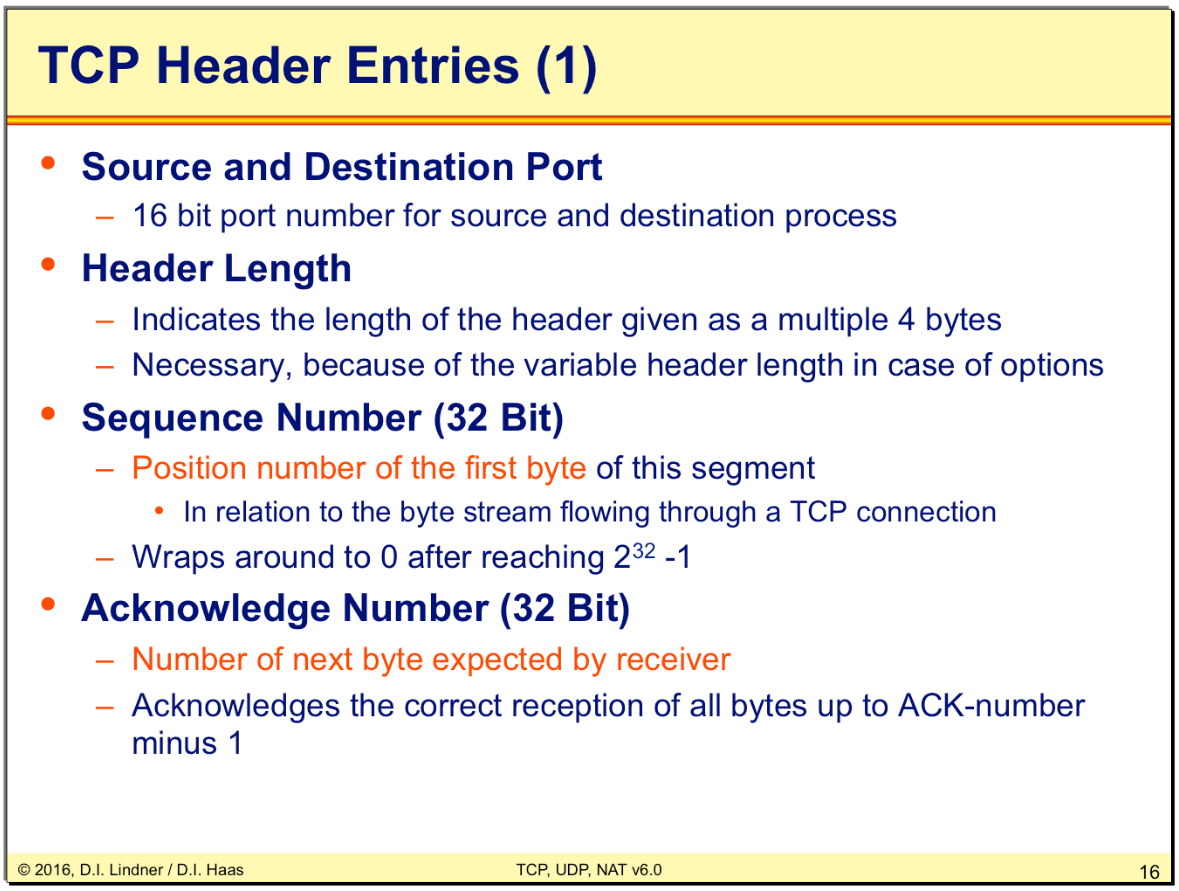
It is important to know these header fields, at least the most important parts:

The Port numbers – most important, to address applications

The Sequence numbers (SQNR and Ack) – used for error recovery

The Window field – used for flow control

The flags SYN, ACK, RST, and FIN – for session control



TCP Header Entries (1)

- **Source and Destination Port**
  - 16 bit port number for source and destination process
- **Header Length**
  - Indicates the length of the header given as a multiple 4 bytes
  - Necessary, because of the variable header length in case of options
- **Sequence Number (32 Bit)**
  - Position number of the first byte of this segment
    - In relation to the byte stream flowing through a TCP connection
  - Wraps around to 0 after reaching 2^{32} -1
- **Acknowledge Number (32 Bit)**
  - Number of next byte expected by receiver
  - Acknowledges the correct reception of all bytes up to ACK-number minus 1

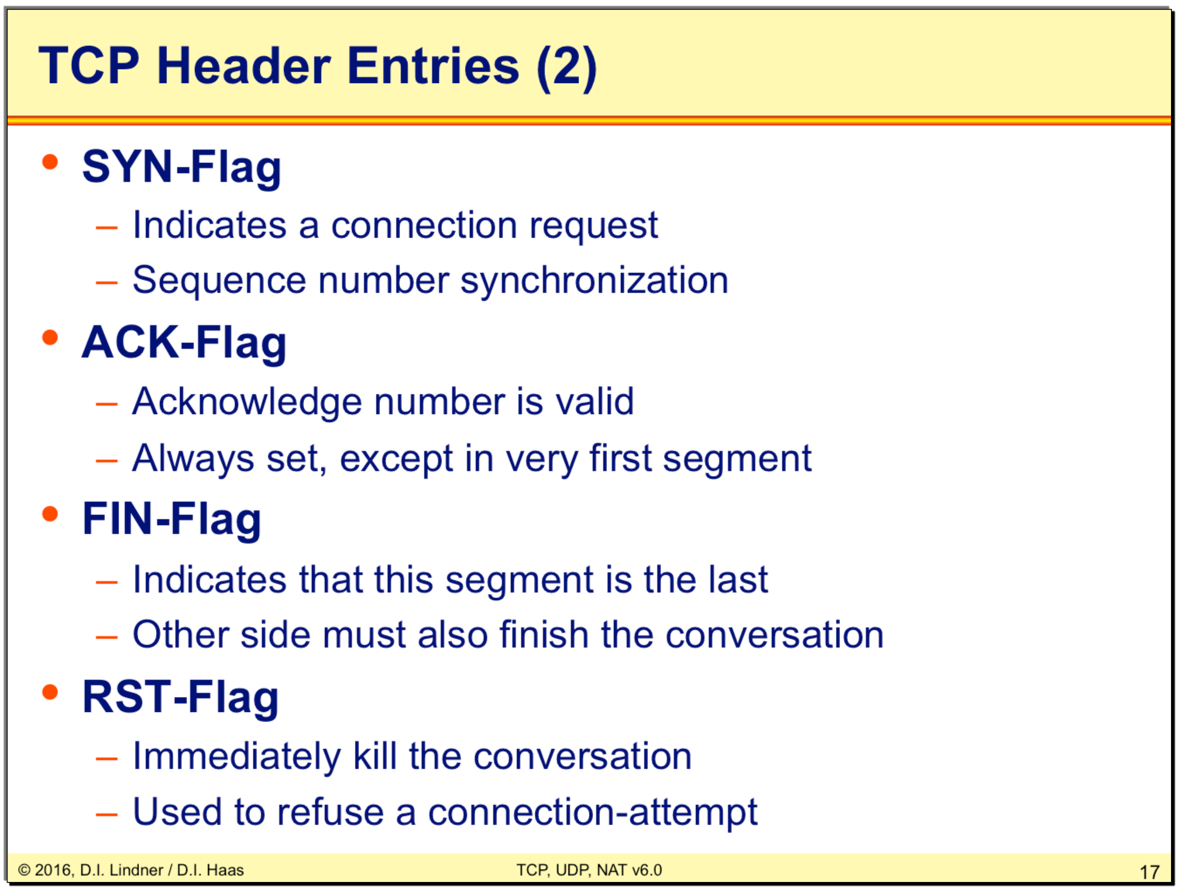
TCP, UDP, NAT v6.0 16

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.

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 byte 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 Entries (2)

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

SYN-Flag: 1 Bit. Control Bit.

Used for call setup. If the SYN bit is set to 1, the application knows that a host want to established a connection with him. Also used to synchronization the sequence numbers because the sequence number holds the initial value for a new session. Most firewalls discard TCP segments with SYN=1 if a host want to established a connection to a server application which is not allowed for security reasons.

ACK-Flag: 1 bit. Control Bit.

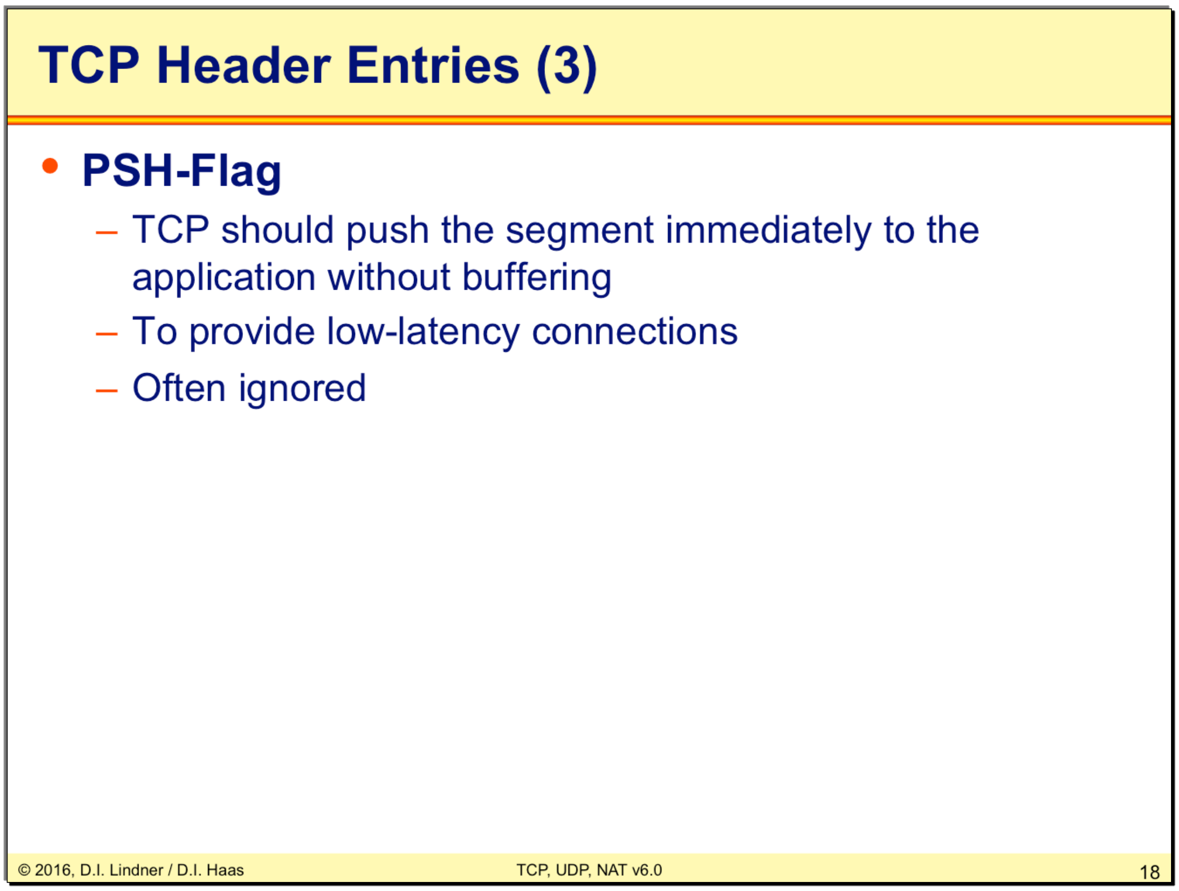
Acknowledgment Bit. If set, the acknowledge number is valid and indicates the sequence number of the next octet expected by the receiver

FIN-Flag: 1 bit. Control Bit.

The FIN-Flag is used in the disconnect phase. It indicates that this segment is the last one. If set, the Sequence Number holds the number of the last transmitted byte of a session. Using this number a process can indicate all data that have been received by him. After the other side has also sent a segment with FIN=1, the connection is closed.

RST-Flag: 1 bit. Control Bit.

If set, the session has to be cleared immediately (reset). Can be used to refuse a connection-attempt or to "kill" a current connection.

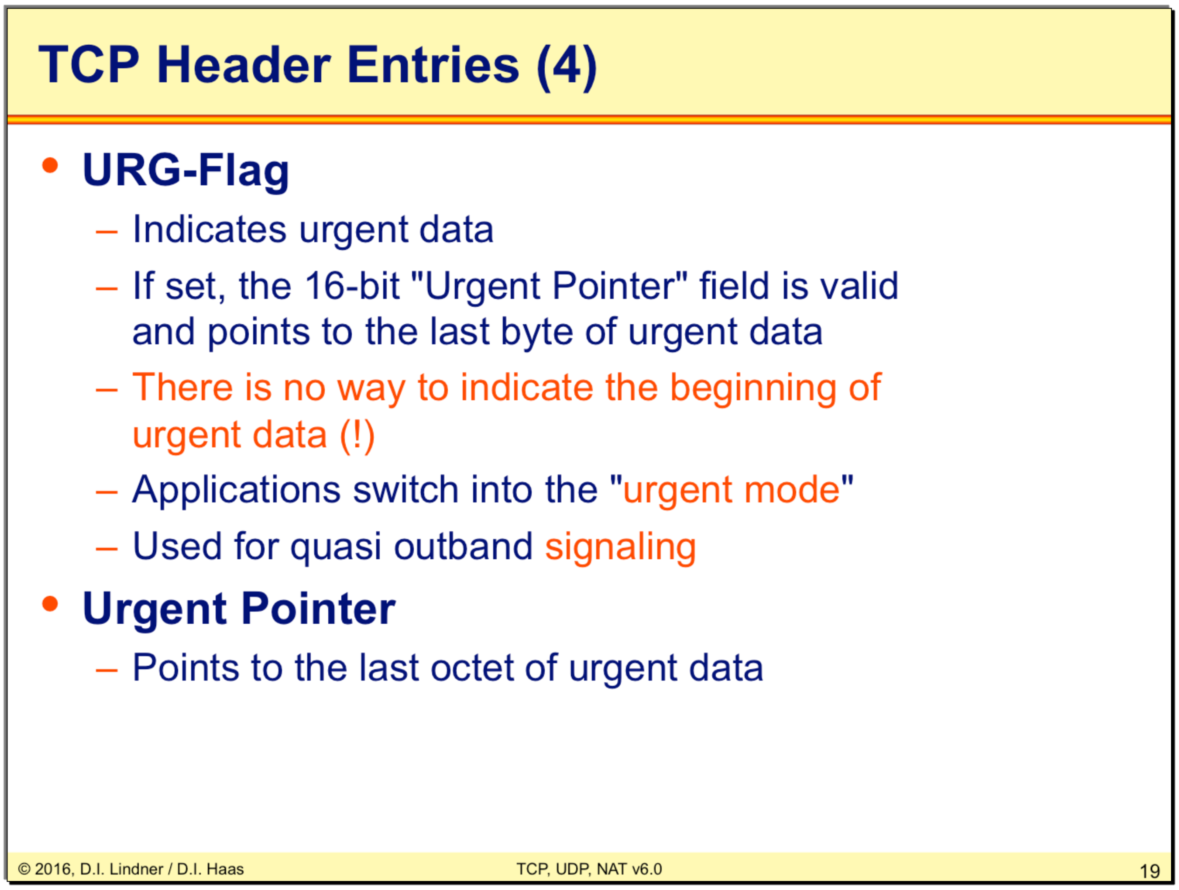


TCP Header Entries (3)

- **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 Entries (4)

- **URG-Flag**
  - Indicates urgent data
  - If set, the 16-bit "Urgent Pointer" field is valid and points to the last byte 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
- **Urgent Pointer**
  - Points to the last octet of urgent data

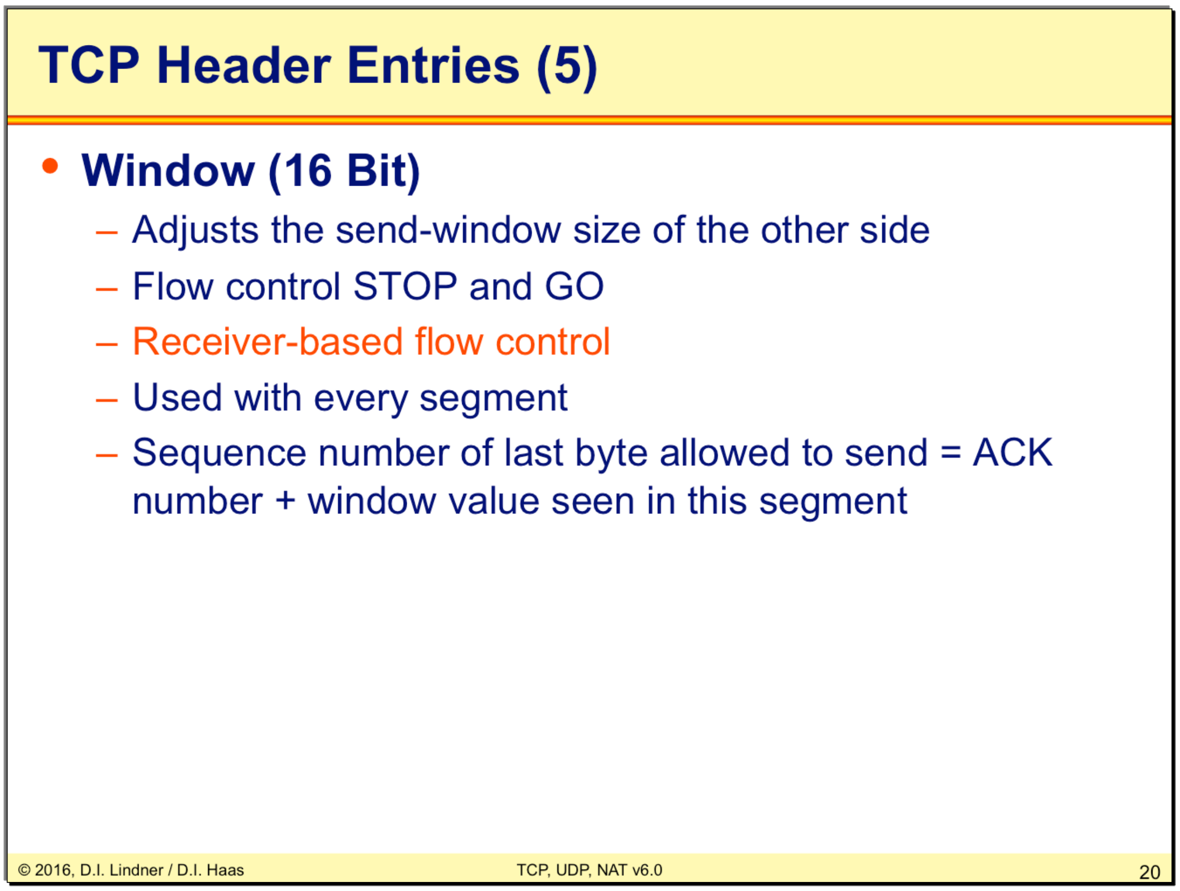
© 2016, D.I. Lindner / D.I. Haas   TCP, UDP, NAT v6.0   19

URG-Flag: 1 Bit. Control Bit.

Sequence number of last urgent byte = actual segment sequence number + urgent pointer

RFC 793 and several implementations assume the urgent pointer to point to the first byte *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 byte of urgent data is received. Examples for usage: Interrupt key in Telnet, Rlogin, or FTP.

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



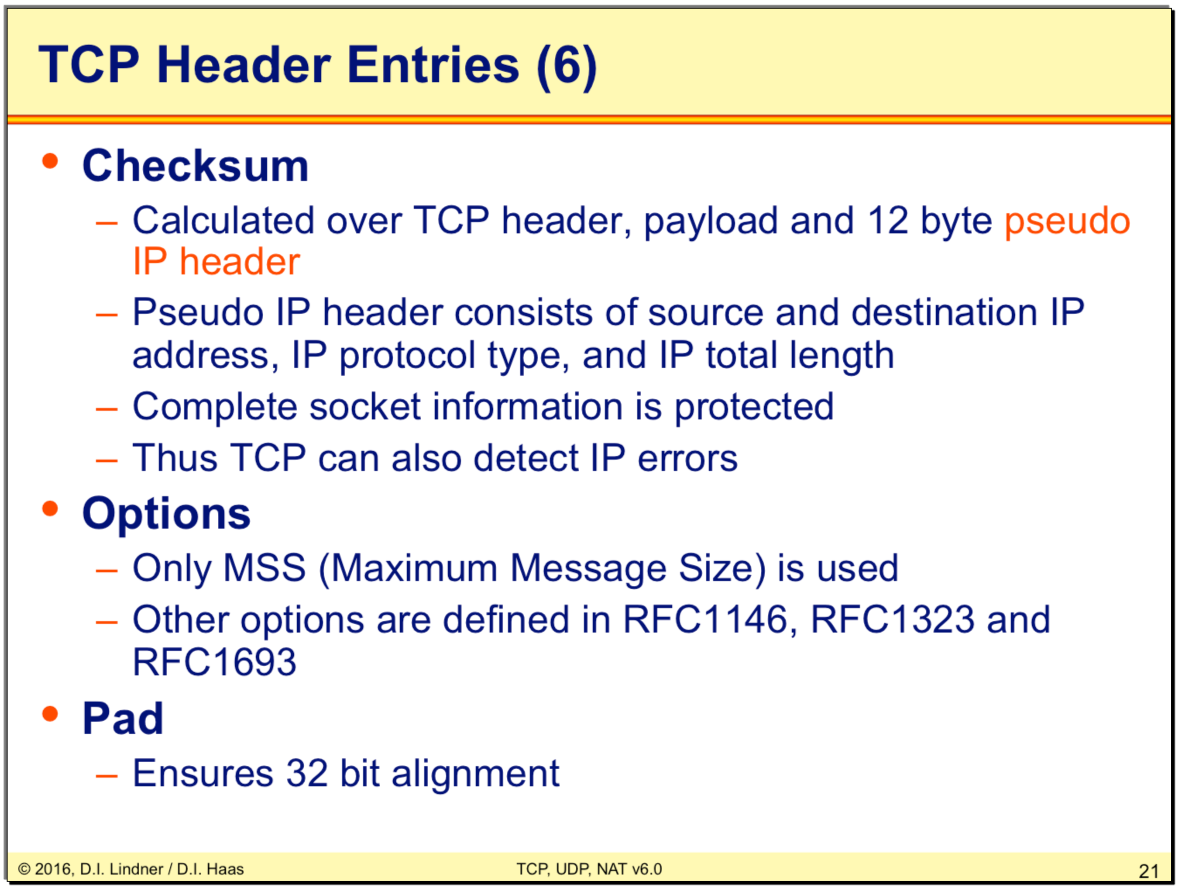
TCP Header Entries (5)

- **Window (16 Bit)**
  - Adjusts the send-window size of the other side
  - Flow control STOP and GO
  - Receiver-based flow control
  - Used with every segment
  - Sequence number of last byte allowed to send = ACK number + window value seen in this segment

Window Size: 16 bit. The number of data bytes beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept. See windowing / flow control slides.

Set by the receiver side of a TCP stream with every transmitted segment to signal the allowed current window size to the sender; this "dynamic windowing" enables receiver-based flow control. The value defines how many additional bytes will be accepted, starting from the current acknowledgment number plus window value seen in this segment.

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.



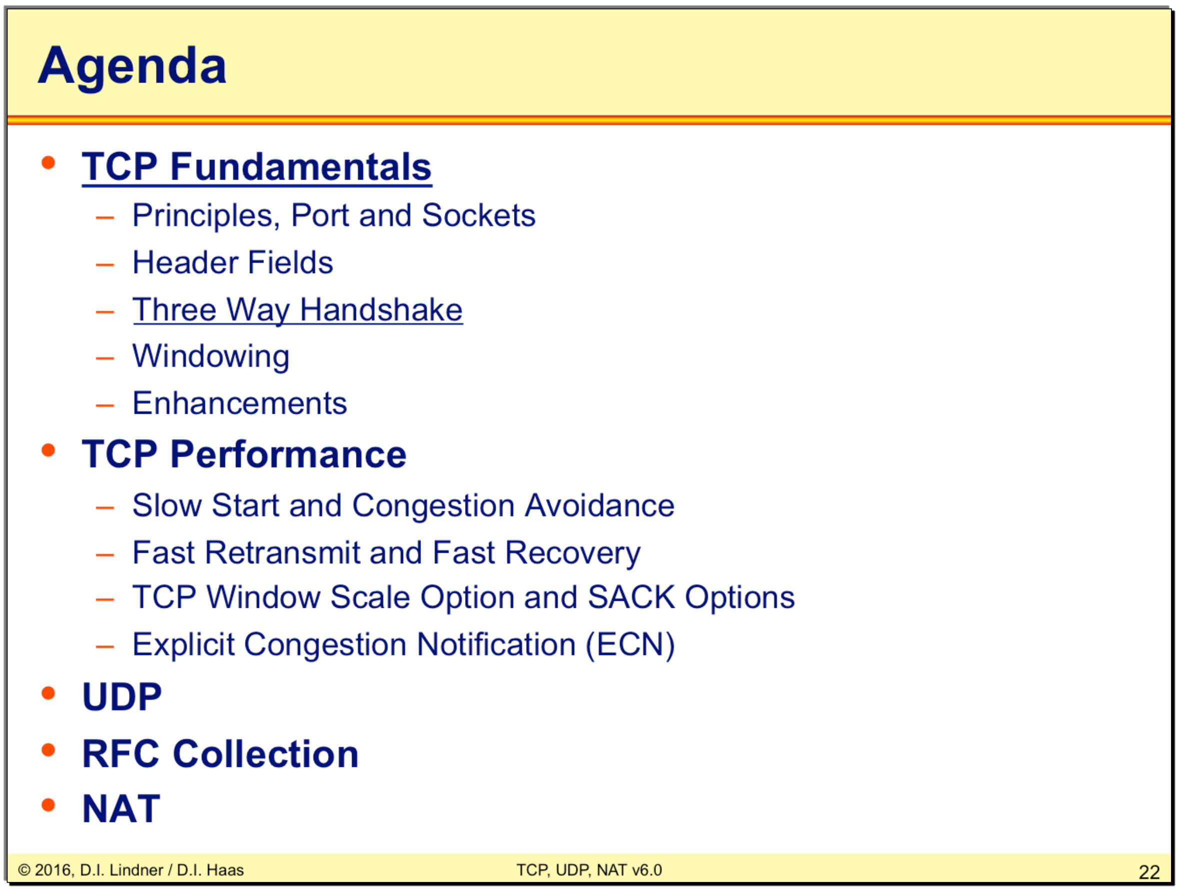
TCP Header Entries (6)

- **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
- **Options**
  - Only MSS (Maximum Message Size) is used
  - Other options are defined in RFC1146, RFC1323 and RFC1693
- **Pad**
  - Ensures 32 bit alignment

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. 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).

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.

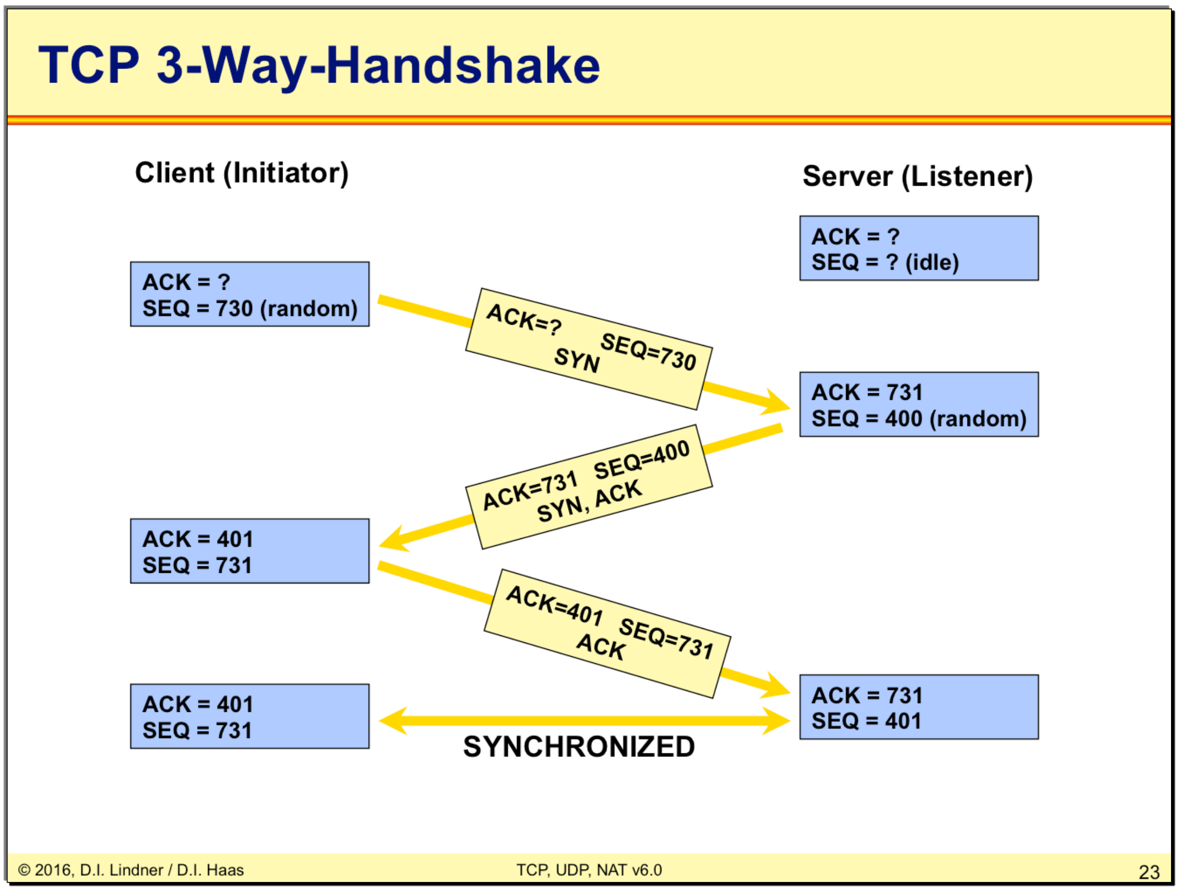
Padding: Variable length. The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.



Agenda

- TCP Fundamentals
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- TCP Performance
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- UDP
- RFC Collection
- NAT

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 22



TCP 3-Way-Handshake

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 23

A TCP connection ist established by a 3-way handshake procedure.

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.)

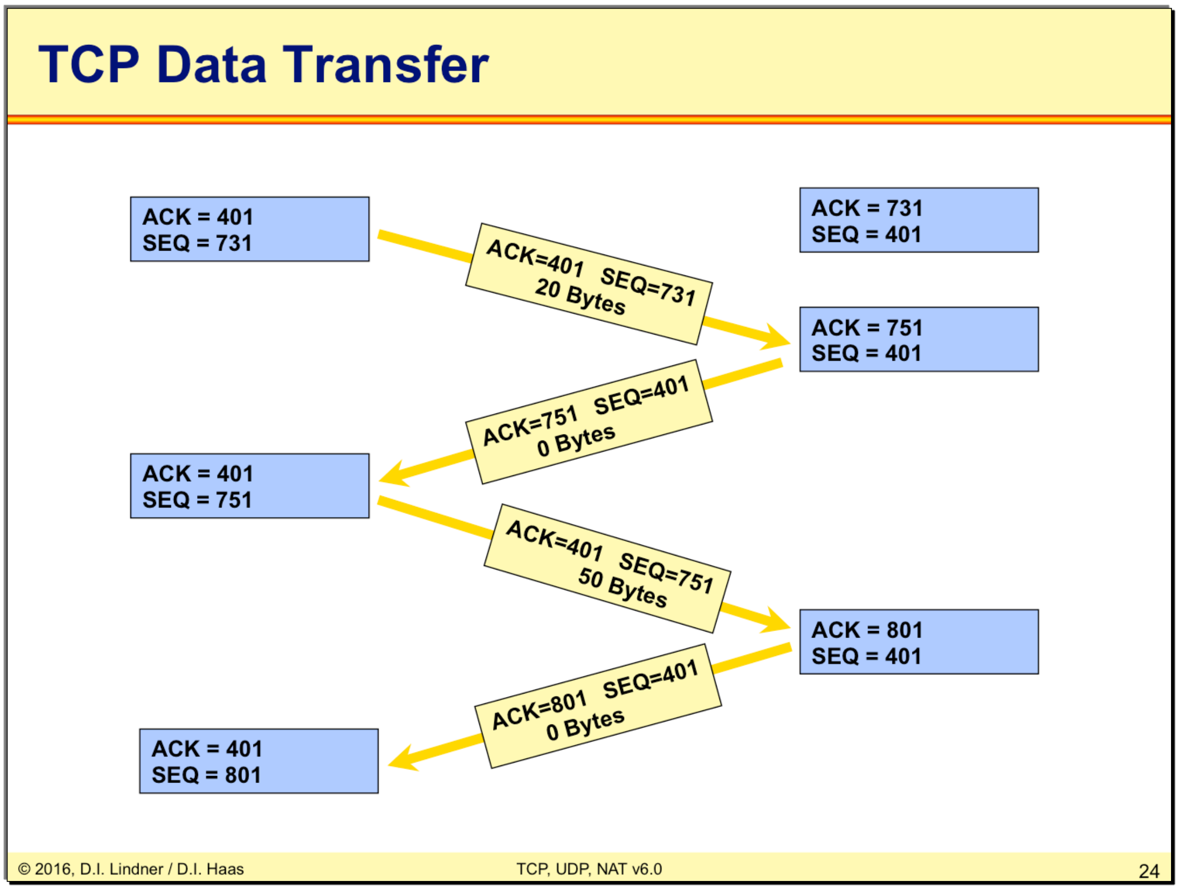
Why do we need such a procedure?

Remember TCP uses the unreliable service of IP, hence TCP segments of old sessions (e.g. retransmitted or delayed segments, duplicates) could disturb the establishment of a new TCP connection but also the new TCP connection itself. Thus sequence numbers must be unique for different sessions of the same socket.

Random starting sequence numbers, an explicit negotiation of starting sequence numbers and a huge sequence number range make a TCP connect immune against spurious datagrams. 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.

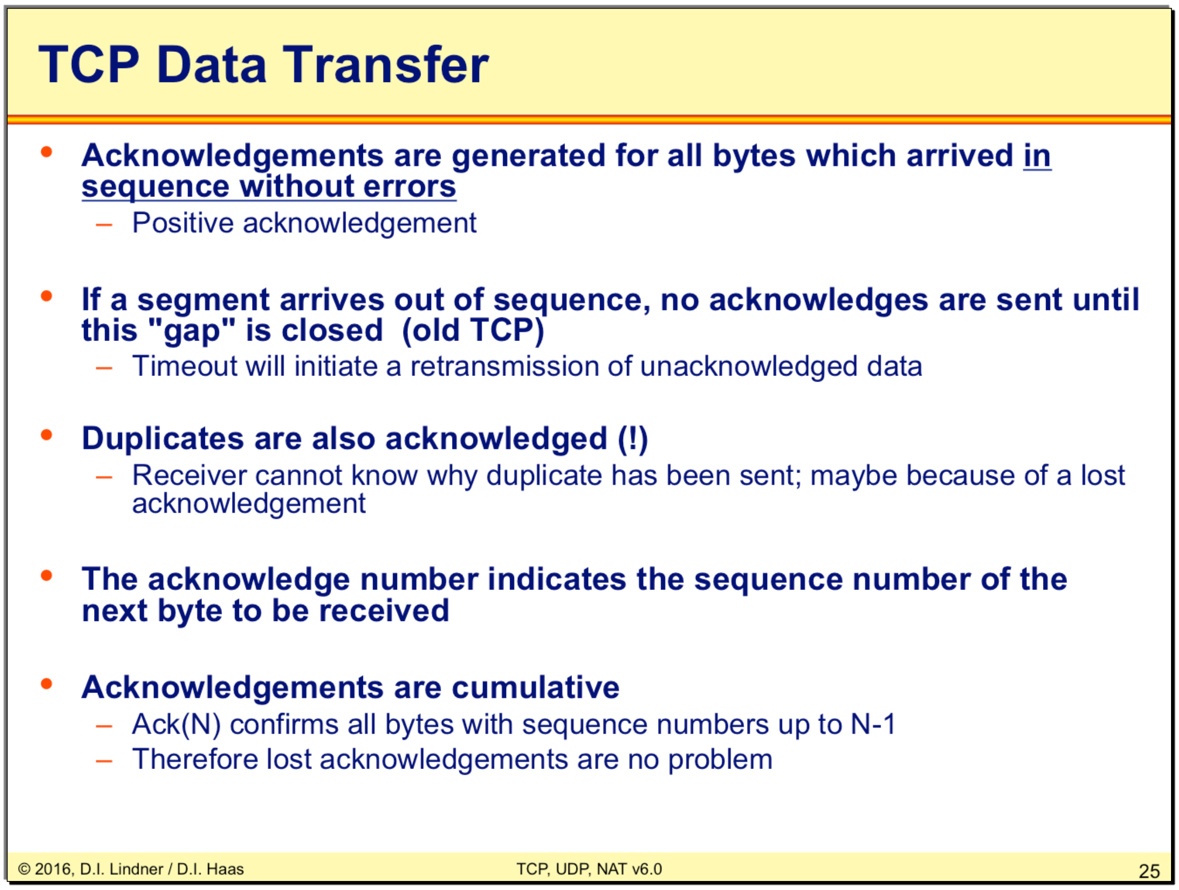
Also disturbing segments (e.g. delayed TCP segments from old sessions) and old "half-open" connections are deleted with the RST flag.



TCP Data Transfer

TCP, UDP, NAT v6.0 24

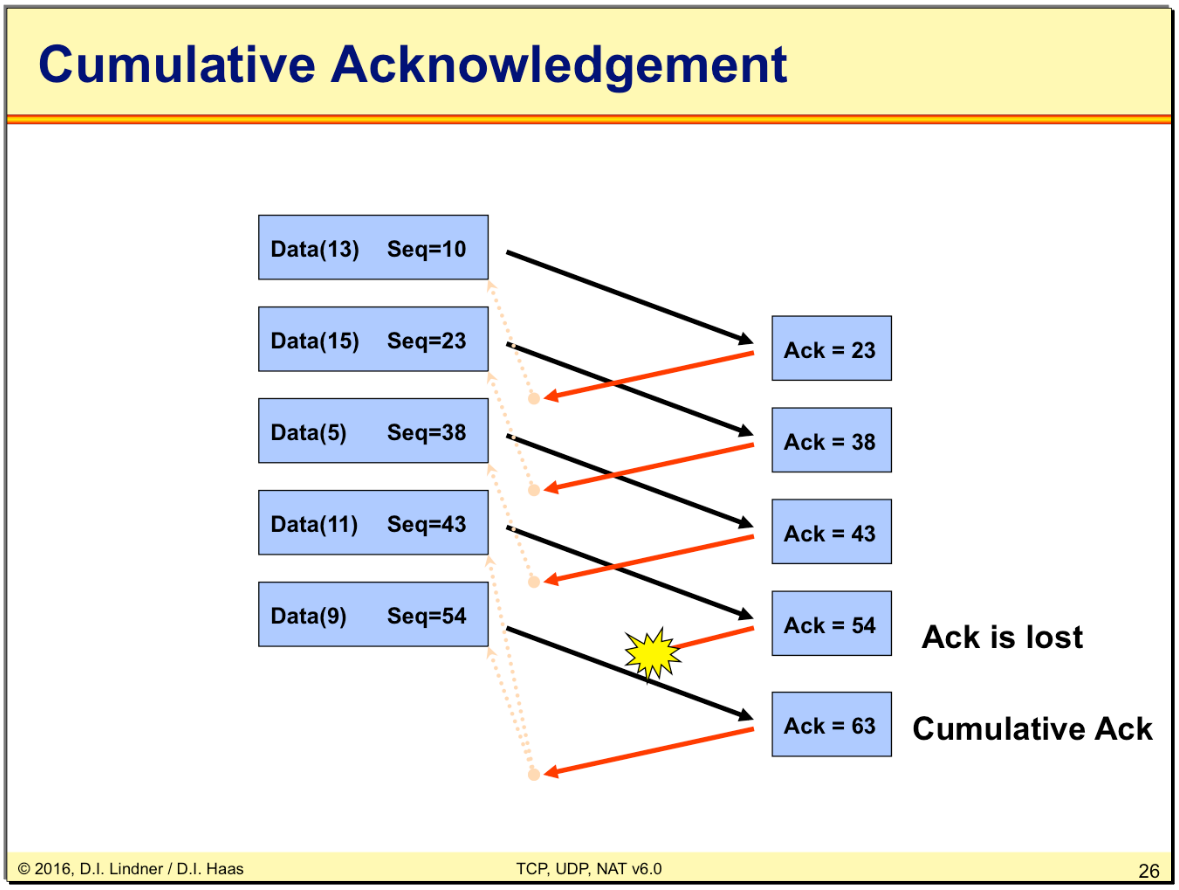
After the 3-way-handshake is finished the real data transfer is stared. A 20 Byte segment is sending to the server (ACK 401, SEQ 731). After the server receives the segment, he sets the ACK-flag to 751 (SEQ+20 Byte) and the SEQ to 401. Then he sends the segment back (ACK 751, SEQ 401) to the host. After 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 bytes which arrived <u>in sequence without errors</u>**
  - Positive acknowledgement
- **If a segment arrives out of sequence, no acknowledges are sent until this "gap" is closed (old TCP)**
  - Timeout will initiate a retransmission of unacknowledged data
- **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.

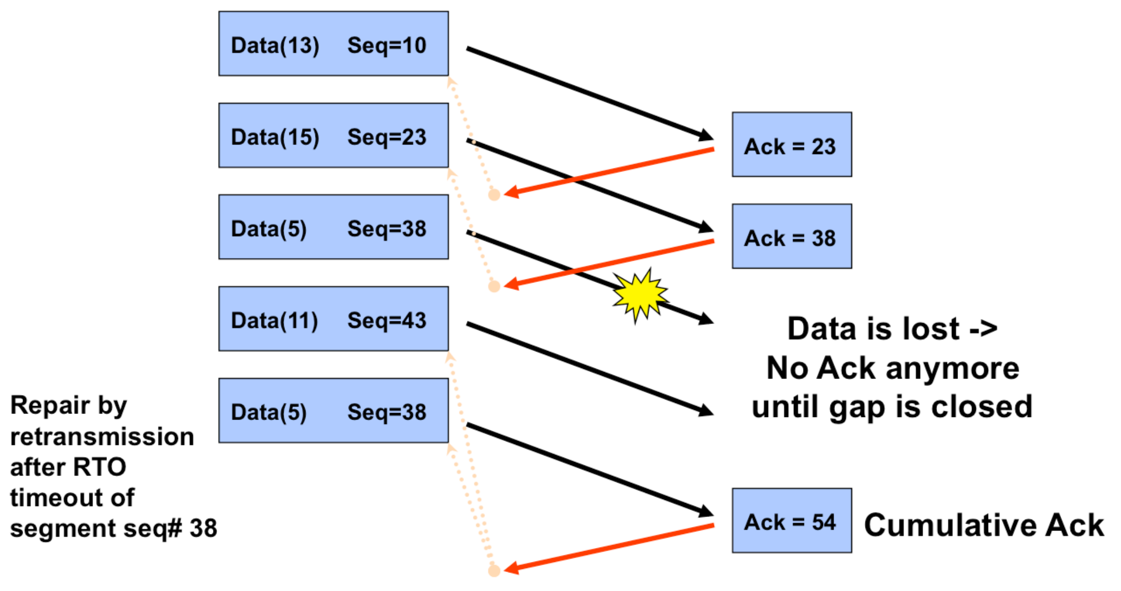


Cumulative Acknowledgement

TCP, UDP, NAT v6.0 26

Its not a problem for TCP when a acknowledgment get lost, because TCP acknowledges all in-sequence received data with every cumulative acknowledgement. The timers, which are started after sending an segment, are immediately stopped by receiving any an ACK.

TCP Duplicates, Lost Original (old TCP)


Data(13) Seq=10
Data(15) Seq=23
Data(5) Seq=38
Data(11) Seq=43
Data(5) Seq=38
Ack = 23
Ack = 38
Data is lost -> No Ack anymore until gap is closed
Ack = 54
Cumulative Ack
Repair by retransmission after RTO timeout of segment seq# 38

In case of out-of-sequence arrival of segments the receiver stops sending ACKs until the failure is repaired. The sender of the lost segment will wait for ACKs and will retransmit the segment as duplicate after the timer, which was started after sending the original segment, runs into timeout. (RTO). That was the original implementation of TCP (old TCP) -> Positive Acknowledgment based on timeouts only for error recovery.

Reasons for appearance of duplicate segments in the network:

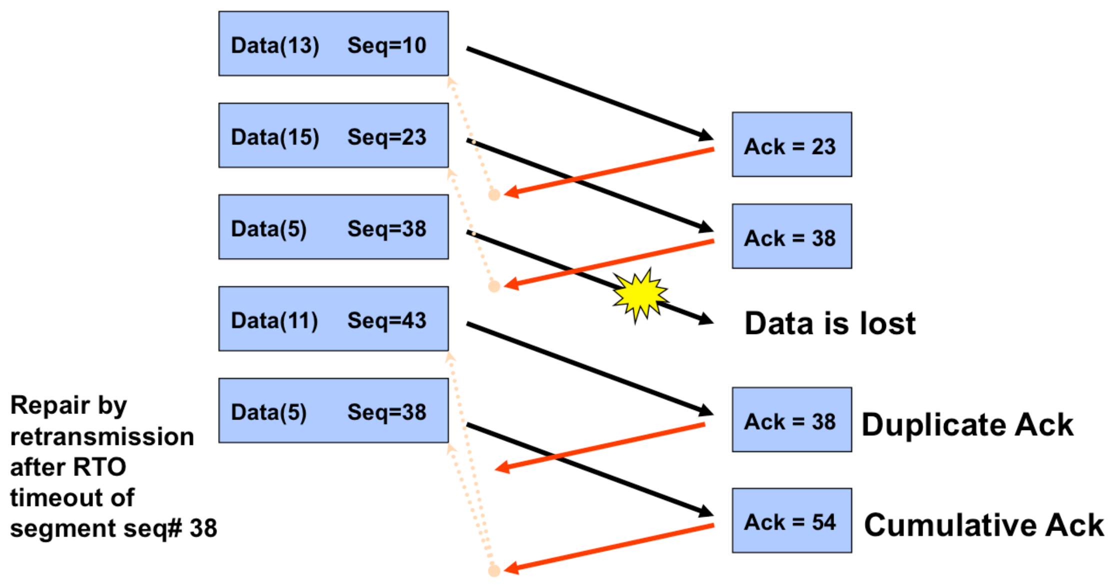
1.) Because original segment was lost: No problem in that case for the receiver. The retransmitted segment fills the gap and no duplicate segment seen at the receiver.

2.) Because ACK was lost or retransmit timeout expired: No problem again. The segment is recognized by the receiver as duplicate through the sequence number.

3.) Because original segment was delayed and timeout expired: No problem again. The segment is recognized by the receiver as duplicate through the sequence number.

The large sequence numbers space of 232 further helps to differentiate segments from old and new TCP in case the same sequence numbers happens to be used by the old and new TCP session. It will need 9h to send 232 bytes in a sequence with 2 Mbit/s before a wrap around will occur. Compare that to usual IP TTL = 128 seconds.

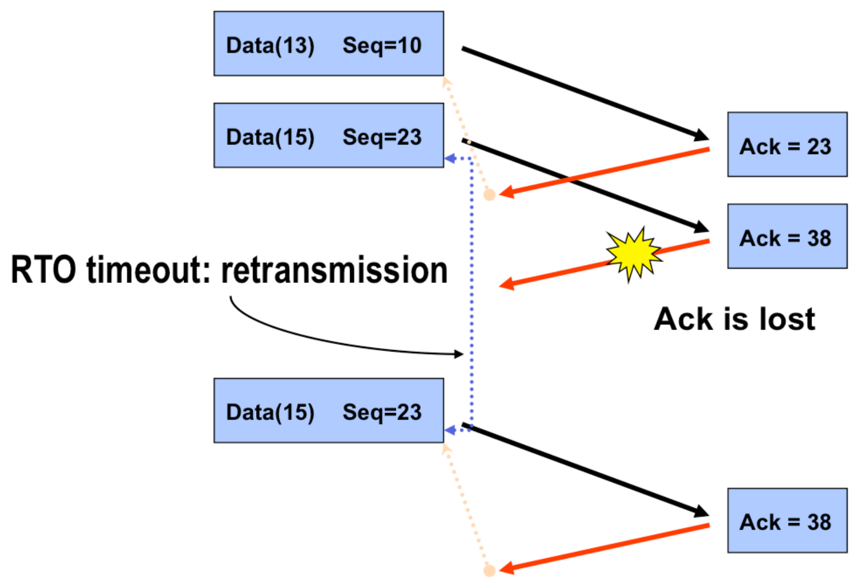
Duplicate Acknowledgement (new TCP)

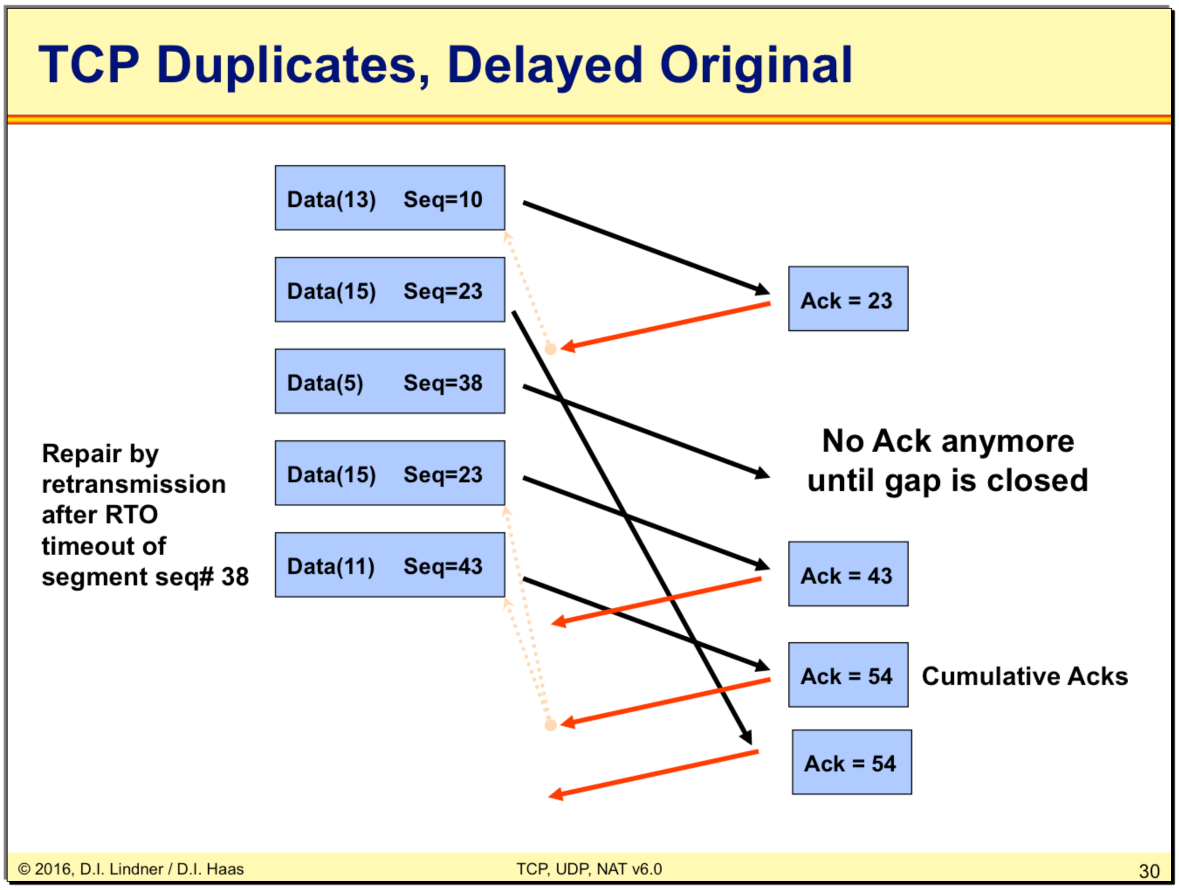


TCP, UDP, NAT v6.0 28

Instead of suspending ACKs in case of out-of-sequence arrival of segments, 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“ handled later in this chapter.

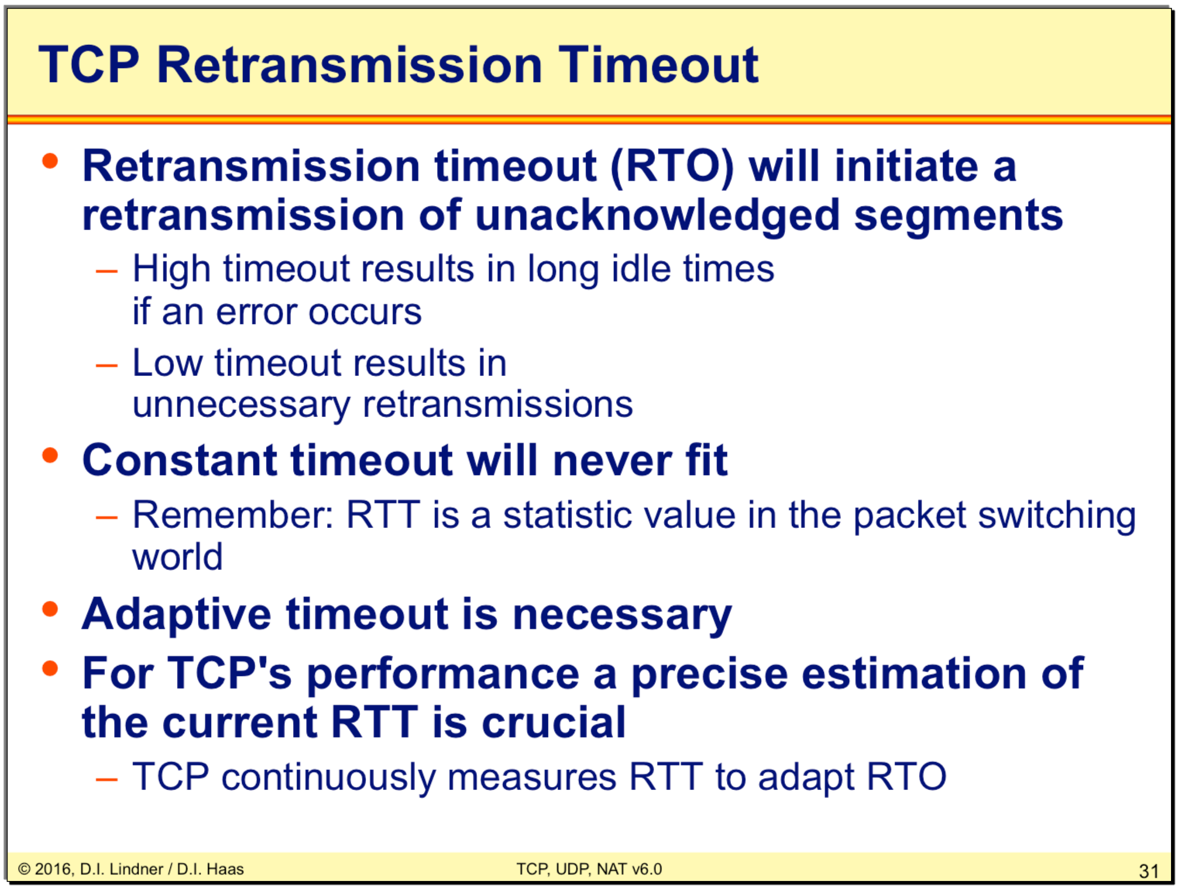
TCP Duplicates, Lost Acknowledgement





TCP Duplicates, Delayed Original

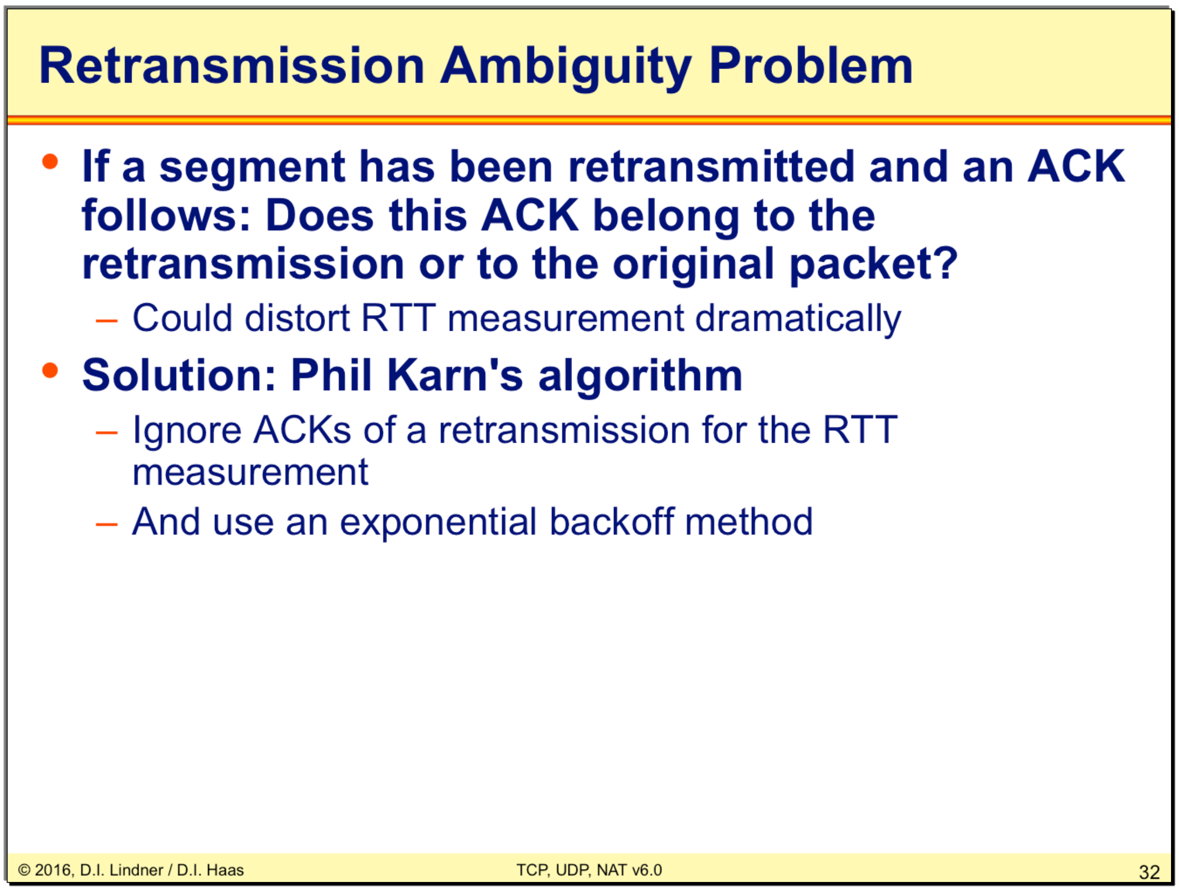
TCP, UDP, NAT v6.0 30



TCP Retransmission Timeout

- **Retransmission timeout (RTO) will initiate a retransmission of unacknowledged segments**
  - High timeout results in long idle times if an error occurs
  - Low timeout results in unnecessary retransmissions
- **Constant timeout will never fit**
  - Remember: RTT is a statistic value in the packet switching world
- **Adaptive timeout is necessary**
- **For TCP's performance a precise estimation of the current RTT is crucial**
  - TCP continuously measures RTT to adapt RTO

Value of retransmission timeout influences performance (timeout should be in relation to round trip delay = round-trip-time RTT). If the timeout is much larger than the actual RTT then in case an error occurred the sender waits to long in order to heal it by retransmission of the lost segment(s). If the timeout is much smaller than the actual RTT then even in the case of no error the sender retransmit a segment to early.



Retransmission Ambiguity Problem

- **If a segment 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.

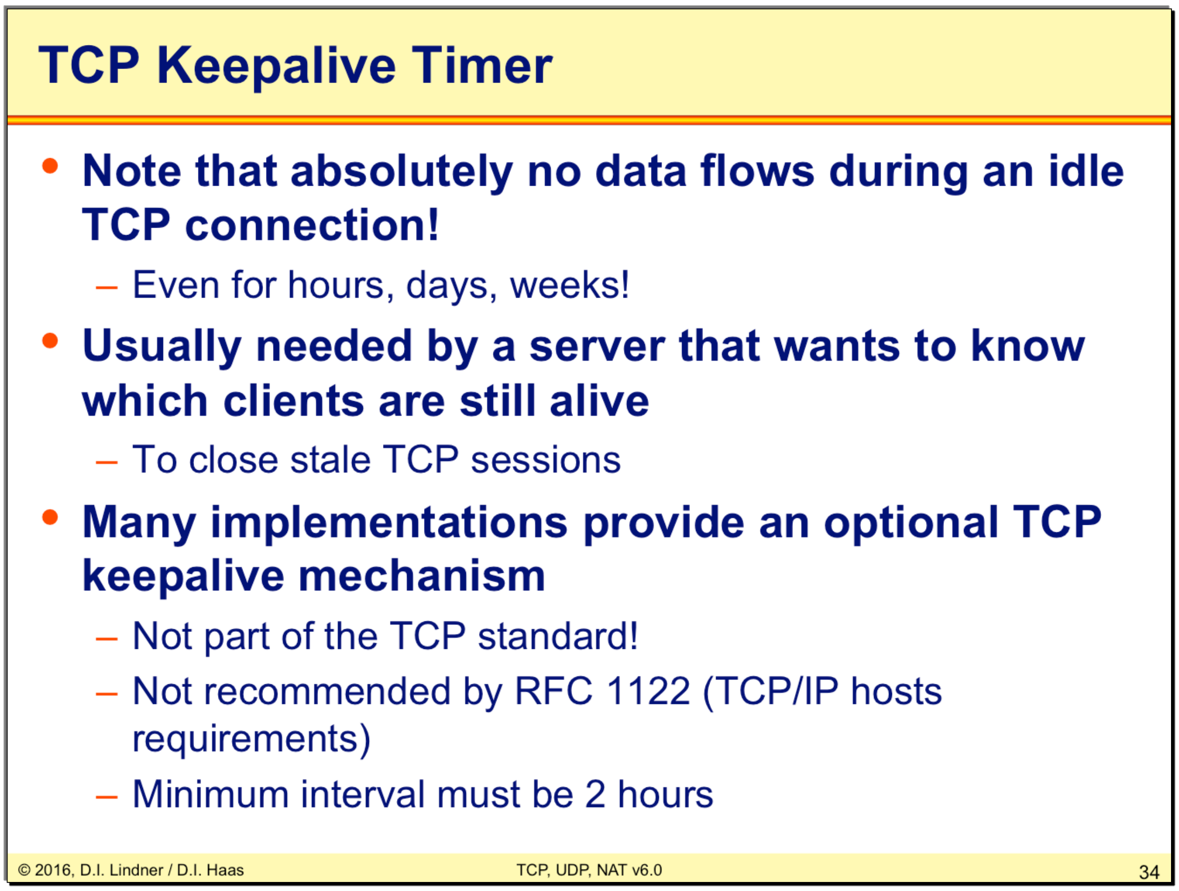
RTT Estimation

FYI

- **Originally a smooth RTT estimator was used (a low pass filter)**
  - M denotes the observed RTT (which is typically imprecise 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 $\beta = 2$

- **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 $h = 0.25$
  - $RTO = A + 4D$



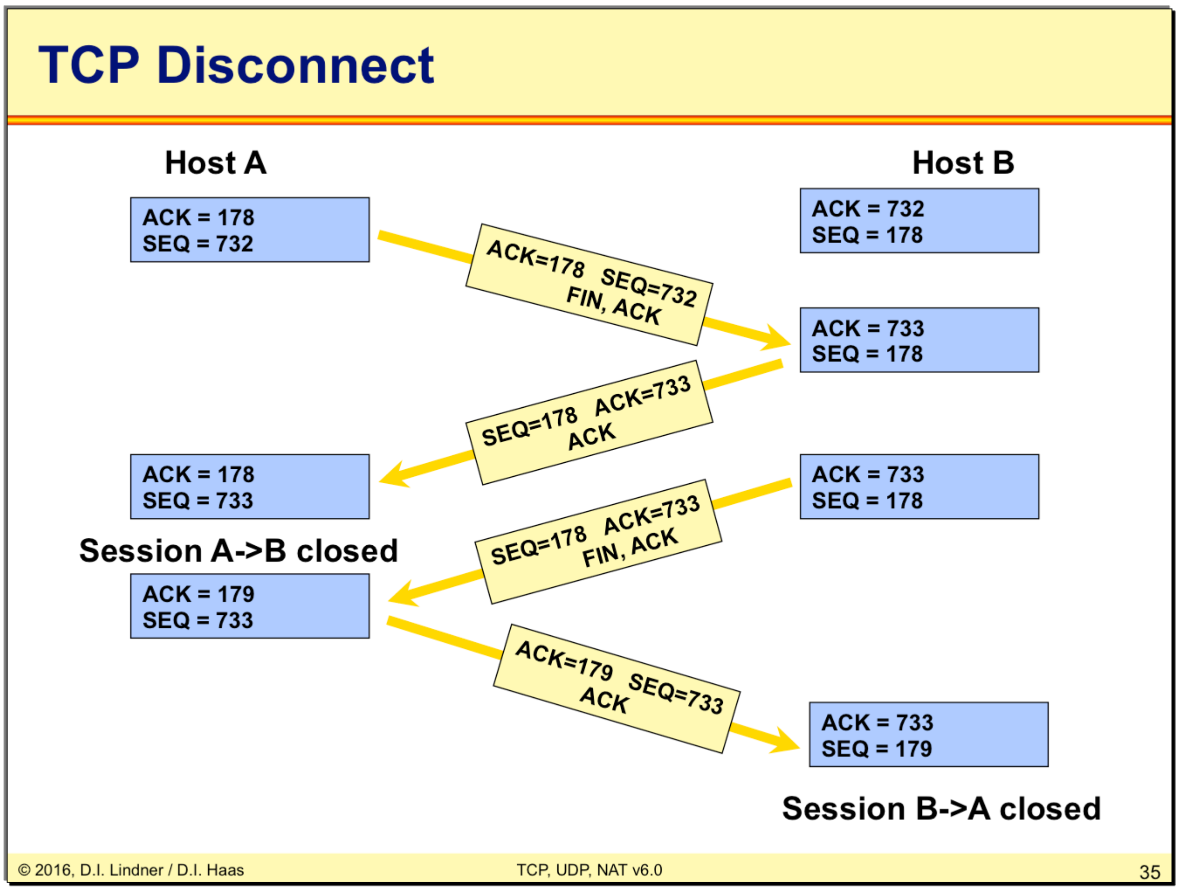
TCP Keepalive Timer

- **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 (TCP/IP hosts requirements)
  – Minimum interval must be 2 hours

TCP, UDP, NAT v6.0 34

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 mechanisms should be implemented at the application layer.

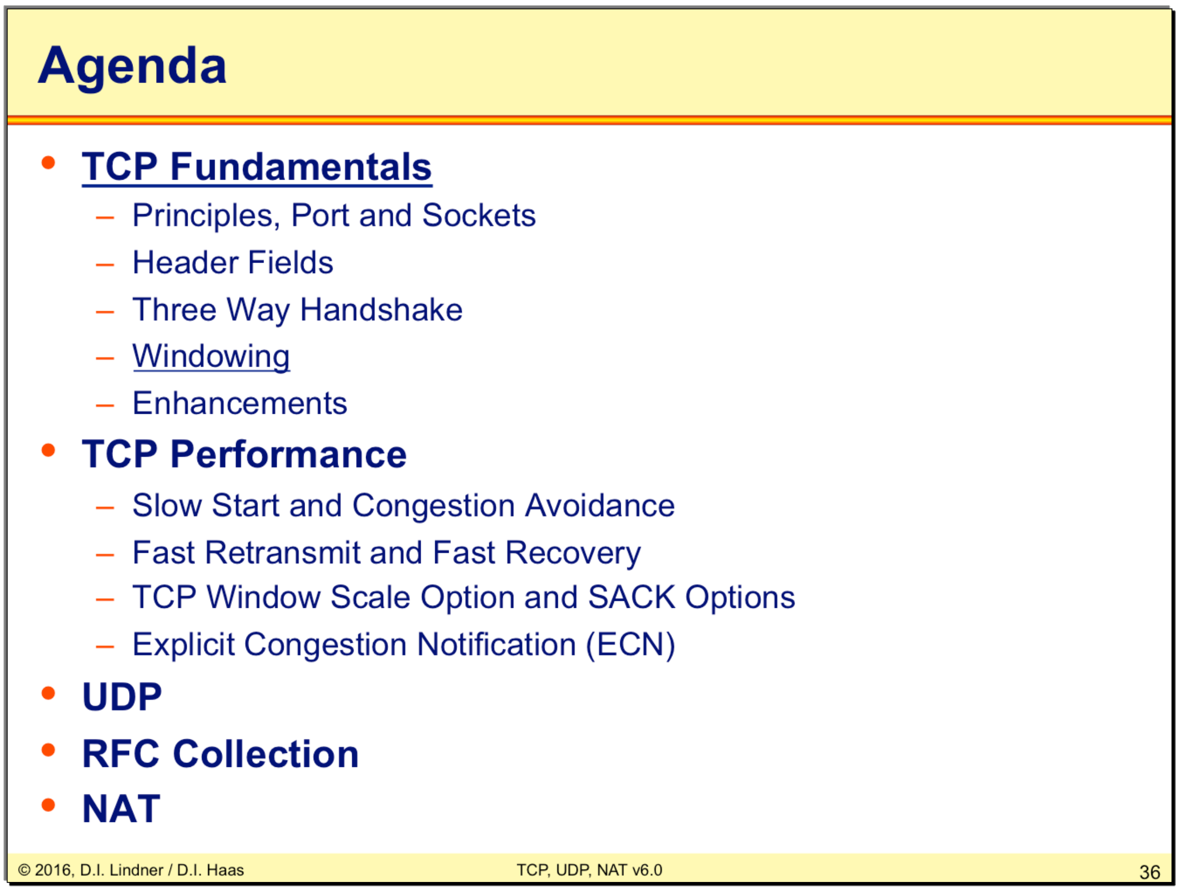


TCP Disconnect

TCP, UDP, NAT v6.0 35

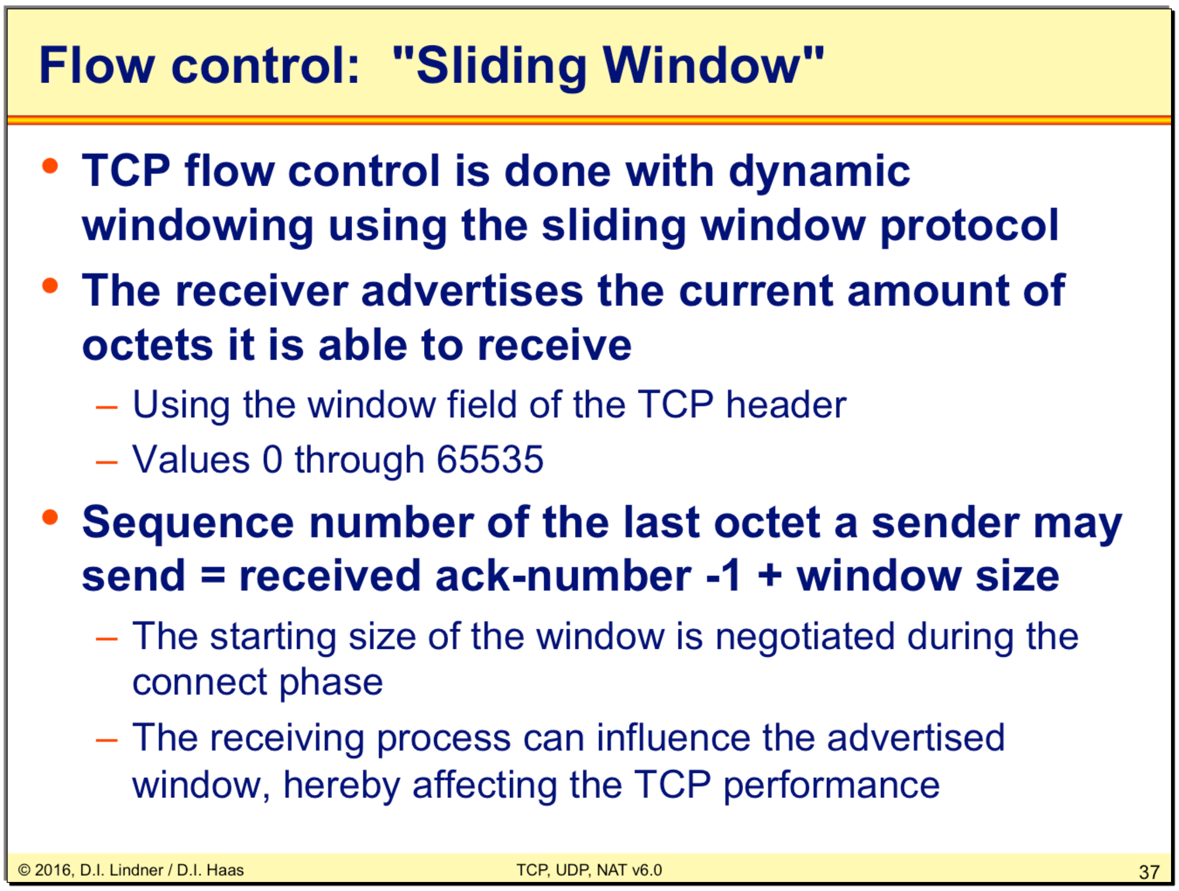
The “ordered” disconnect process is also a handshake, slightly similar to the 3-Way-Handshake. The exchange of FIN and ACK flags ensures, that both parties have received all octets.

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 in such a way ensures, that both parties have received all bytes. The RST flag can be used if an error occurs during the disconnect phase



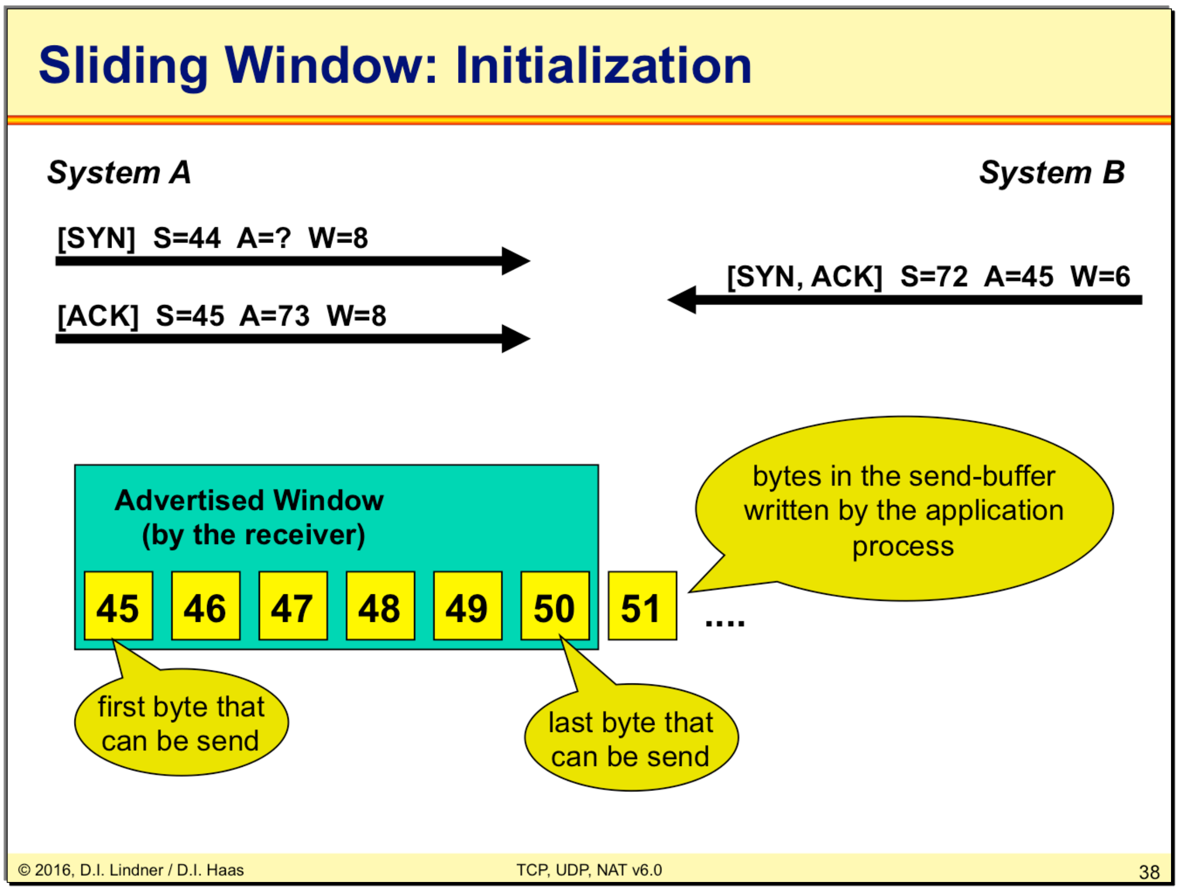
Agenda

- TCP Fundamentals
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- TCP Performance
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- UDP
- RFC Collection
- NAT



Flow control: "Sliding Window"

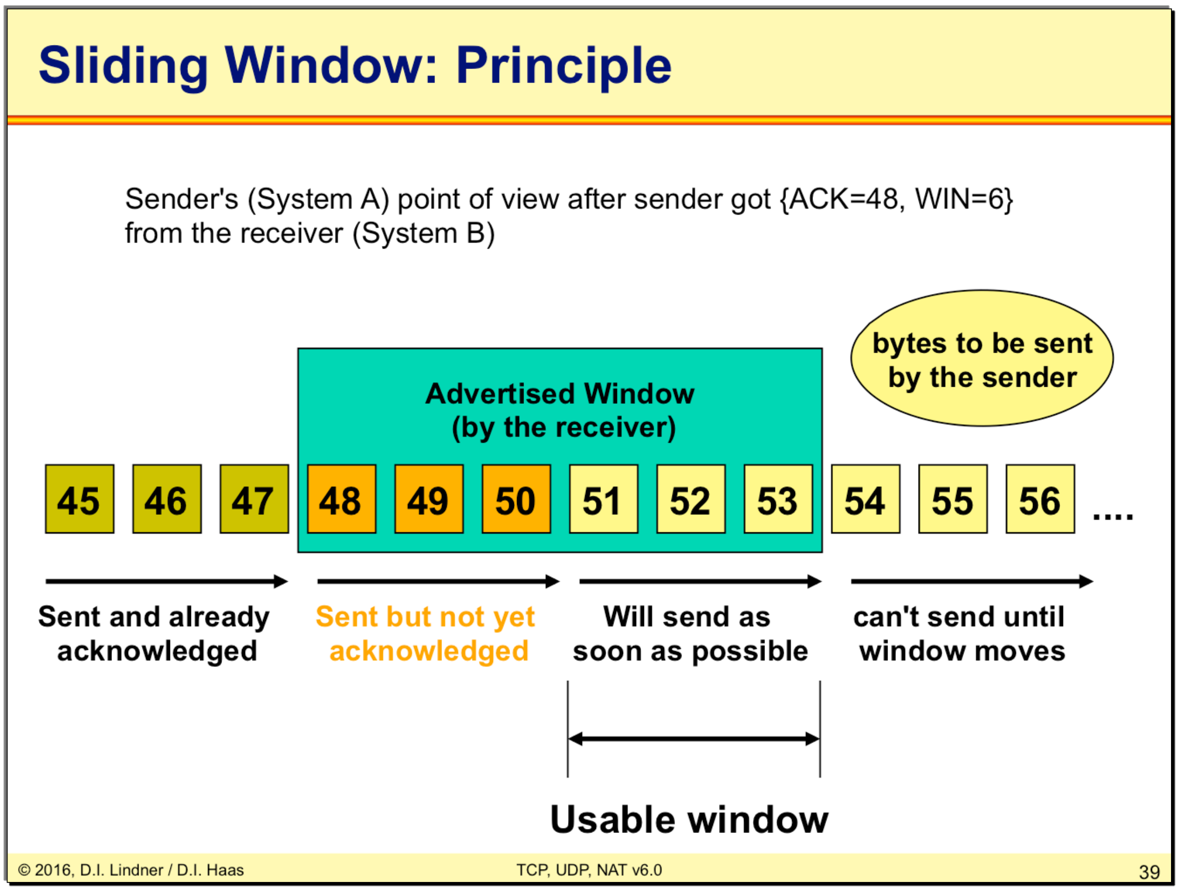
- **TCP flow control is done with dynamic windowing using the sliding window protocol**
- **The receiver advertises the current amount of octets it is able to receive**
  - Using the window field of the TCP header
  - Values 0 through 65535
- **Sequence number of the last octet a sender may send = received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

System A
System B
[SYN] S=44 A=? W=8
[SYN, ACK] S=72 A=45 W=6
[ACK] S=45 A=73 W=8
Advertised Window (by the receiver)
45
46
47
48
49
50
51
....
bytes in the send-buffer written by the application process
first byte that can be send
last byte that can be send

TCP, UDP, NAT v6.0 38



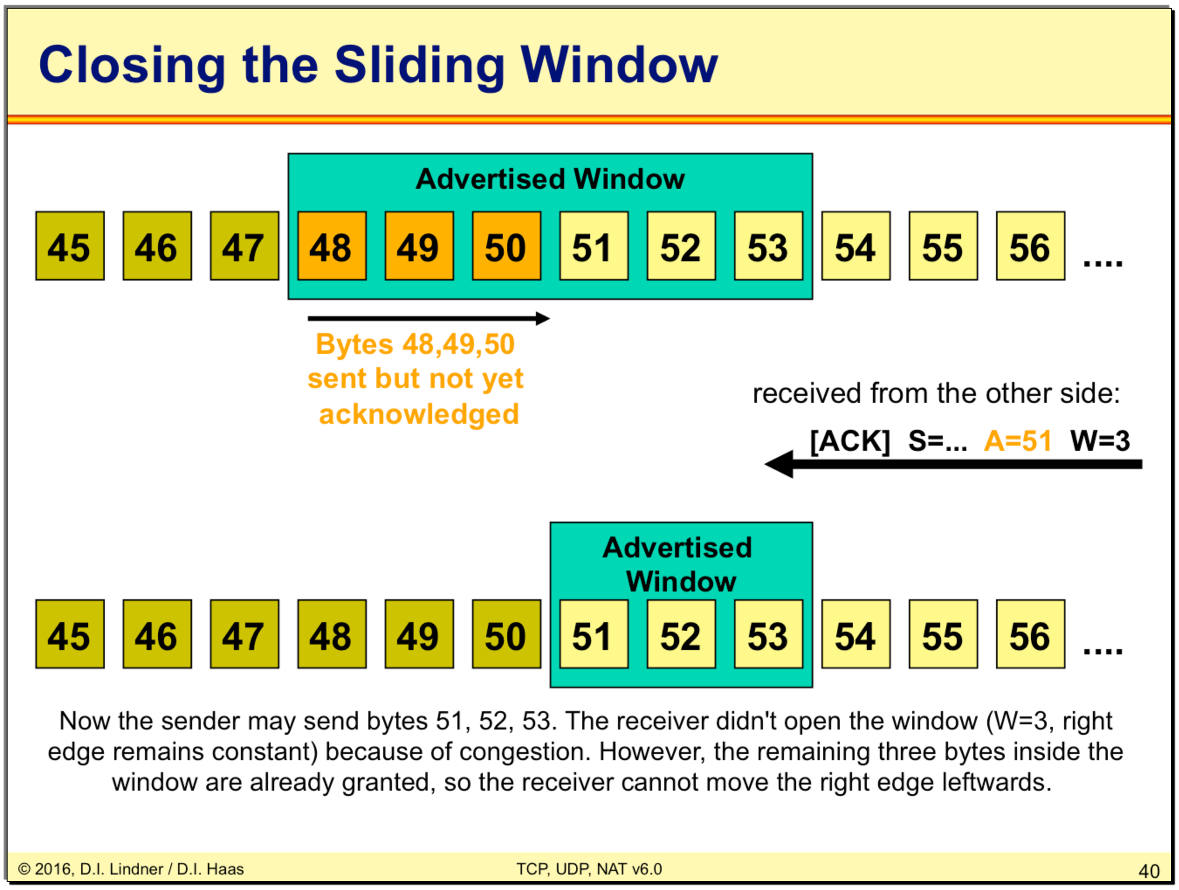
Sliding Window: Principle

Sender's (System A) point of view after sender got {ACK=48, WIN=6} from the receiver (System B)

bytes to be sent by the sender
Advertised Window (by the receiver)
45
46
47
48
49
50
51
52
53
54
55
56
....
Sent and already acknowledged
Sent but not yet acknowledged
Will send as soon as possible
can't send until window moves
Usable window

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 39

During the transmission the sliding window moves from left to right, as the receiver acknowledges data.



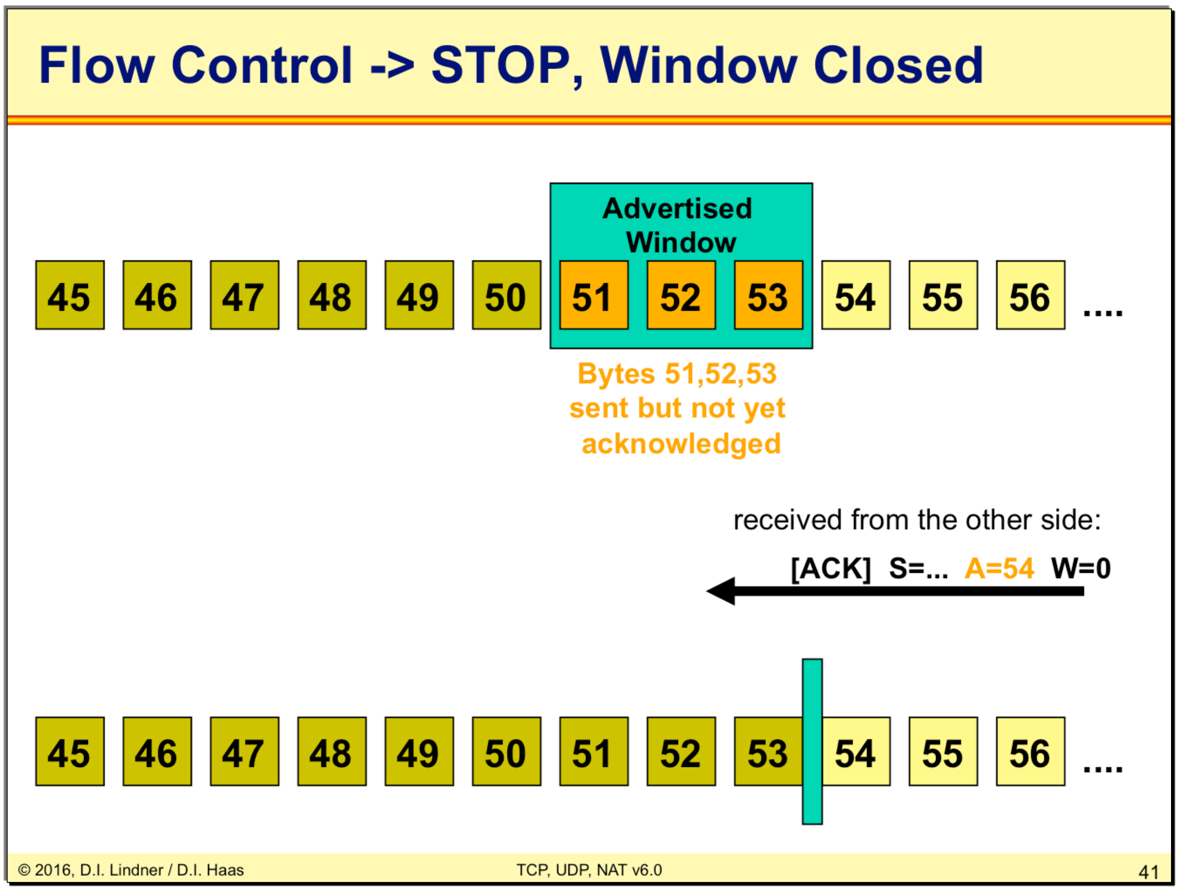
Closing the Sliding Window

Now the sender may send bytes 51, 52, 53. 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.

The relative motion of the two ends of the window *open* or *closes* the window.

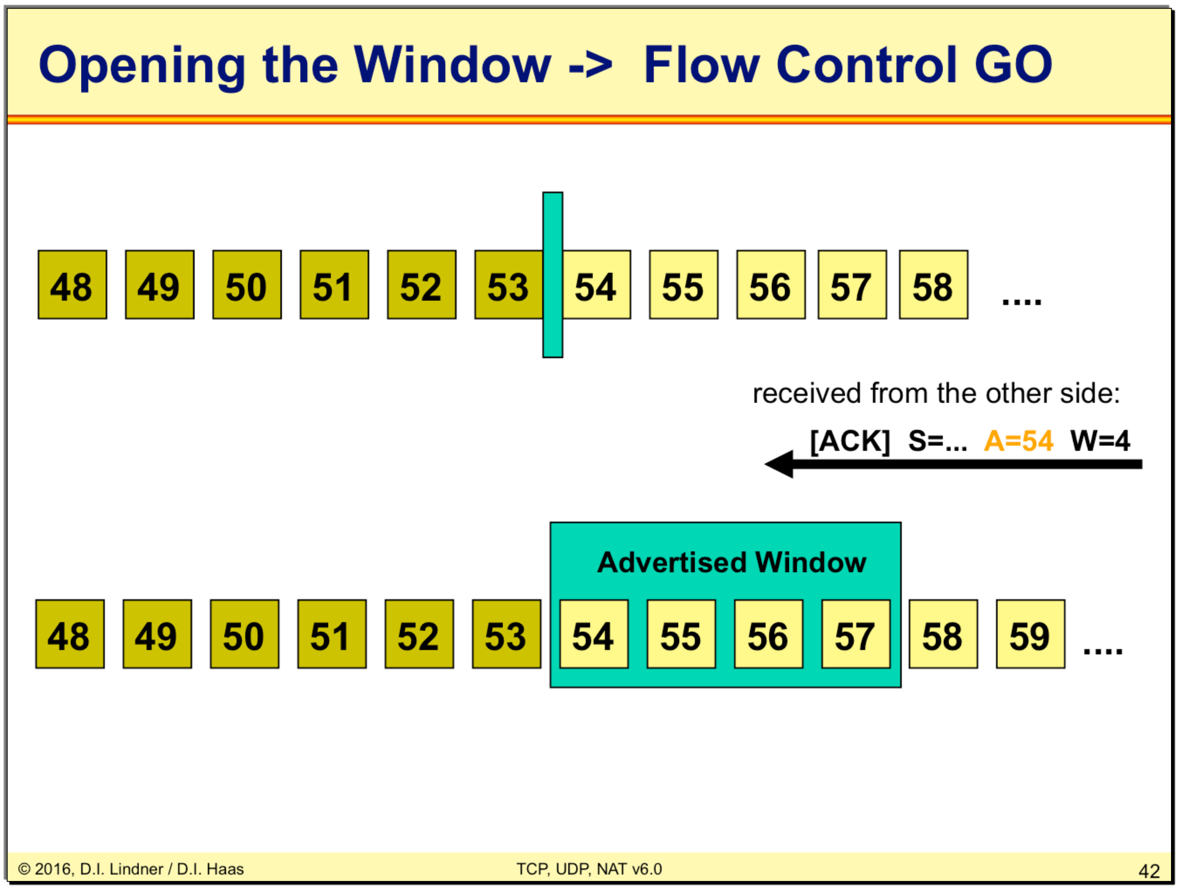
The window closes when data - already sent - is acknowledged (the left edge advances to the right).

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).

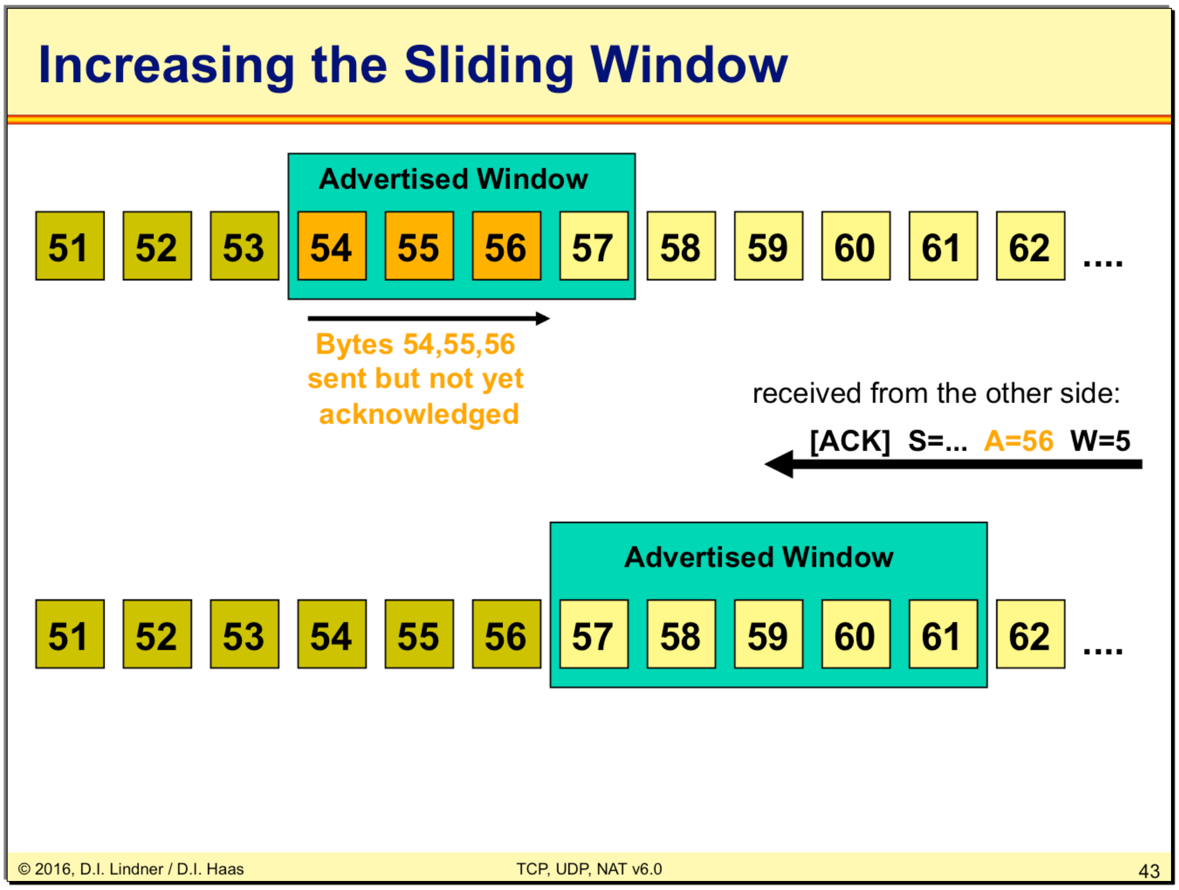


Flow Control -> STOP, Window Closed

If the left edge reaches the right edge, the sender stops transmitting data - *zero usable window*



Opening the Window -> Flow Control GO



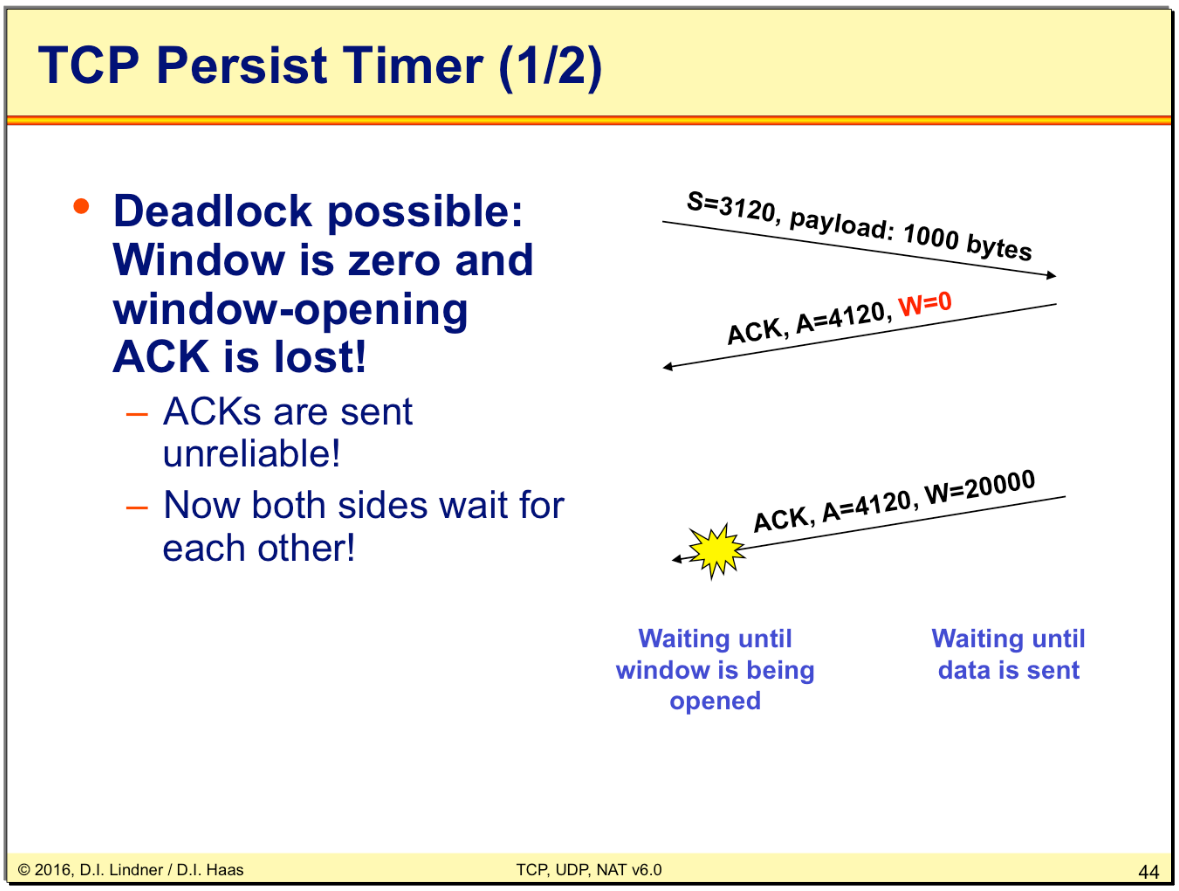
Increasing the Sliding Window

TCP, UDP, NAT v6.0 43

Some rules for handling sliding window in TCP:

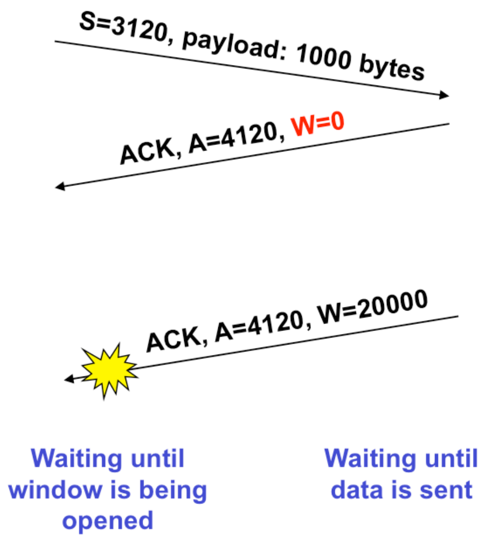
The right edge of the window must not move leftward! Would be called *shrinking* window. However, TCP must be able to cope with a peer doing that by e.g. resetting the TCP connection with RST flag.

The left edge of the window 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 silently discarded.

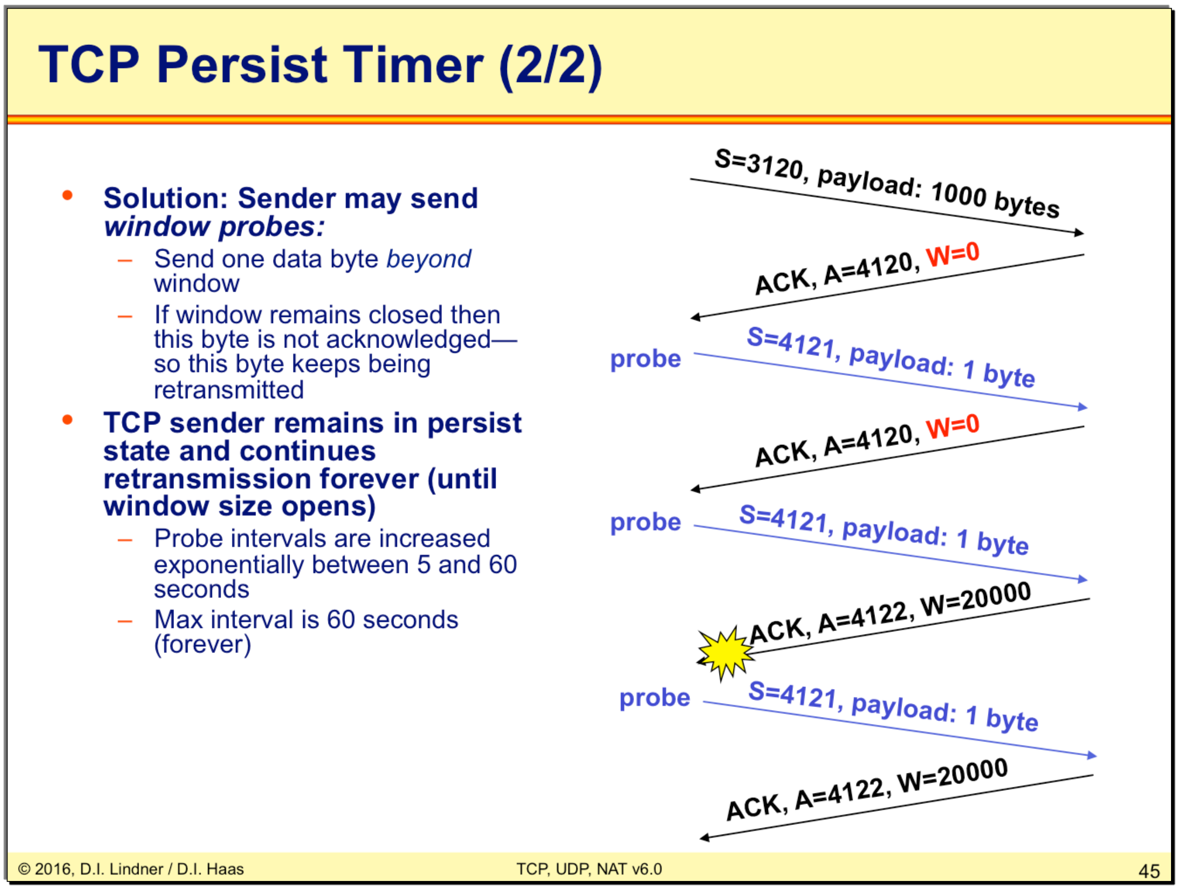


TCP Persist Timer (1/2)

- **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.

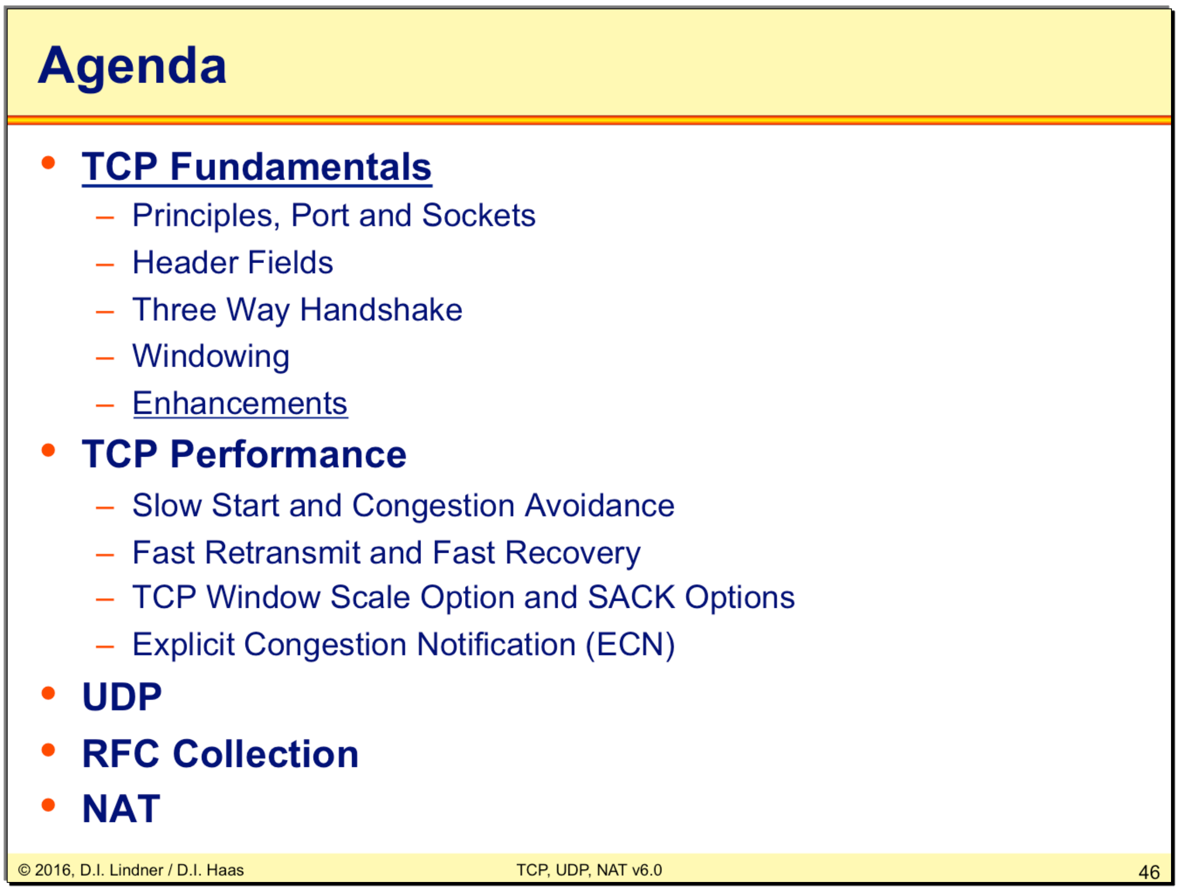


TCP Persist Timer (2/2)

- **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)

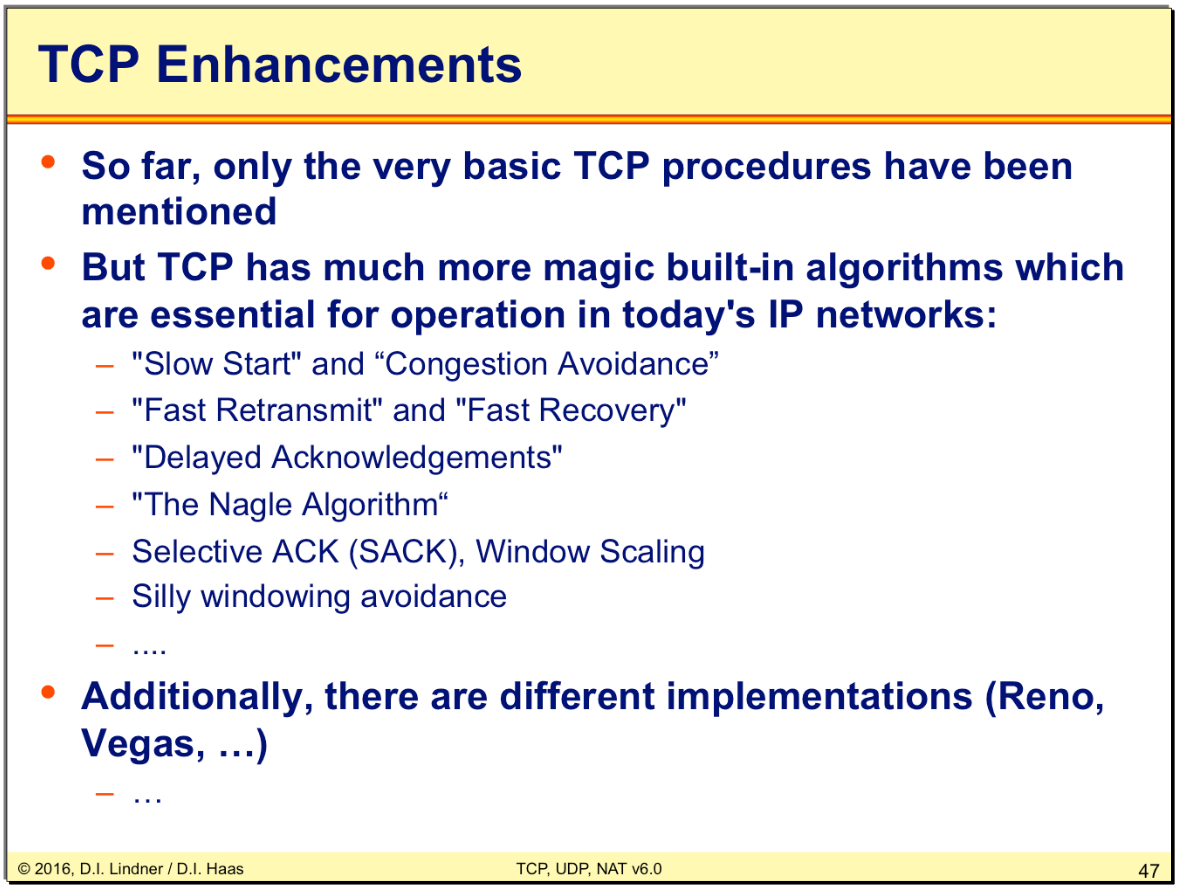
TCP, UDP, NAT v6.0 45

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.



Agenda

- TCP Fundamentals
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- TCP Performance
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- UDP
- RFC Collection
- NAT



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). It allows a sender-controlled flow control as add on to the receiver-controlled flow control based on the window field.

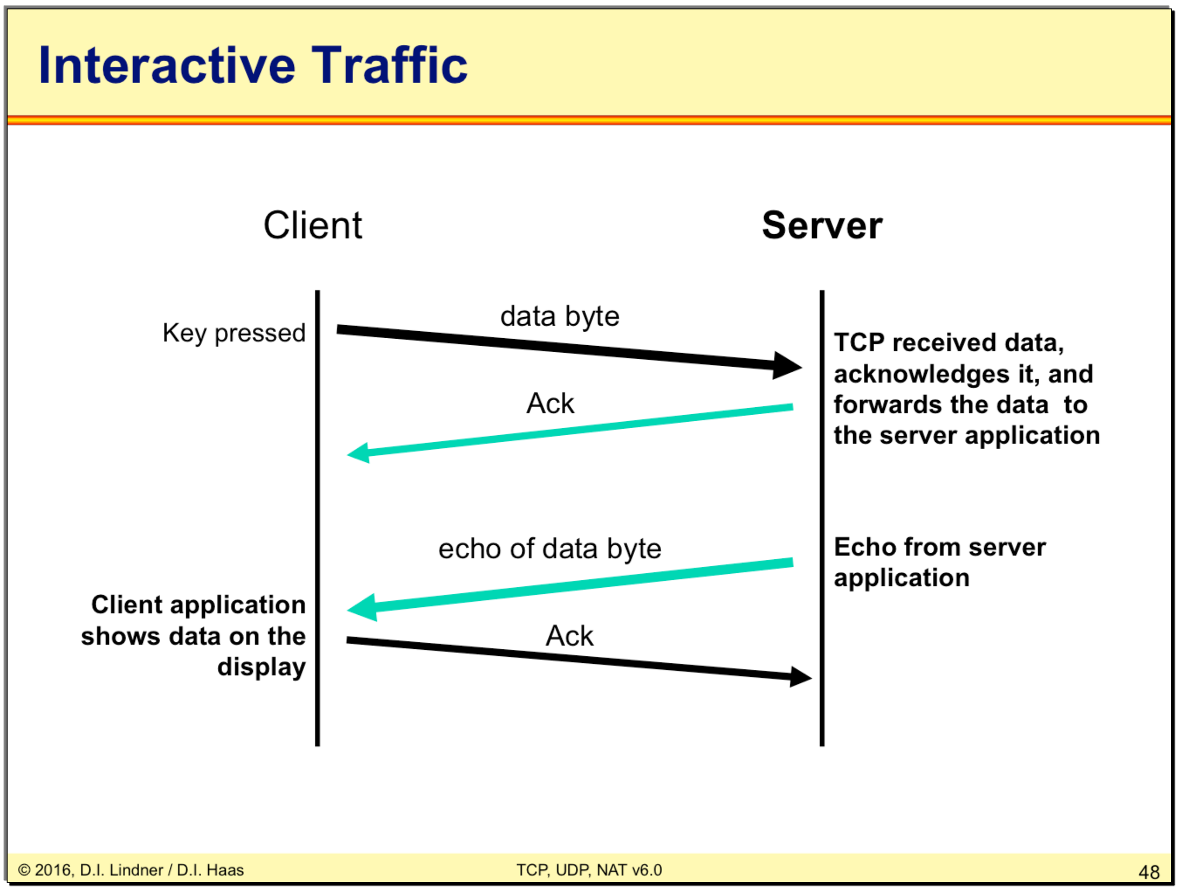
“Fast Retransmit” and “Fast Recovery” are mechanisms to avoid waiting for the timeout in case of retransmission and to avoid slow start after a fast retransmission.

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).

These topics are covered in the TCP performance chapter.

Delayed ACKs and Nagle algorithm is shown on the next slides.



Interactive Traffic

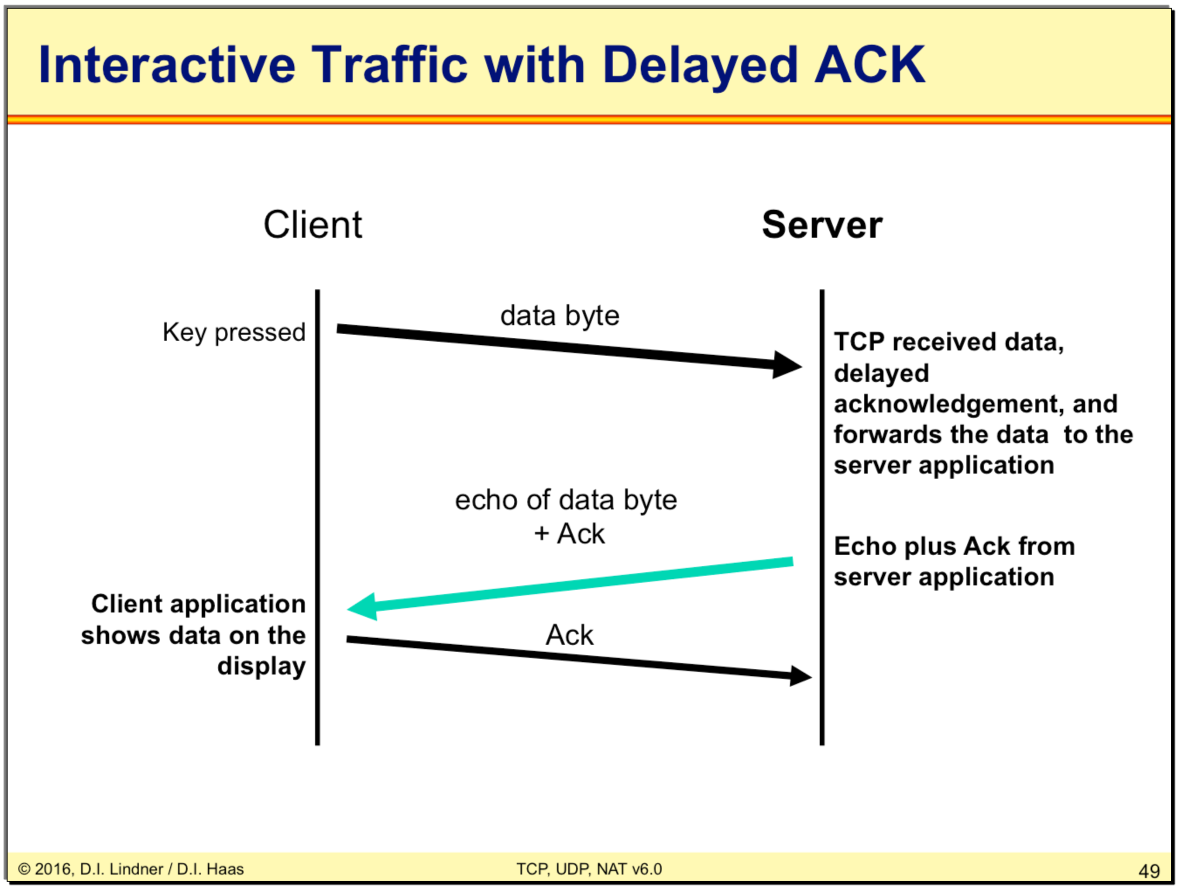
Client
Server
Key pressed
data byte
TCP received data, acknowledges it, and forwards the data to the server application
Ack
echo of data byte
Echo from server application
Client application shows data on the display
Ack

TCP, UDP, NAT v6.0 48

Immediate acknowledgements may cause an unnecessary amount of data transmissions.

Normally, an acknowledgement would be send immediately after the receiving of data.

But in <u>interactive</u> applications, the send-buffer at the receiver side gets filled by the application soon after an acknowledgement has been sent (e.g. Telnet echoes).



Interactive Traffic with Delayed ACK

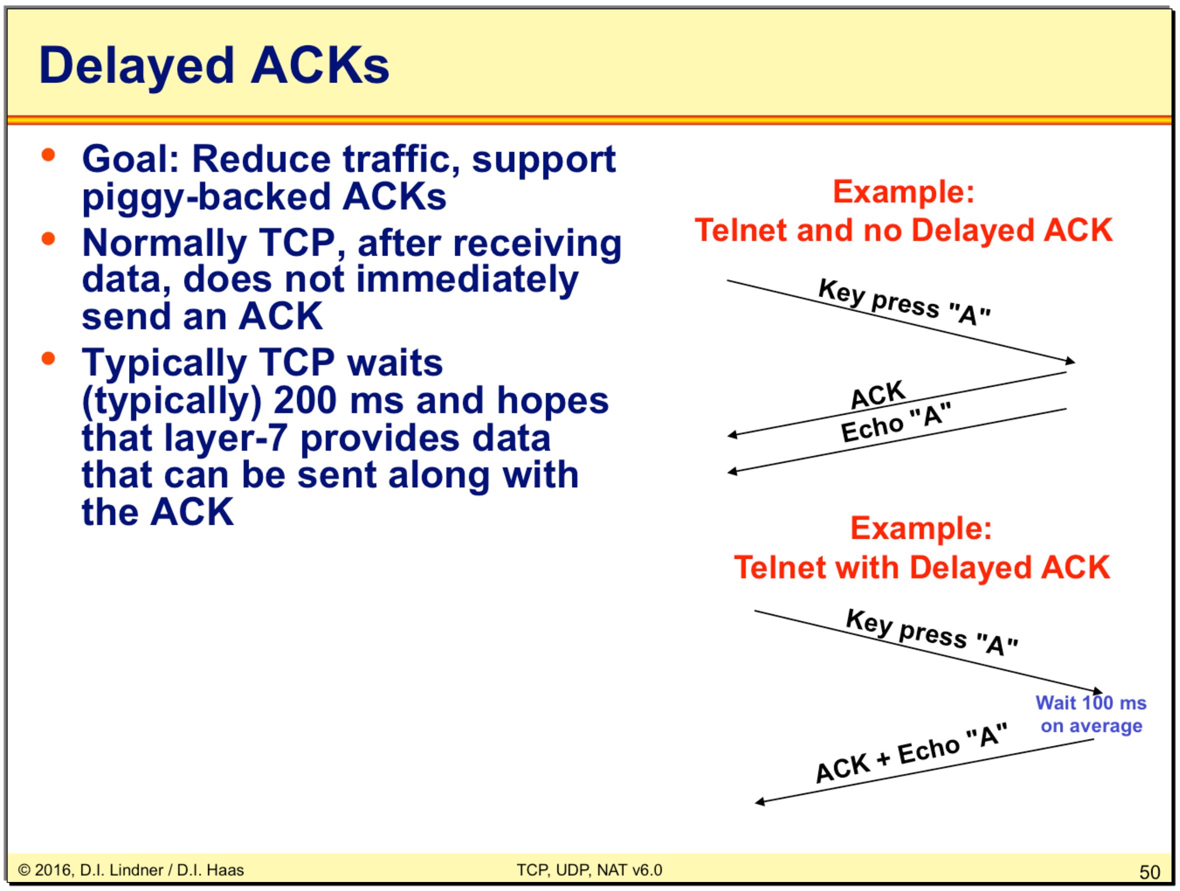
Client
Server
Key pressed
data byte
TCP received data, delayed acknowledgement, and forwards the data to the server application
echo of data byte + Ack
Echo plus Ack from server application
Client application shows data on the display
Ack

TCP, UDP, NAT v6.0 49

In order to support piggy-backed acknowledgements (i.e. Acks combined with user data), the TCP stack waits 200 ms before sending the delayed acknowledgement. During this time, the receiving application might also have data to send.

That is: 50% less (interactive!) traffic using delayed acknowledgements .

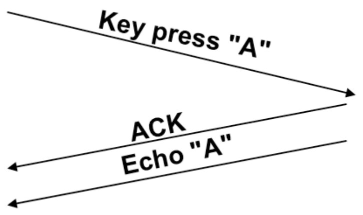
.



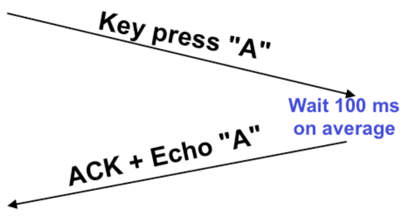
Delayed ACKs

- 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

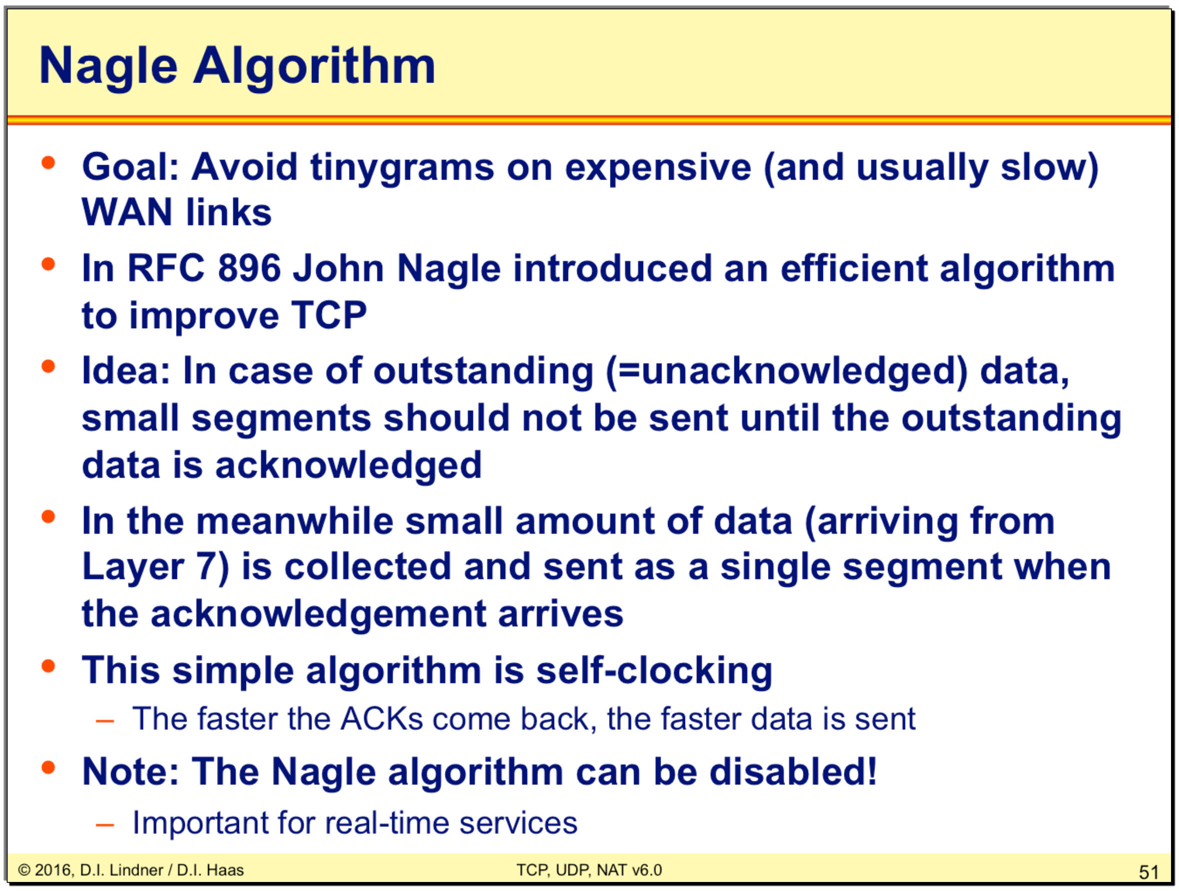


TCP, UDP, NAT v6.0 50

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.

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 real-time services

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 51

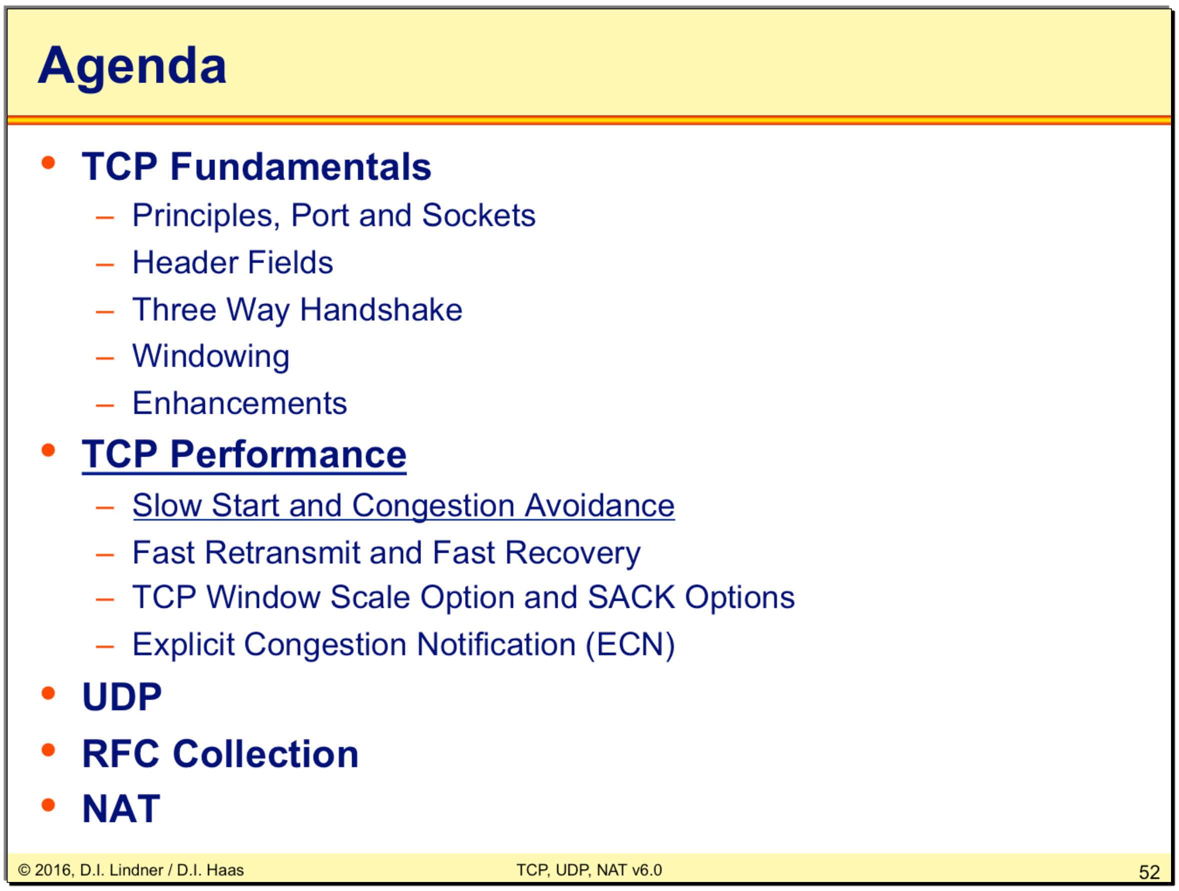
The Nagle algorithm tries to make WAN connections more efficient. We simply delay the segment transmission in order to collect more bytes from layer 7.

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.

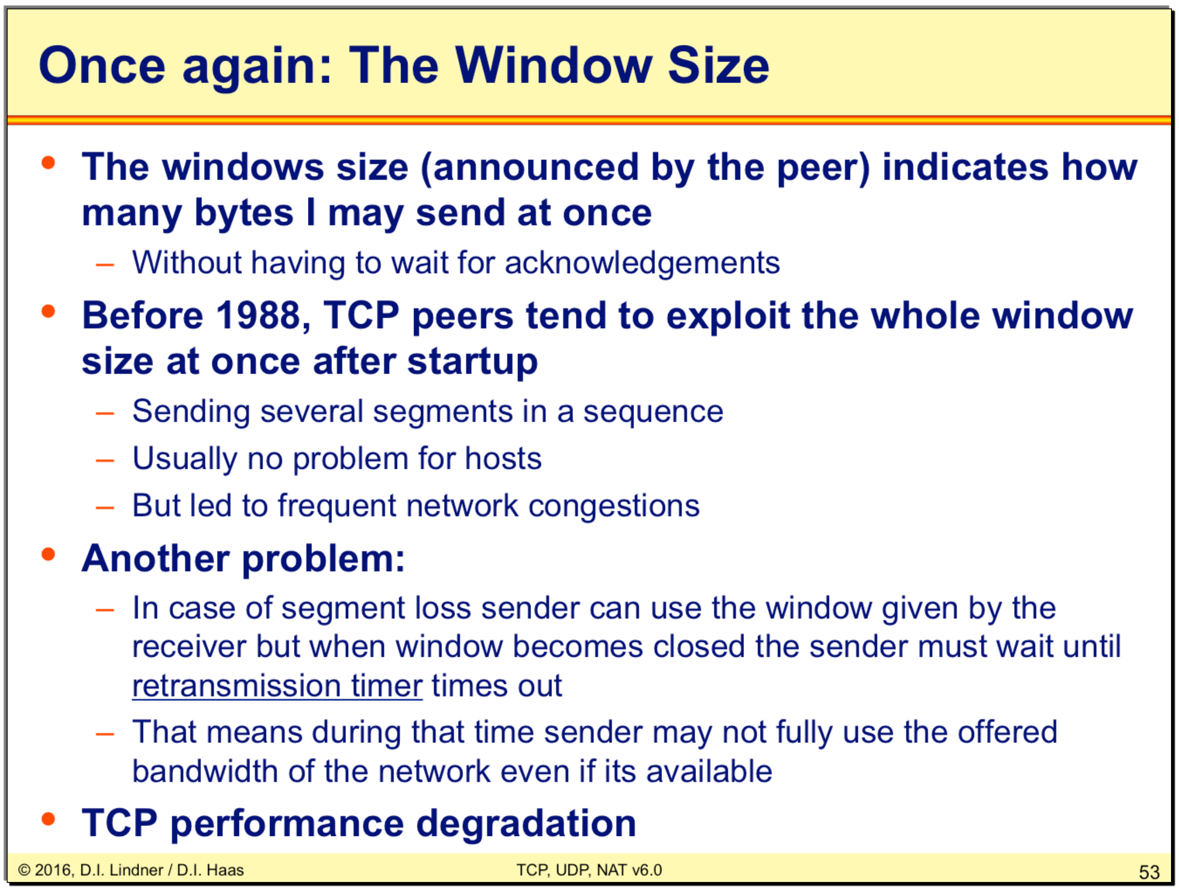
In this context, "small" means less than the segment size.

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

The socket API provides the symbol TCP_NODELAY.



Agenda



Once again: The Window Size

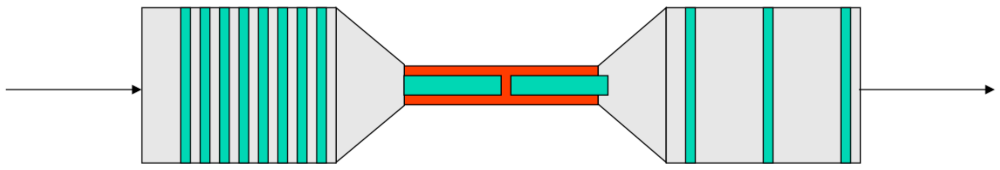
- **The windows size (announced by the peer) indicates how many bytes I may send at once**
  - Without having to wait for acknowledgements
- **Before 1988, TCP peers tend to exploit the whole window size at once after startup**
  - Sending several segments in a sequence
  - Usually no problem for hosts
  - But led to frequent network congestions
- **Another problem:**
  - In case of segment 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 may not fully use the offered bandwidth of the network even if its available
- **TCP performance degradation**

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 53

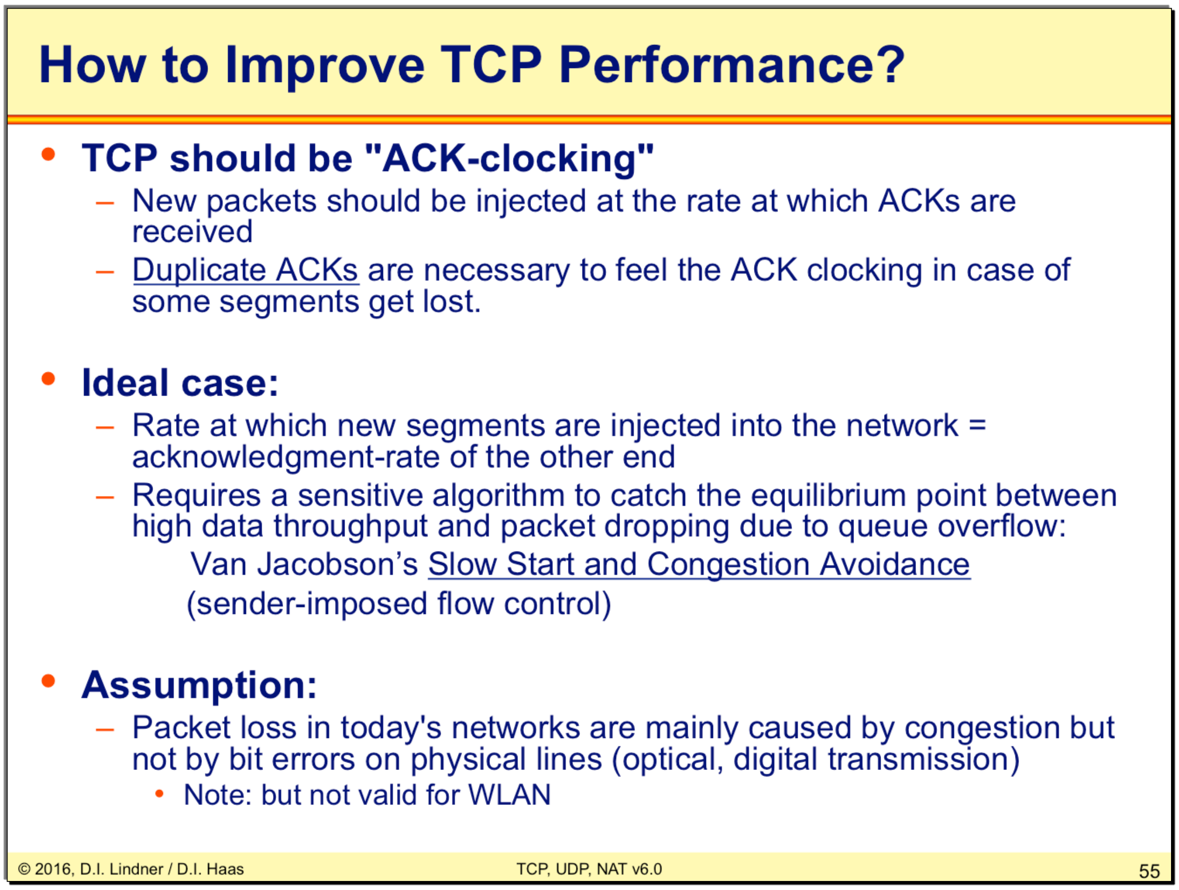
Note that hosts only need to deal with a single or a few TCP connections while network nodes such as routers and switches must transfer thousands, sometimes even millions of connections. Those nodes must queue datagrams and schedule them on outgoing interfaces (which might be slower than the inbound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

Congestion

- **Problem (buffer overflows) appears at bottleneck links**
  - Some intermediate router must queue packets
  - Queue overflow -> retransmission -> even more overflow!
  - Can't be solved by traditional receiver-imposed flow control (using the window field)



Pipe model 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.



How to Improve TCP Performance?

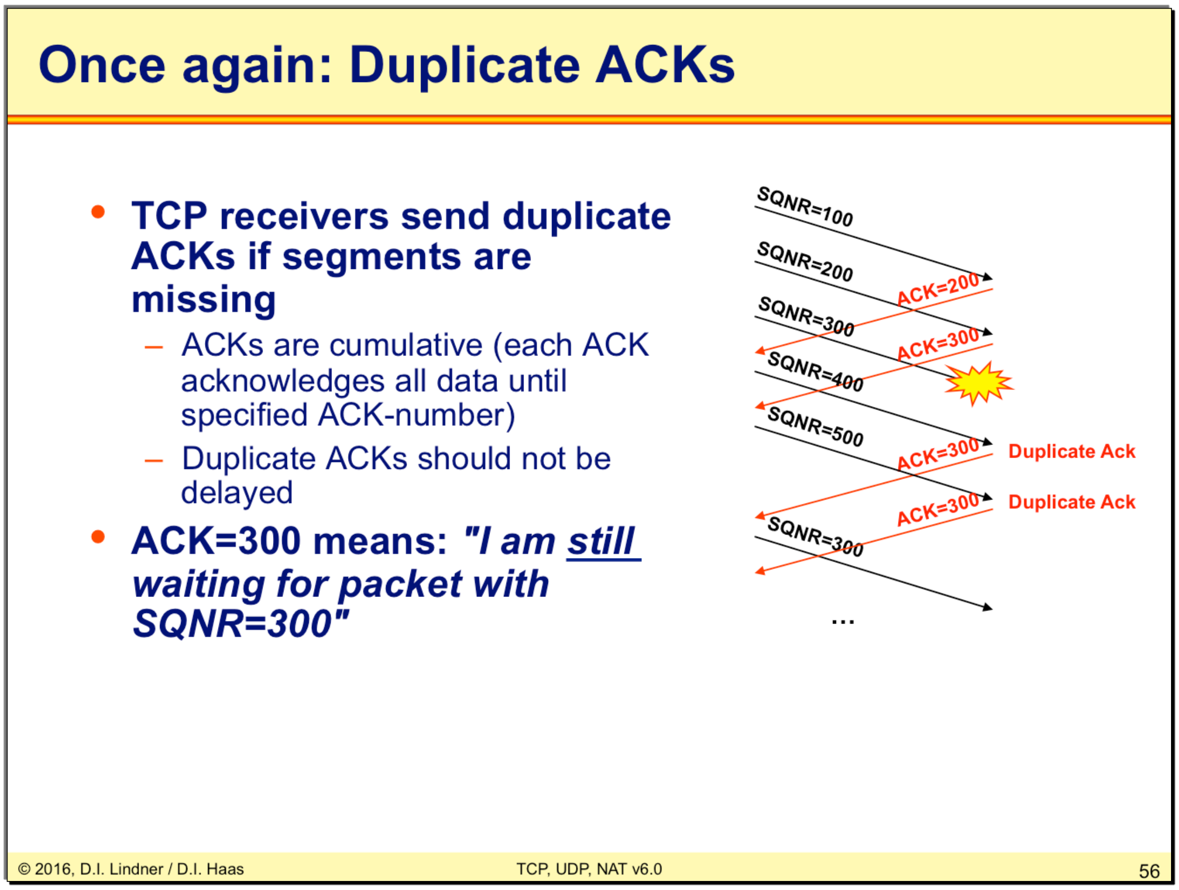
- **TCP should be "ACK-clocking"**
  - New packets should be injected at the rate at which ACKs are received
  - Duplicate ACKs are necessary to feel the ACK clocking in case of some segments get lost.
- **Ideal case:**
  - Rate at which new segments are injected into the network = acknowledgment-rate 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

    (sender-imposed flow control)
- **Assumption:**
  - Packet loss in today's networks are mainly caused by congestion but not by bit errors on physical lines (optical, digital transmission)
    - Note: but not valid for WLAN

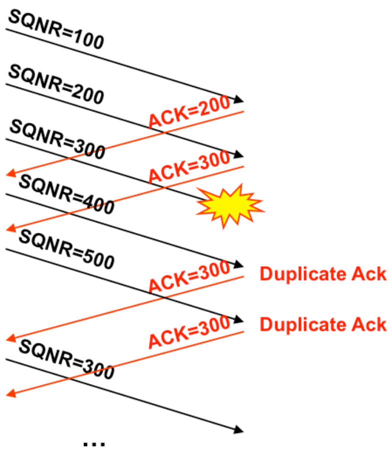
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 55

Using TCP the depths of the queues at network bottlenecks 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.



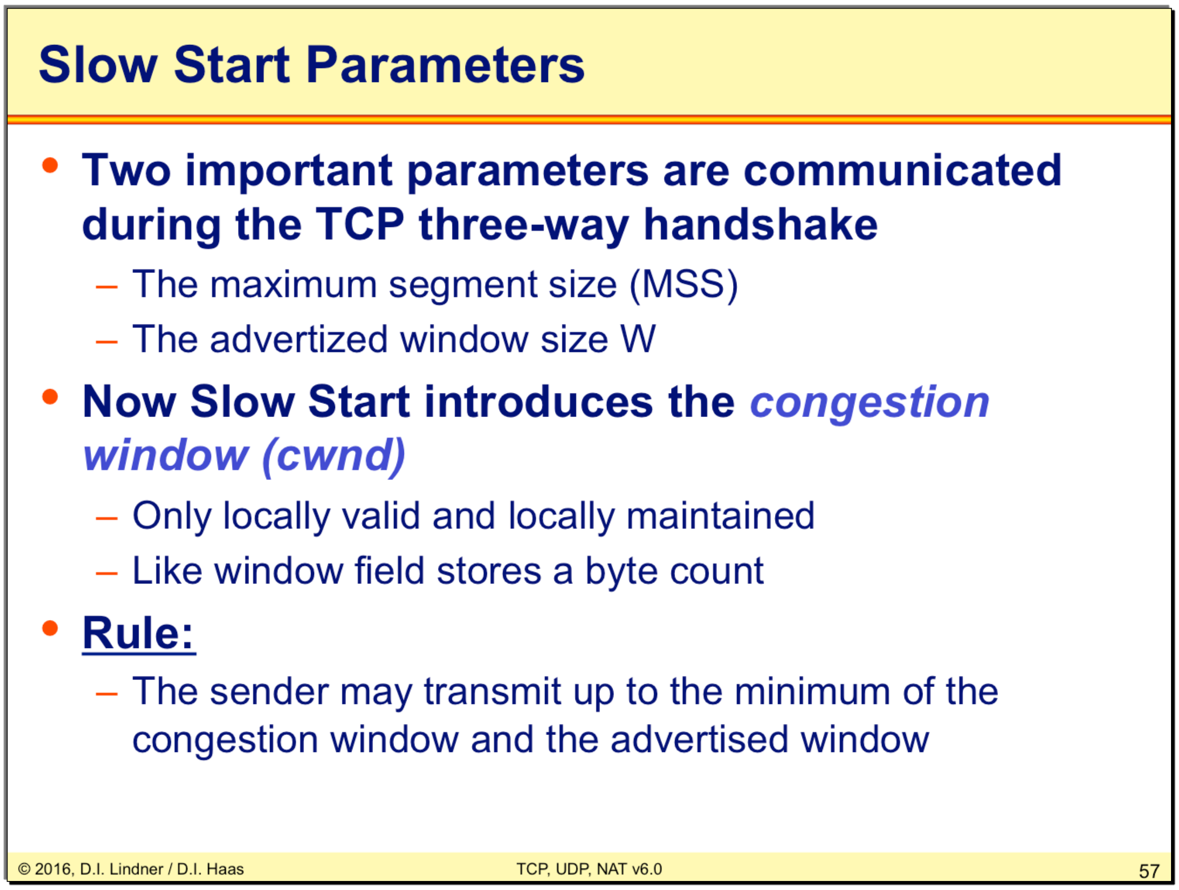
Once again: Duplicate ACKs

- **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 still waiting for packet with SQNR=300"***



TCP, UDP, NAT v6.0 56

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



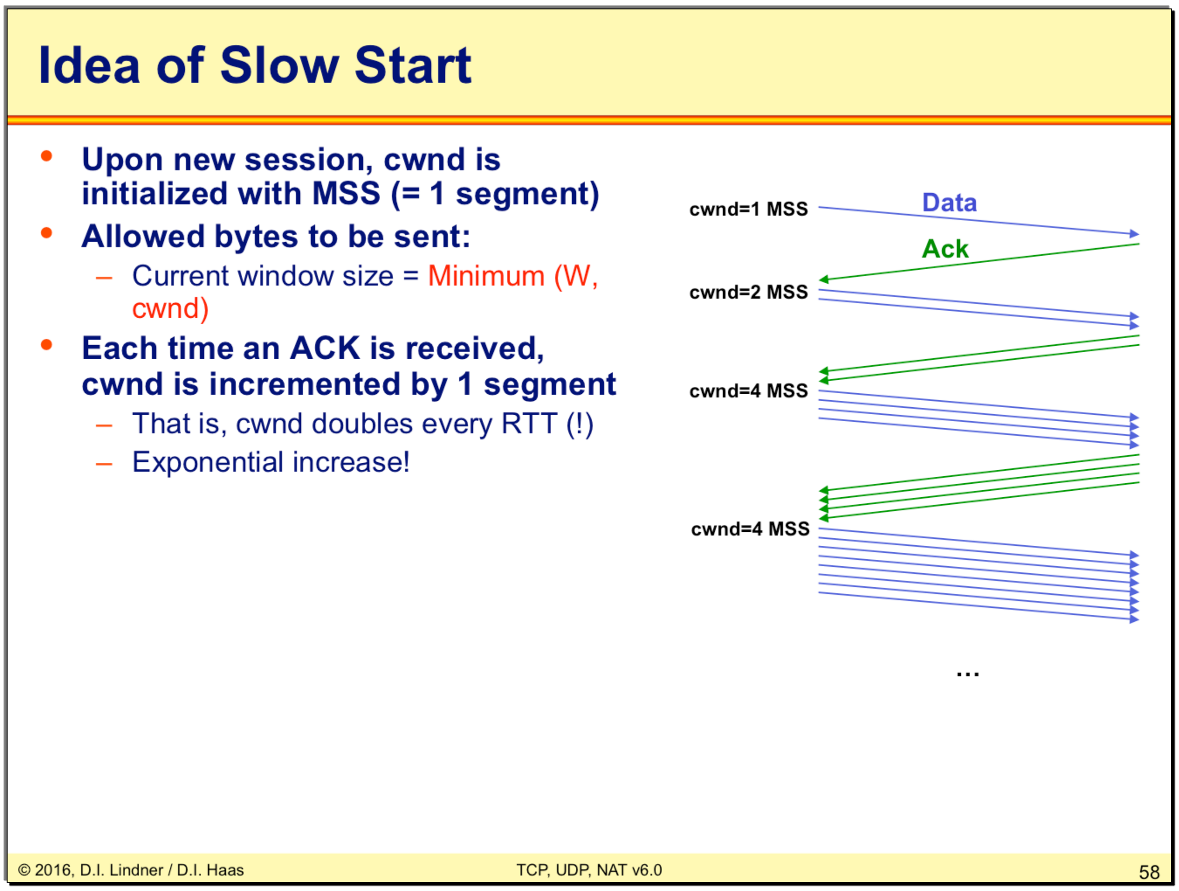
Slow Start Parameters

- **Two important parameters are communicated during the TCP three-way handshake**
  - The maximum segment size (MSS)
  - The advertized window size W
- **Now Slow Start introduces the *congestion window (cwnd)***
  - Only locally valid and locally maintained
  - Like window field stores a byte count
- **Rule:**
  - The sender may transmit up to the minimum of the congestion window and the advertised window

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 57

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 its 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

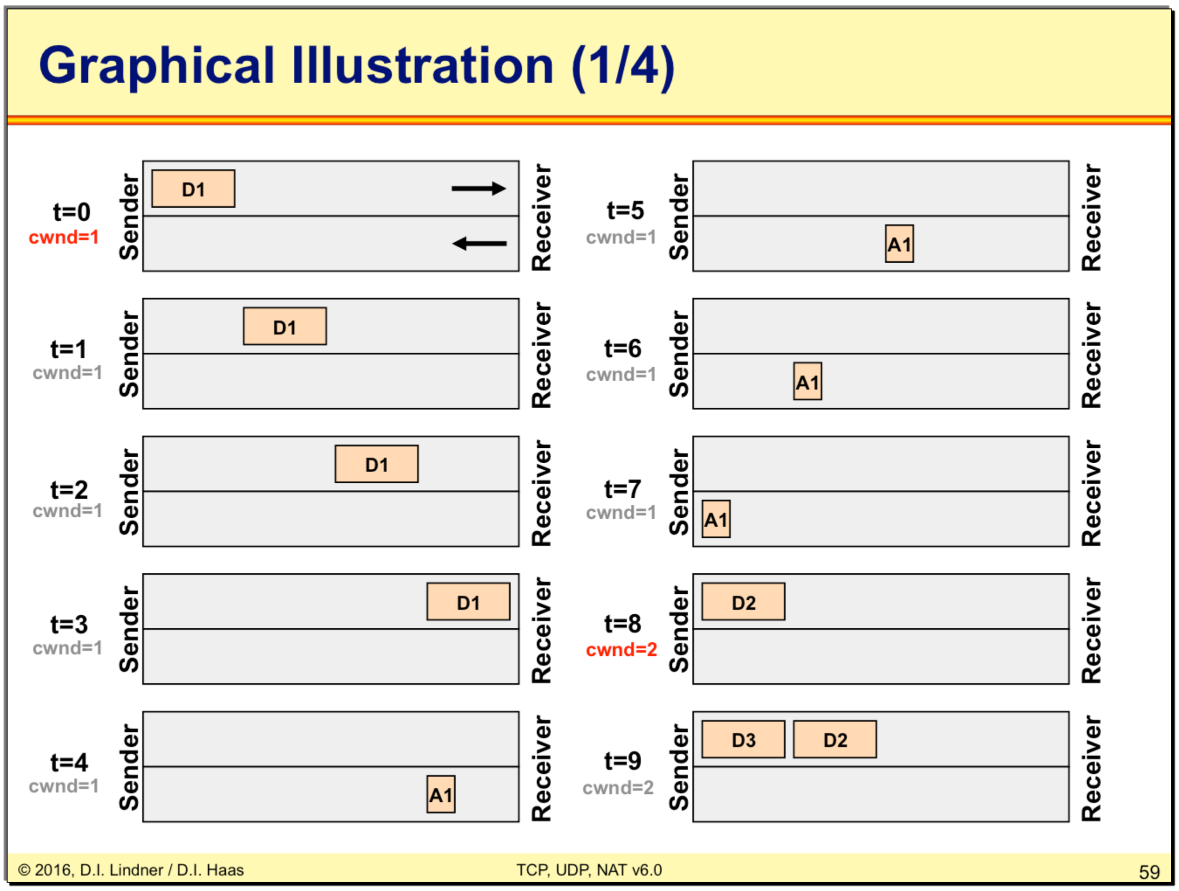
- **Upon new session, cwnd is initialized with MSS (= 1 segment)**
- **Allowed bytes to be sent:**
  - Current window size = Minimum (W, cwnd)
- **Each time an ACK is received, cwnd is incremented by 1 segment**
  - That is, cwnd doubles every RTT (!)
  - Exponential increase!

cwnd=1 MSS
Data
Ack
cwnd=2 MSS
cwnd=4 MSS
cwnd=4 MSS
...

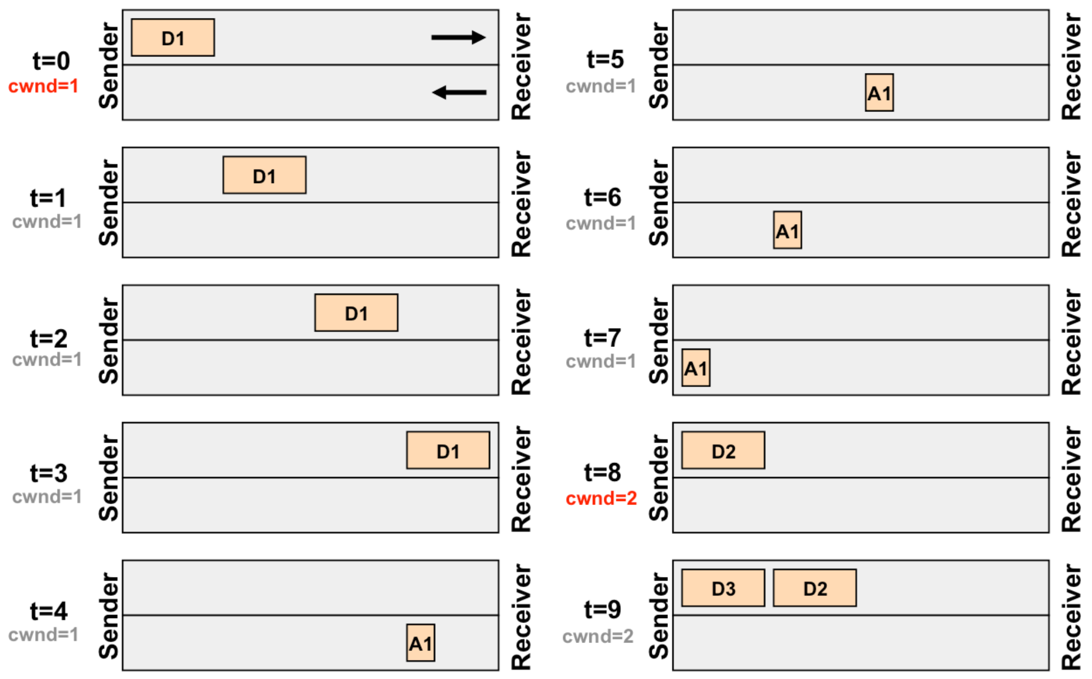
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 58

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!



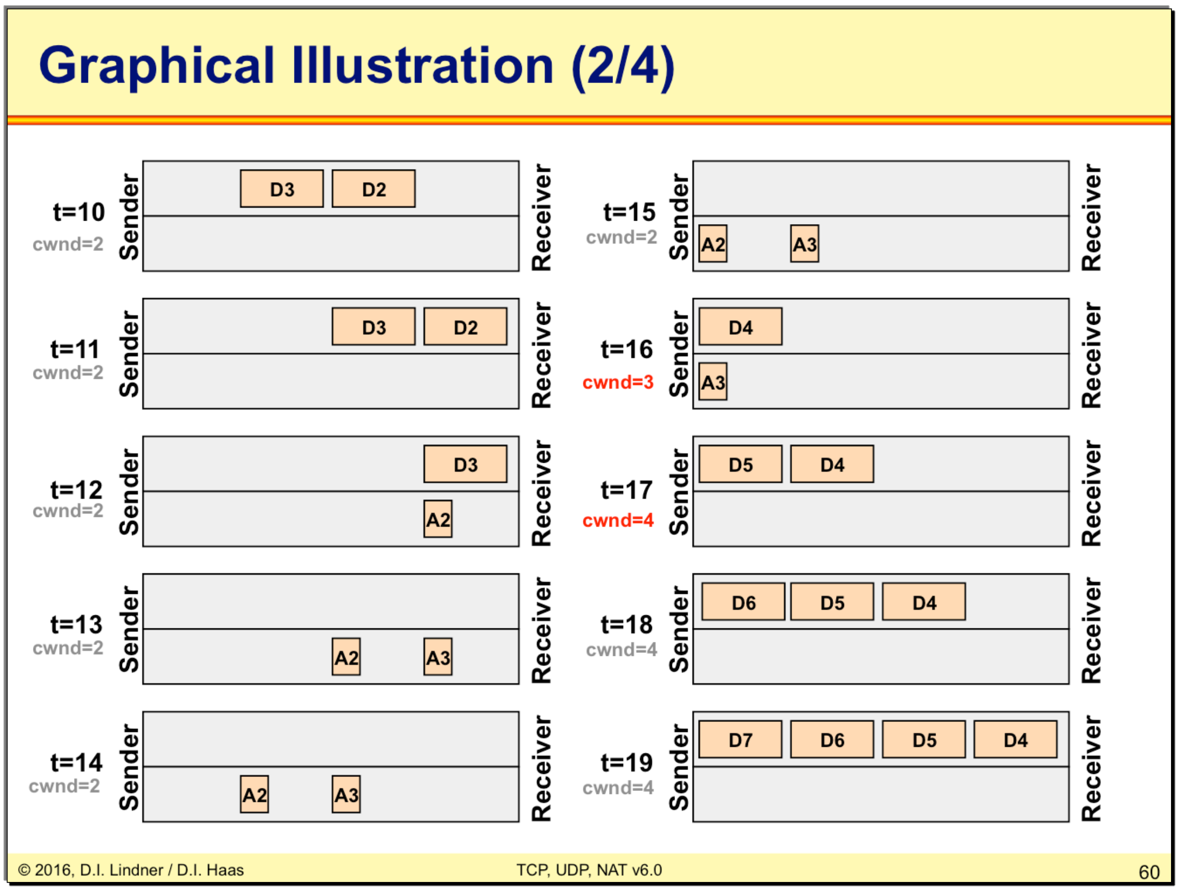
Graphical Illustration (1/4)



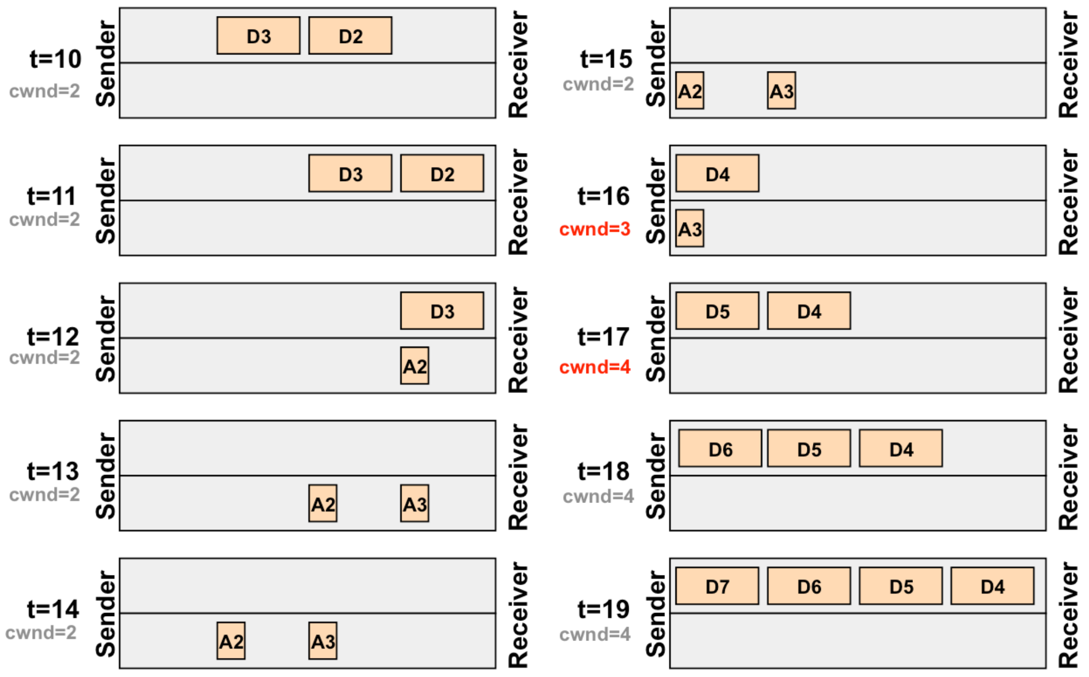
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 59

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

Observe how the cwnd is increased upon reception of ACKs.

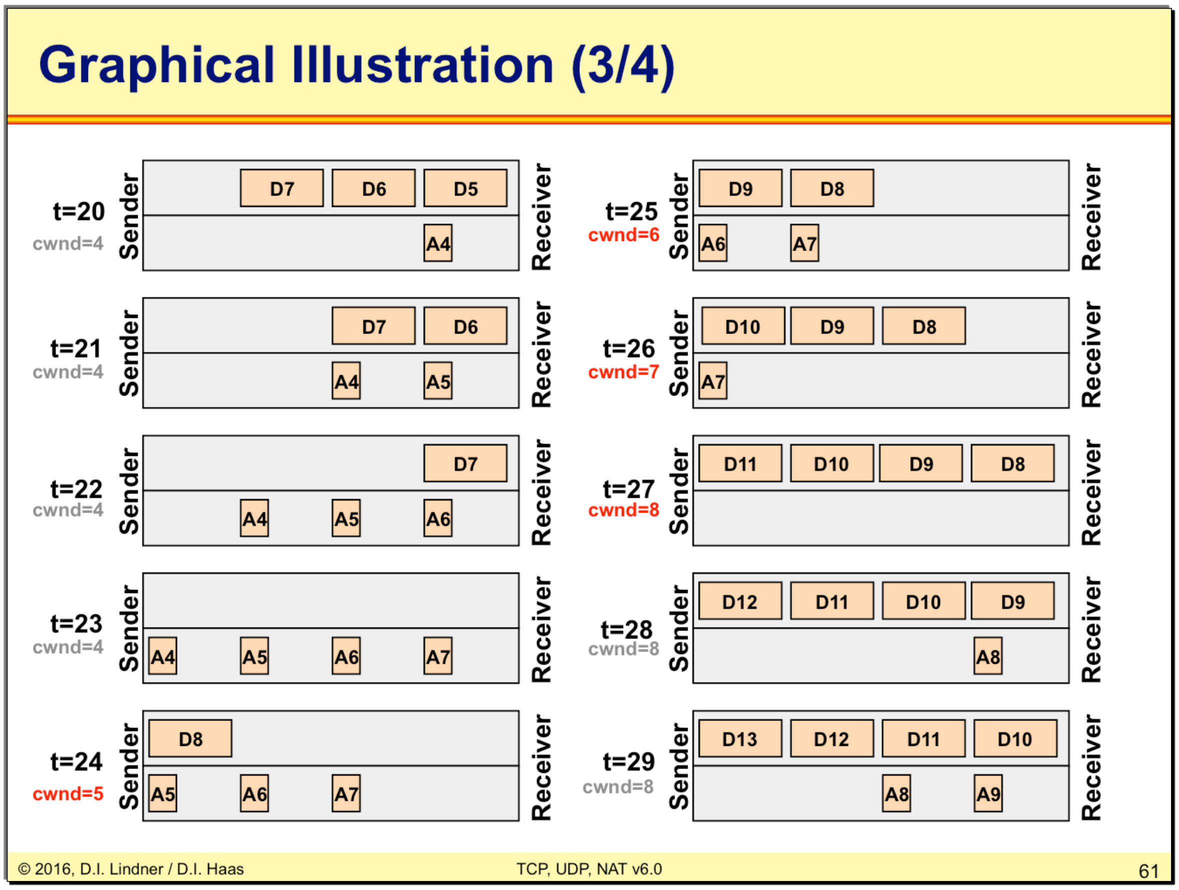


Graphical Illustration (2/4)

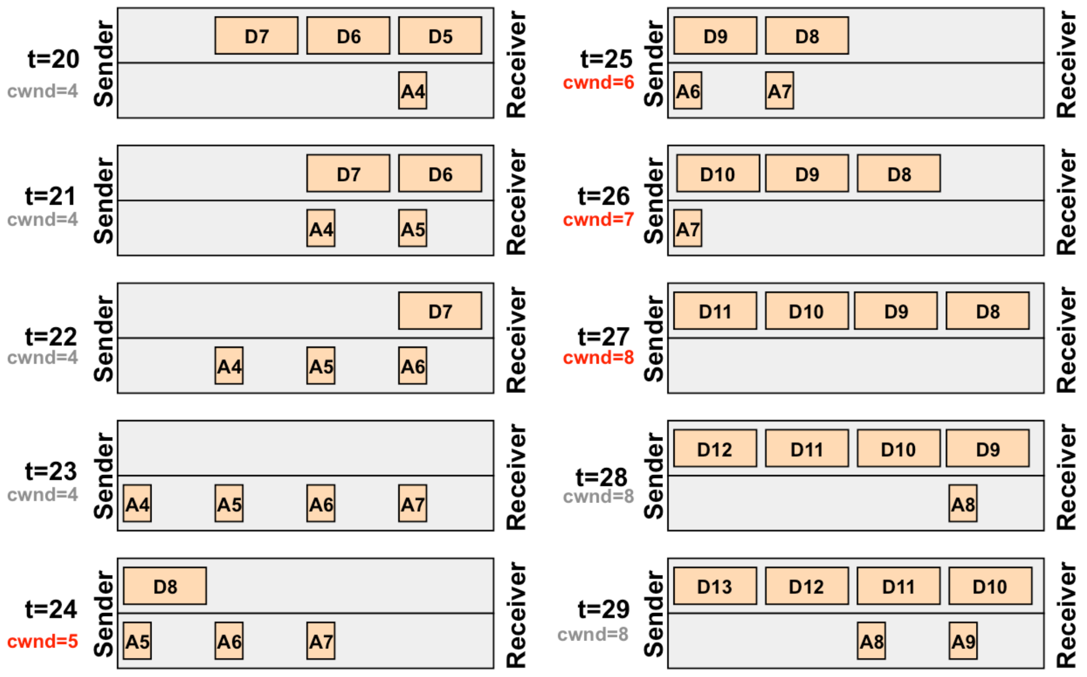


TCP, UDP, NAT v6.0 60

Observe the exponential growth of the data rate.

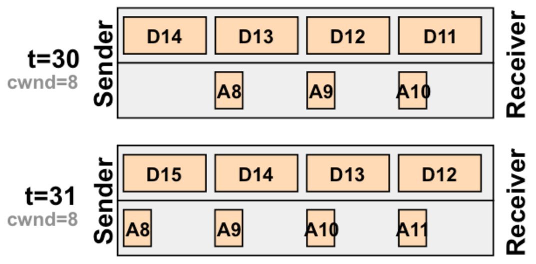


Graphical Illustration (3/4)



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 bandwidth-delay product (8 = RTT x BW)
  - In other words: the pipe can accept 8 segments 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.

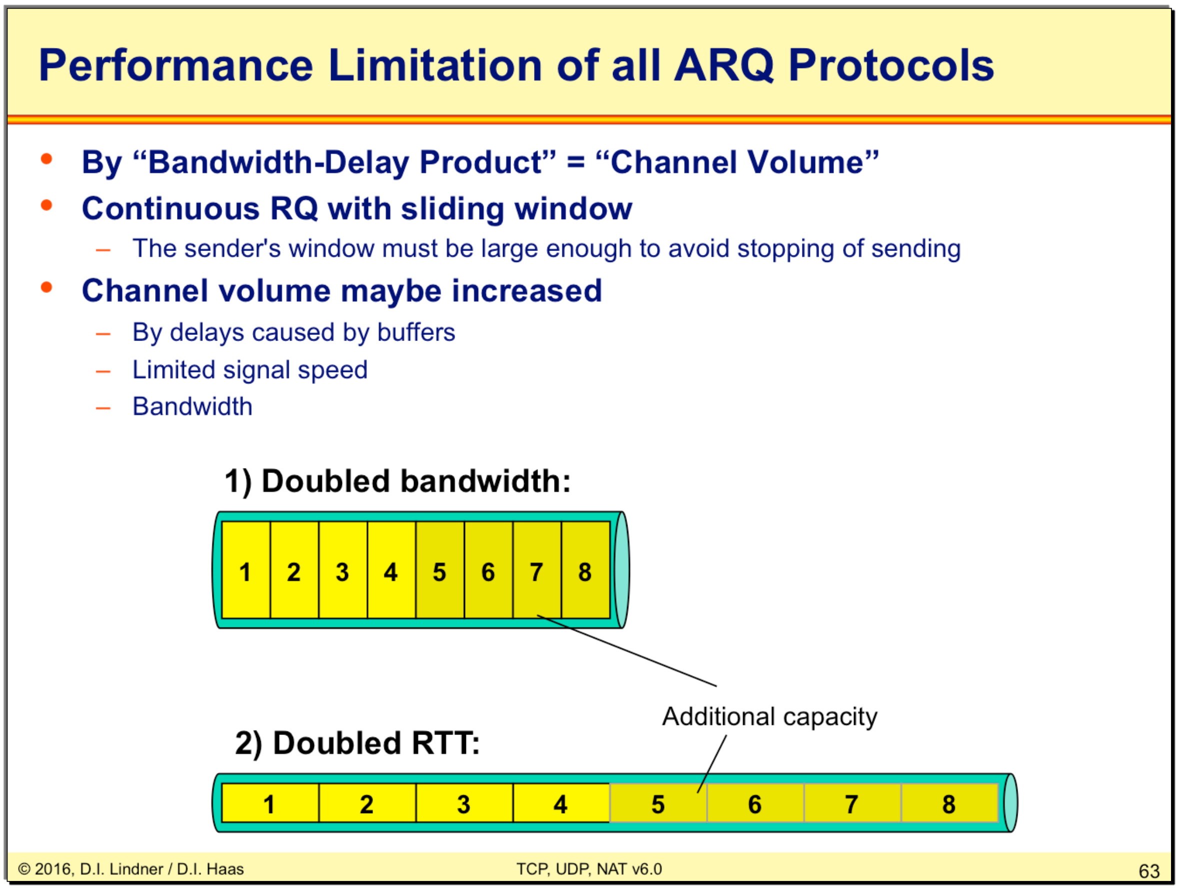
In our example cwnd=8 is the optimum, corresponding to 8 packets that can be sent before waiting for an acknowledgement. This optimum is expressed via the famous bandwidth-delay product, i. e.

pipe capacity = BW x RTT ,

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.)



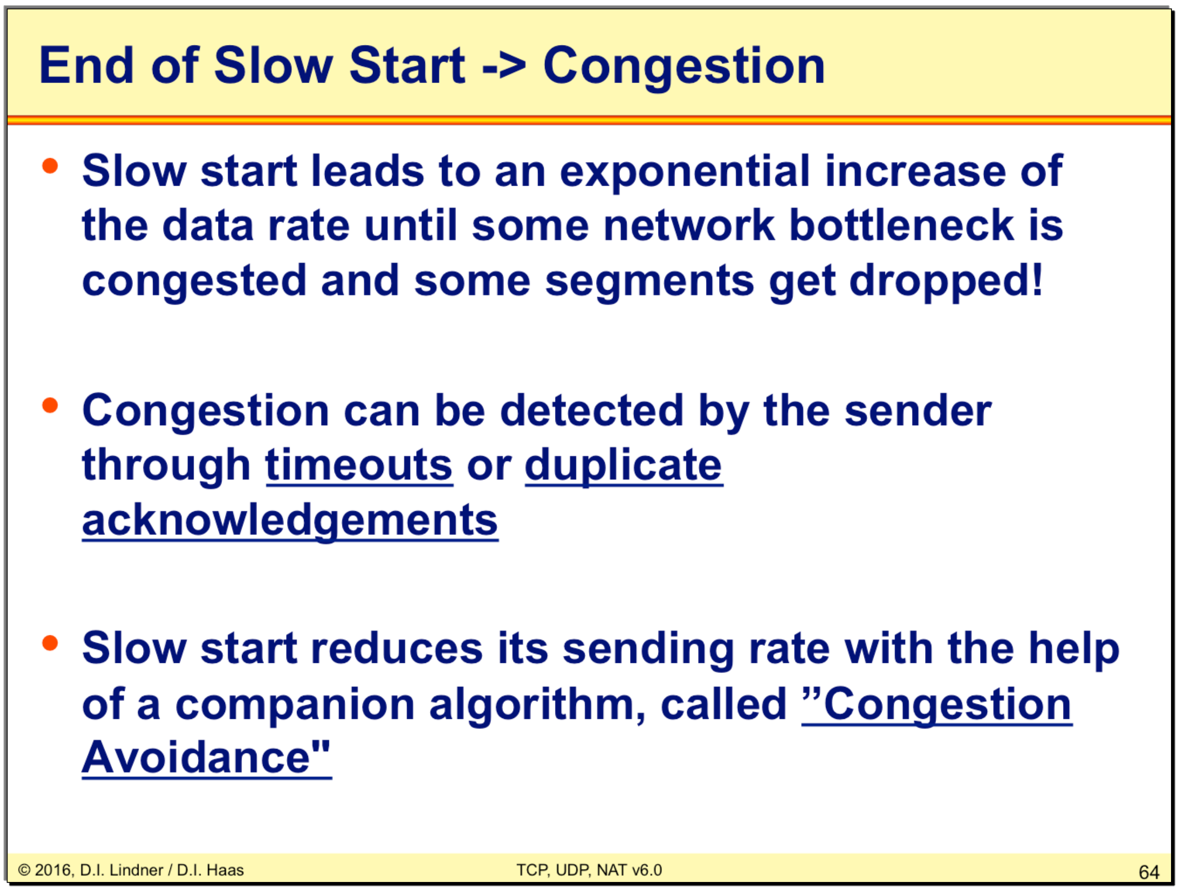
Performance Limitation of all ARQ Protocols

- By "Bandwidth-Delay Product" = "Channel Volume"
- Continuous RQ with sliding window
  - The sender's window must be large enough to avoid stopping of sending
- Channel volume maybe increased
  - By delays caused by buffers
  - Limited signal speed
  - Bandwidth

1) Doubled bandwidth:
1 2 3 4 5 6 7 8
Additional capacity
2) Doubled RTT:
1 2 3 4 5 6 7 8

Large enough means a value which covers the sum of serialization-, switching- and propagation-delays.

Note: window size maybe also be limited because of memory constraints (buffer) at the sender or receiver side



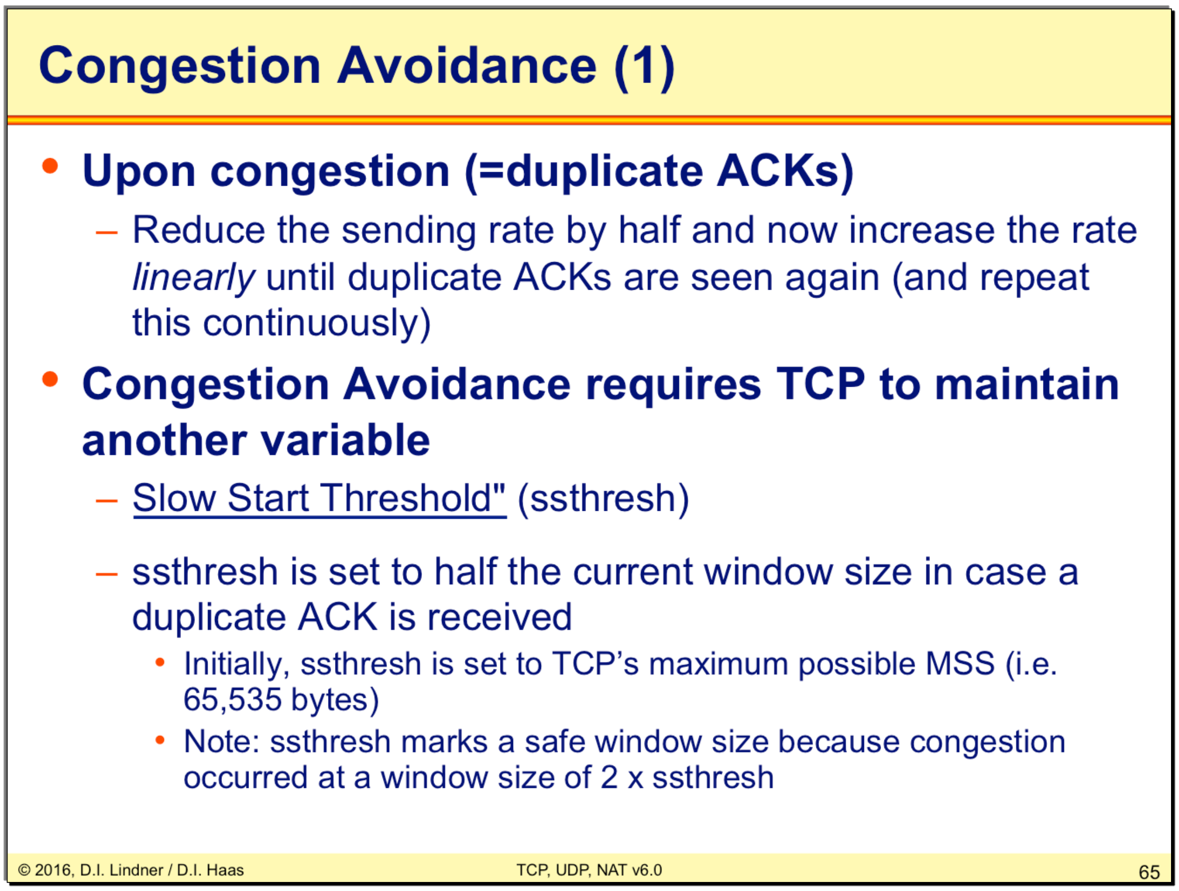
End of Slow Start -> Congestion

- Slow start leads to an exponential increase of the data rate until some network bottleneck is congested and some segments get dropped!
- Congestion can be detected by the sender through timeouts or duplicate acknowledgements
- Slow start reduces its sending rate with the help of a companion algorithm, called "Congestion Avoidance"

Timeout means heavy or high congestion -> all segments in a row were dropped in a tail-drop queue.

Duplicate ACK means, that still something is reaching the destination -> small or low congestion which causes maybe a single segment loss only.

Note this central TCP assumption: Segments are dropped because of buffer overflows and NOT because of bit errors! Therefore segment loss indicates congestion somewhere in the network.

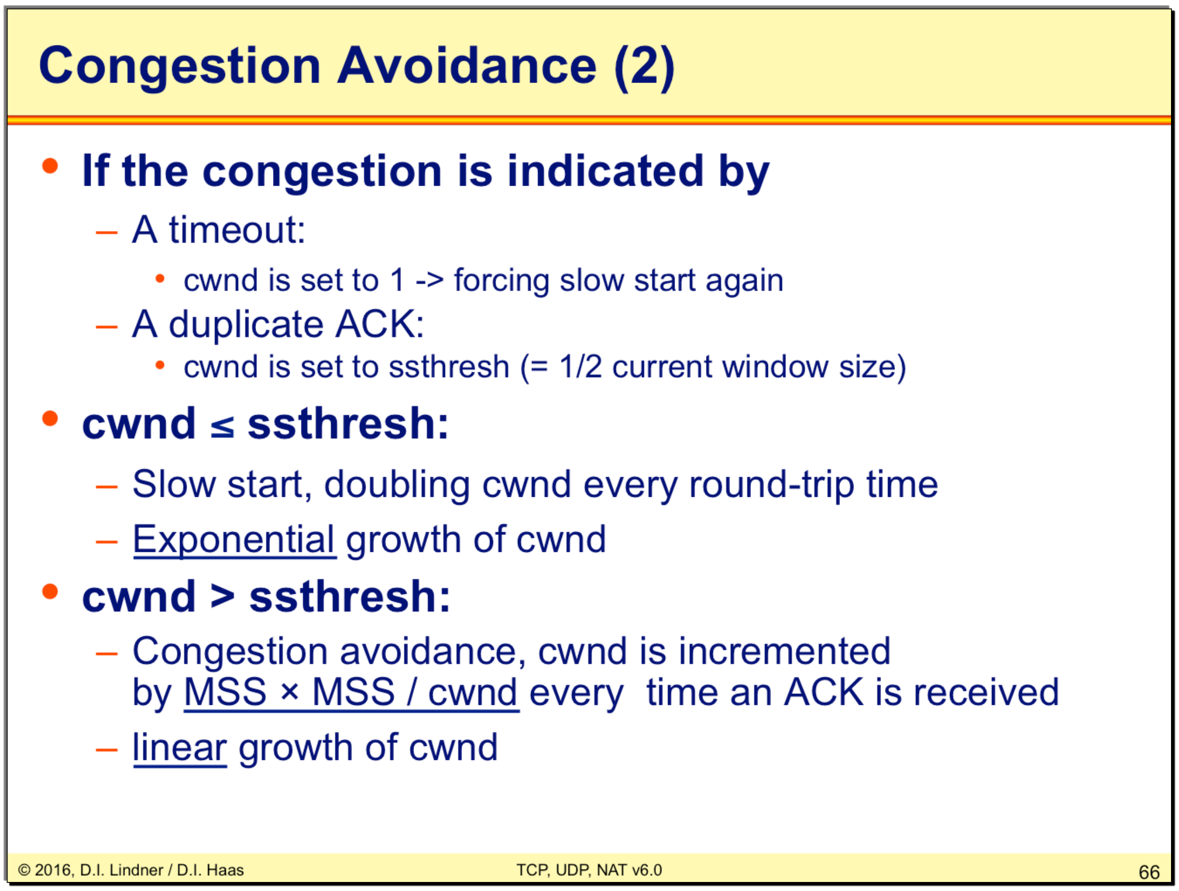


Congestion Avoidance (1)

- **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)
- **Congestion Avoidance requires TCP to maintain another variable**
  - Slow Start Threshold" (ssthresh)
  - ssthresh is set to half the current window size in case a duplicate ACK is received
    - Initially, ssthresh is set to TCP's maximum possible MSS (i.e. 65,535 bytes)
    - Note: ssthresh marks a safe window size because congestion occurred at a window size of 2 x ssthresh

TCP, UDP, NAT v6.0 65

Note: ssthresh marks a safe window size because congestion occurred at a window size of 2 x ssthresh.

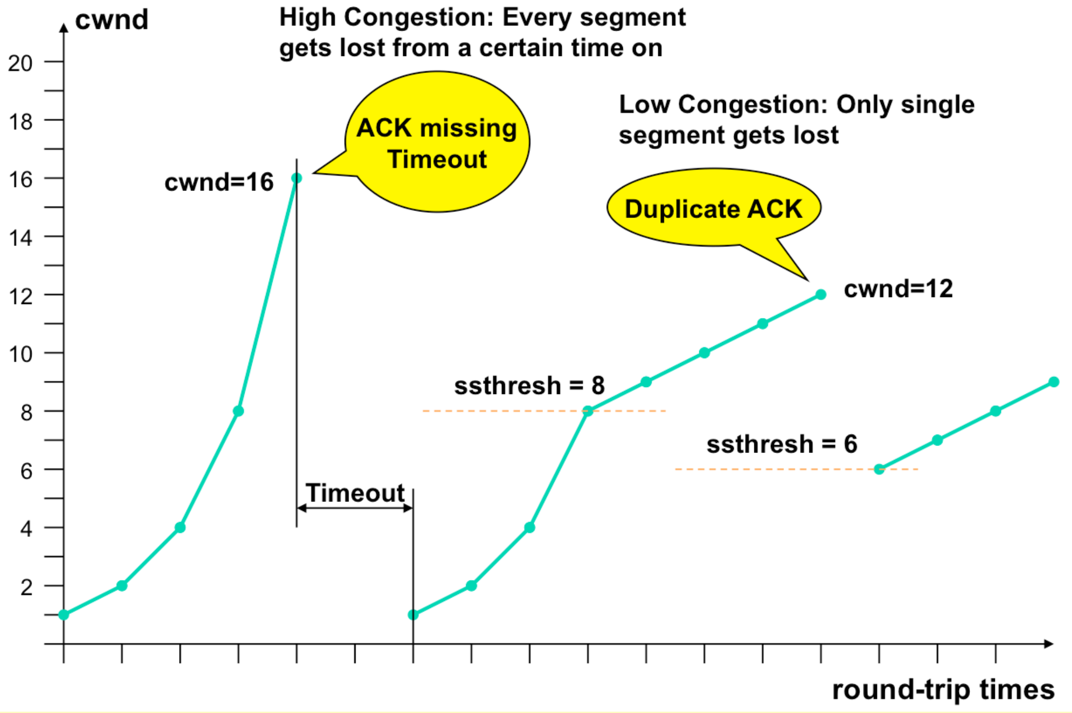


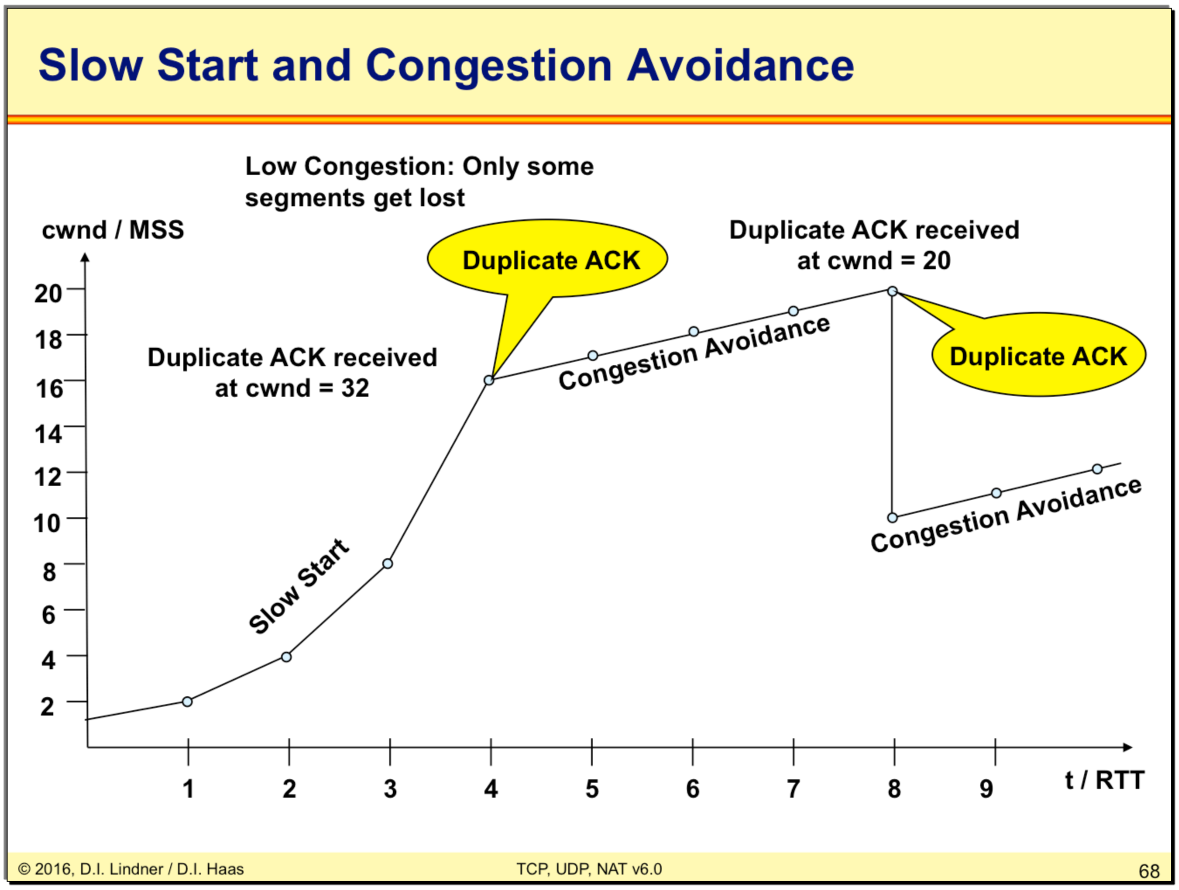
Congestion Avoidance (2)

- **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

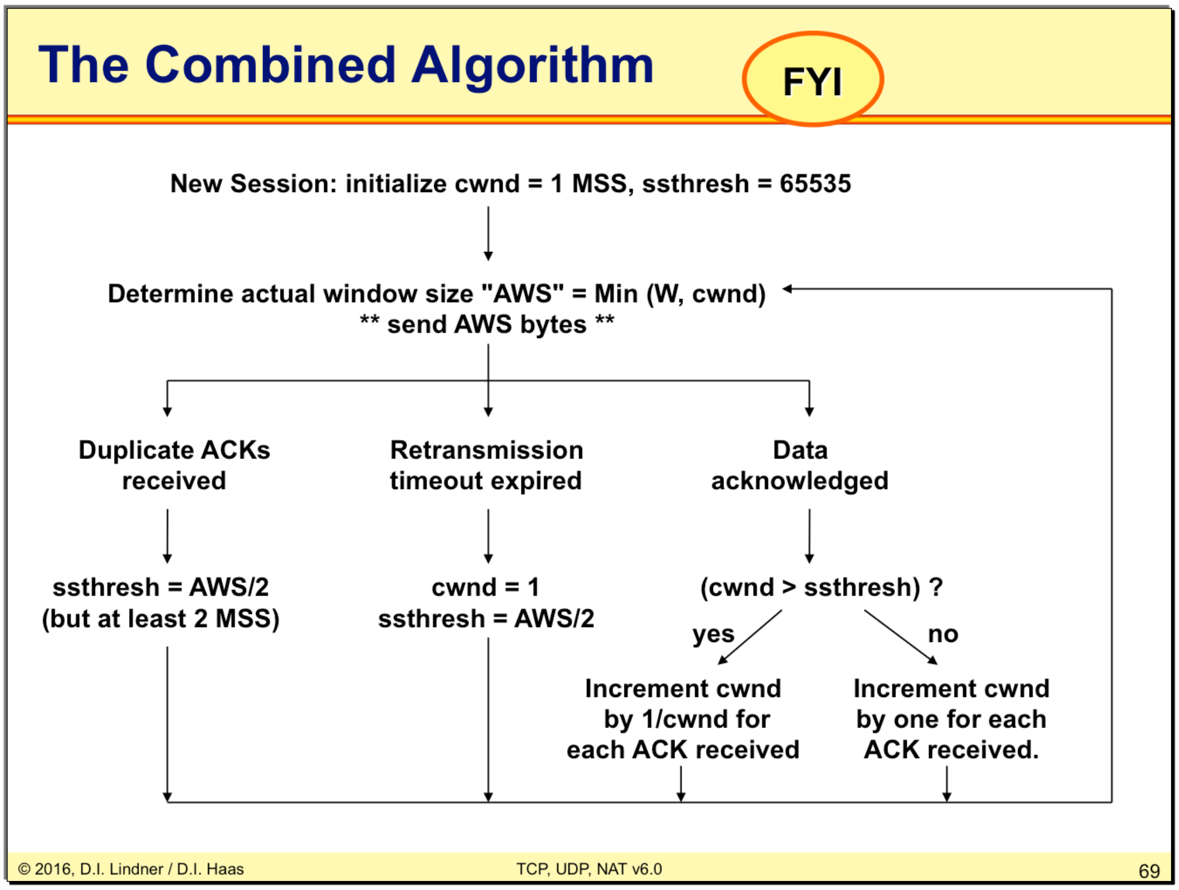
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 66

Slow Start and Congestion Avoidance





Slow Start and Congestion Avoidance



The Combined Algorithm

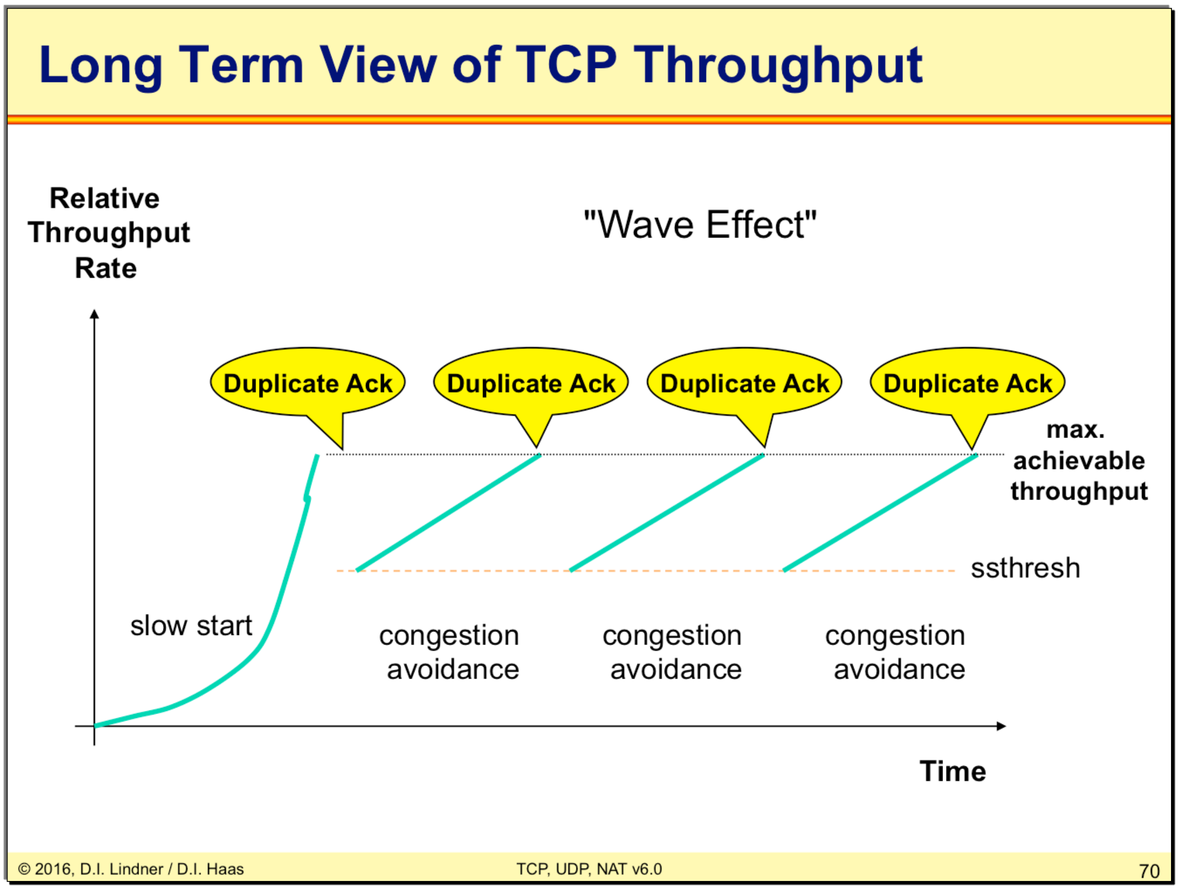
FYI

New Session: initialize cwnd = 1 MSS, ssthresh = 65535
Determine actual window size "AWS" = Min (W, cwnd)
** send AWS bytes **
Duplicate ACKs received
Retransmission timeout expired
Data acknowledged
ssthresh = AWS/2 (but at least 2 MSS)
cwnd = 1 ssthresh = AWS/2
(cwnd > ssthresh) ?
yes
no
Increment cwnd by 1/cwnd for each ACK received
Increment cwnd by one for each ACK received.

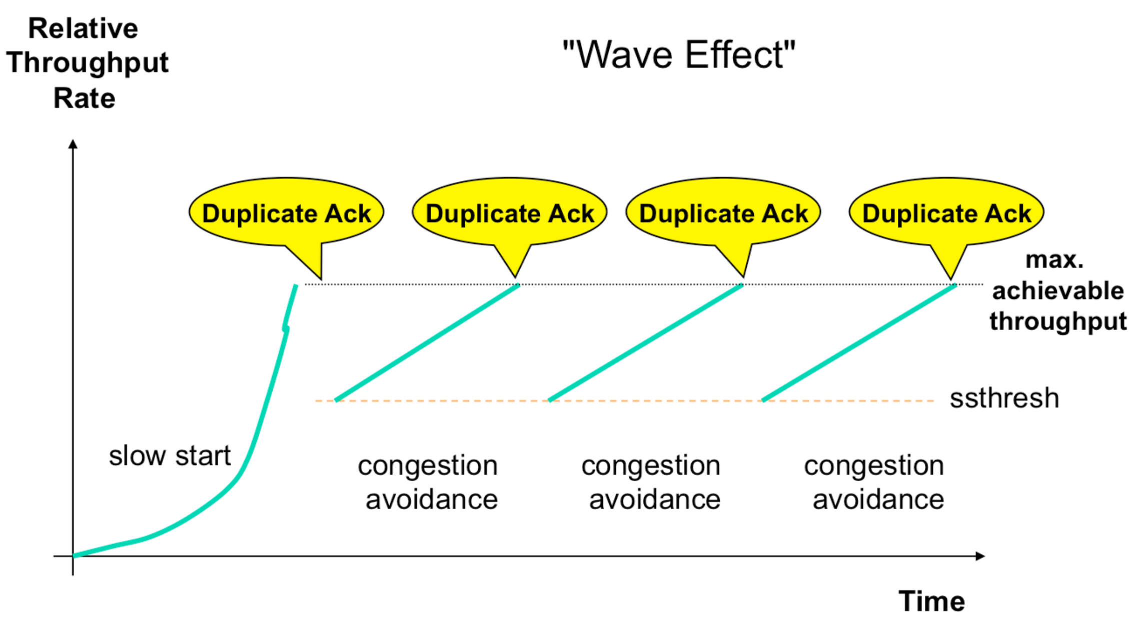
TCP, UDP, NAT v6.0 69

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 Recovery—see next slides.



Long Term View of TCP Throughput

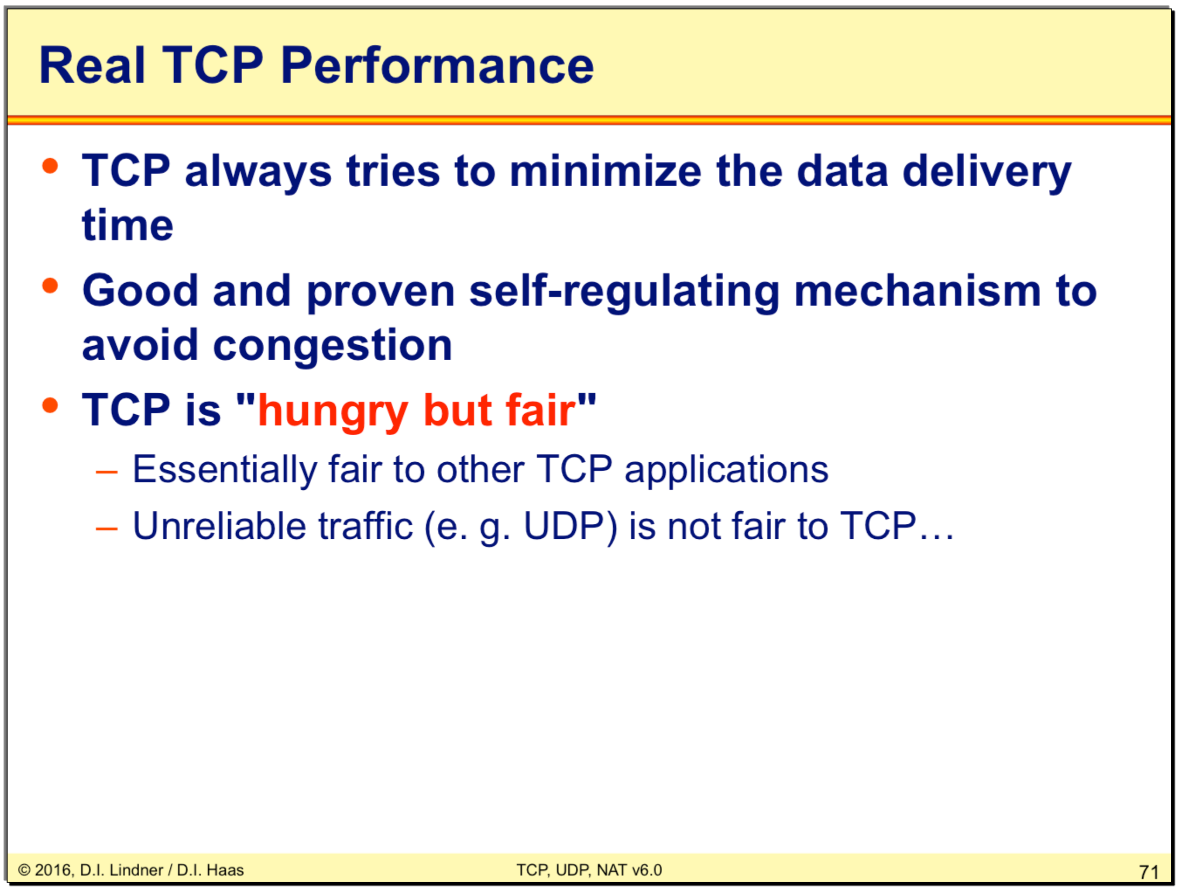


© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 70

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.

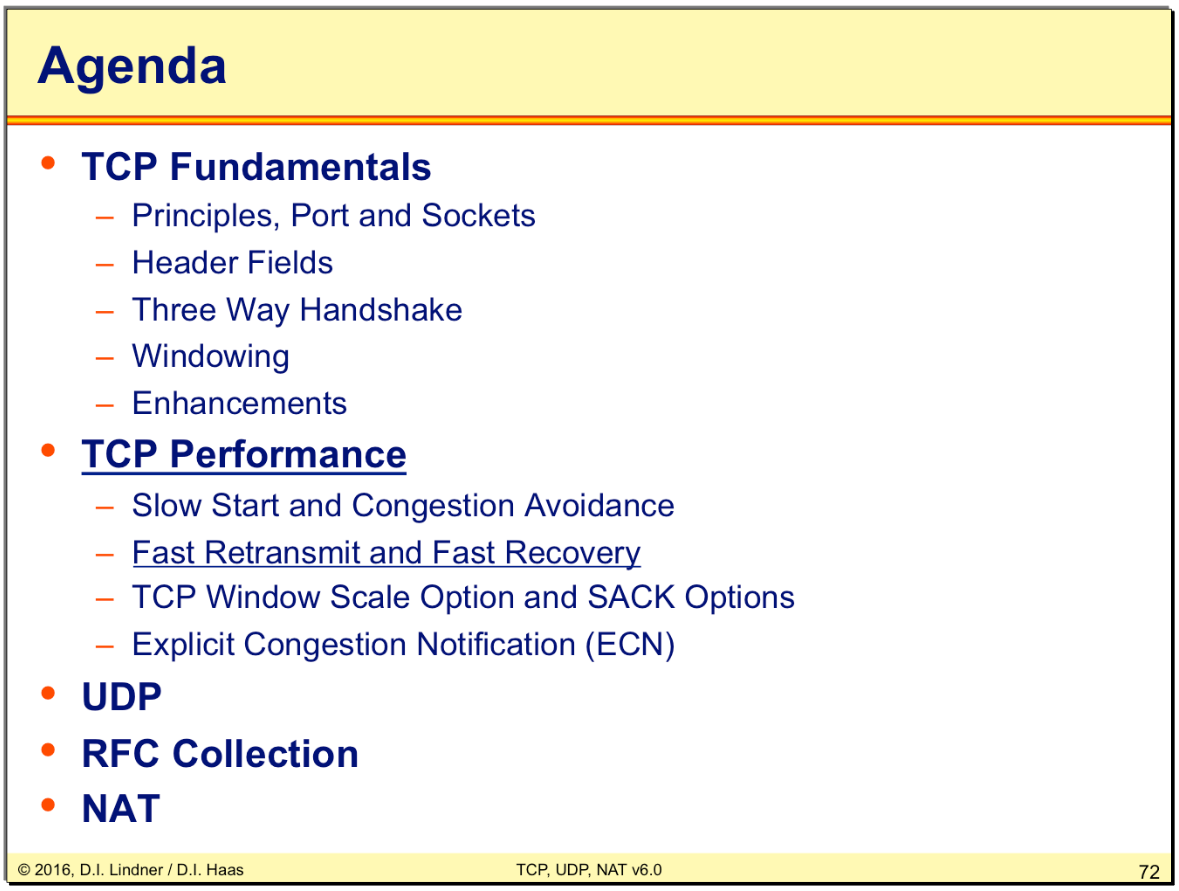


Real TCP Performance

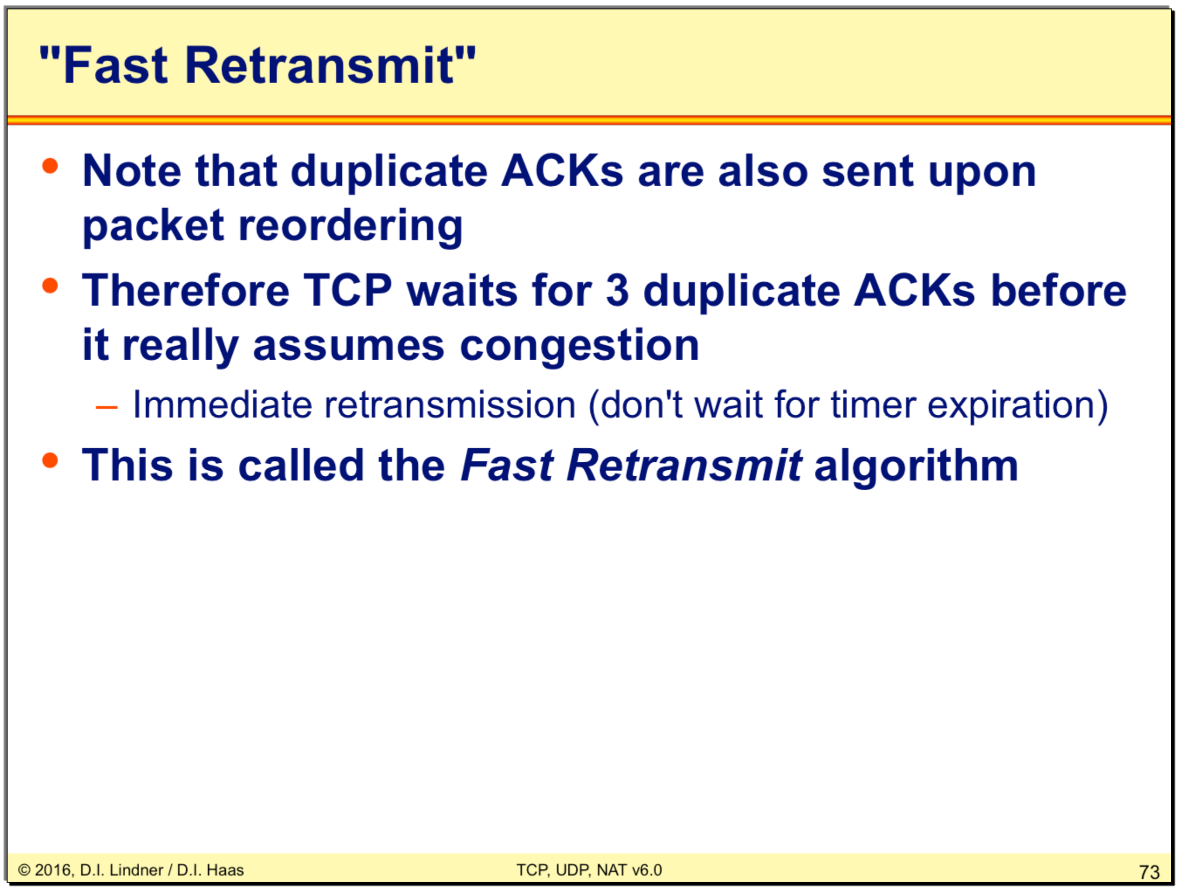
- **TCP always tries to minimize the data delivery time**
- **Good and proven self-regulating mechanism to avoid congestion**
- **TCP is "hungry but fair"**
  - Essentially fair to other TCP applications
  - Unreliable traffic (e. g. UDP) is not fair to TCP…

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.



Agenda

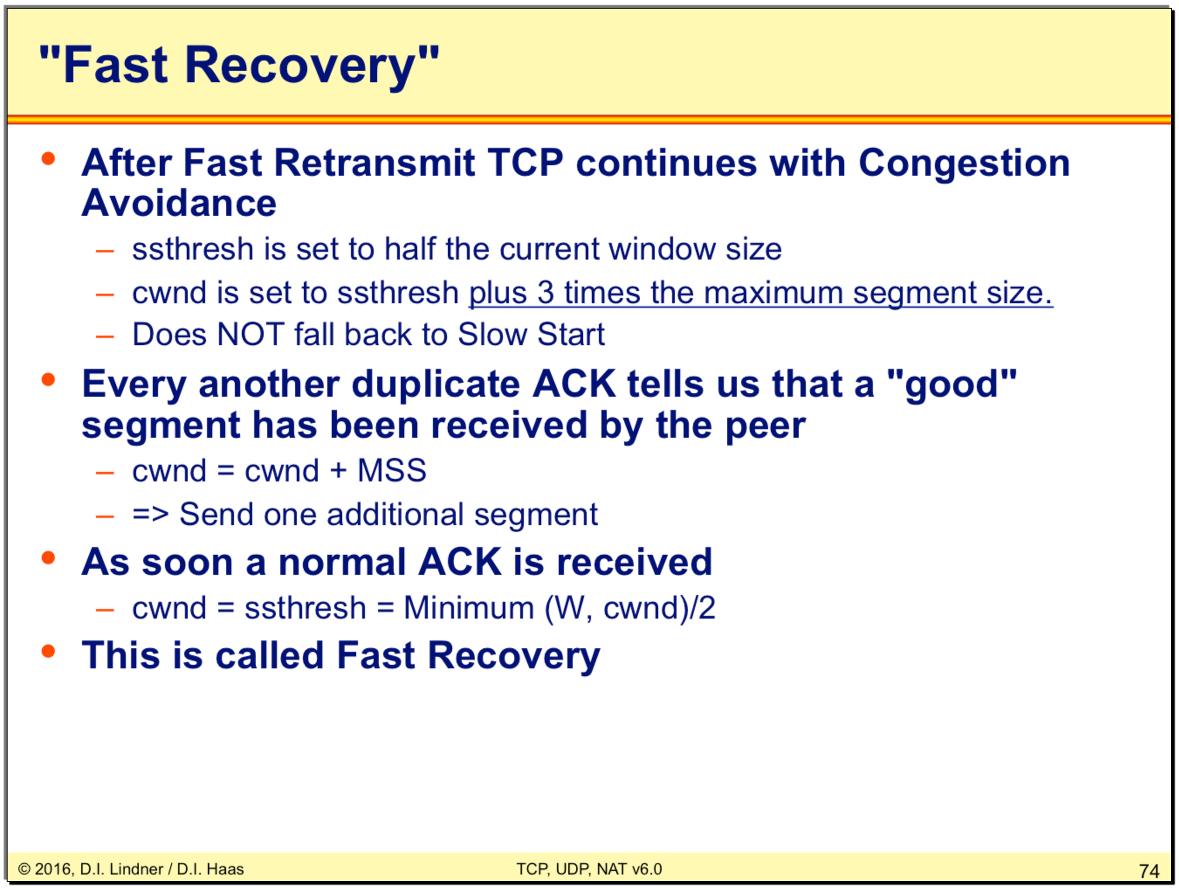


"Fast Retransmit"

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

Fast Retransmit requires a receiver to send an immediate duplicate acknowledgement in order to notify the sender which segments are (still) expected by the receiver.

But when should retransmission occur? The receiver will also send duplicate acknowledgements when segments are arriving in the wrong order typically caused by a rerouting event in the network. Observations have shown that reordering in such a case causes one or two duplicate Acks on the average and only if three or more duplicate acks are seen then this is a strong indication for a lost segment. In such a case Fast Retransmission is done, i. e. TCP does not wait until segment's retransmission timer expires.



"Fast Recovery"

- **After Fast Retransmit TCP continues with Congestion Avoidance**
  - ssthresh is set to half the current window size
  - cwnd is set to ssthresh <u>plus 3 times the maximum segment size.</u>
  - Does NOT fall back to Slow Start
- **Every another duplicate ACK tells us that a "good" segment has been received by the peer**
  - cwnd = cwnd + MSS
  - => Send one additional segment
- **As soon a normal ACK is received**
  - cwnd = ssthresh = Minimum (W, cwnd)/2
- **This is called Fast Recovery**

Why cwnd= ssthresh/s + 3 x MSS?

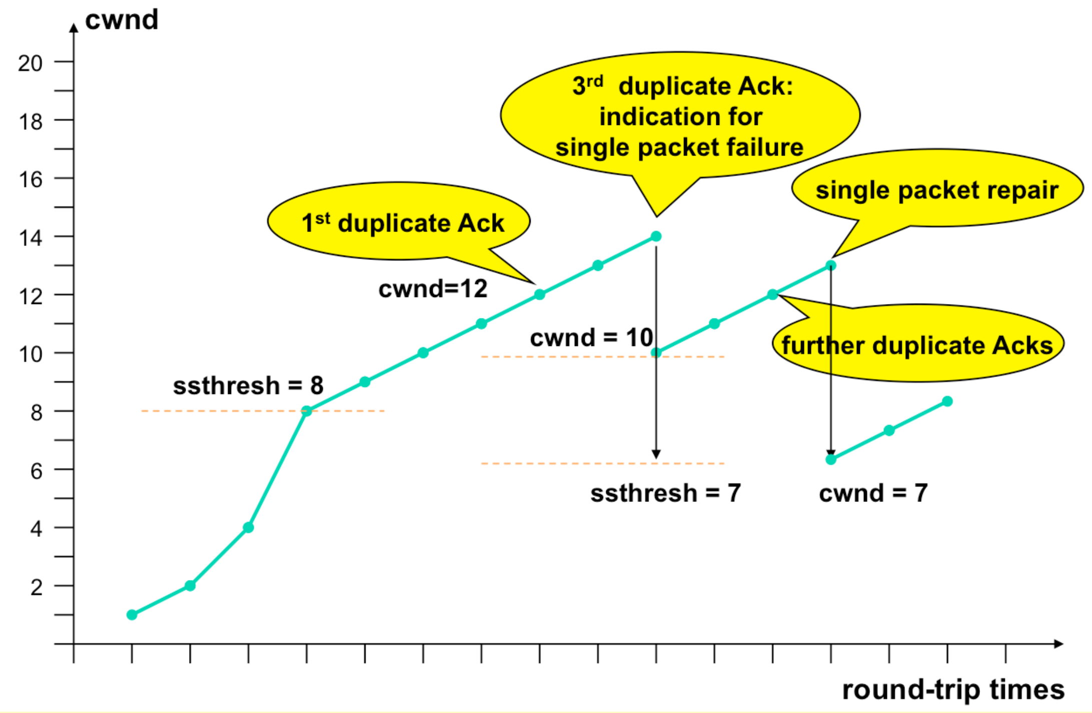
Remember: Fast Retransmit waits for 3 duplicate ACKs; from this can be concluded that the receiver must have received 3 segments already.

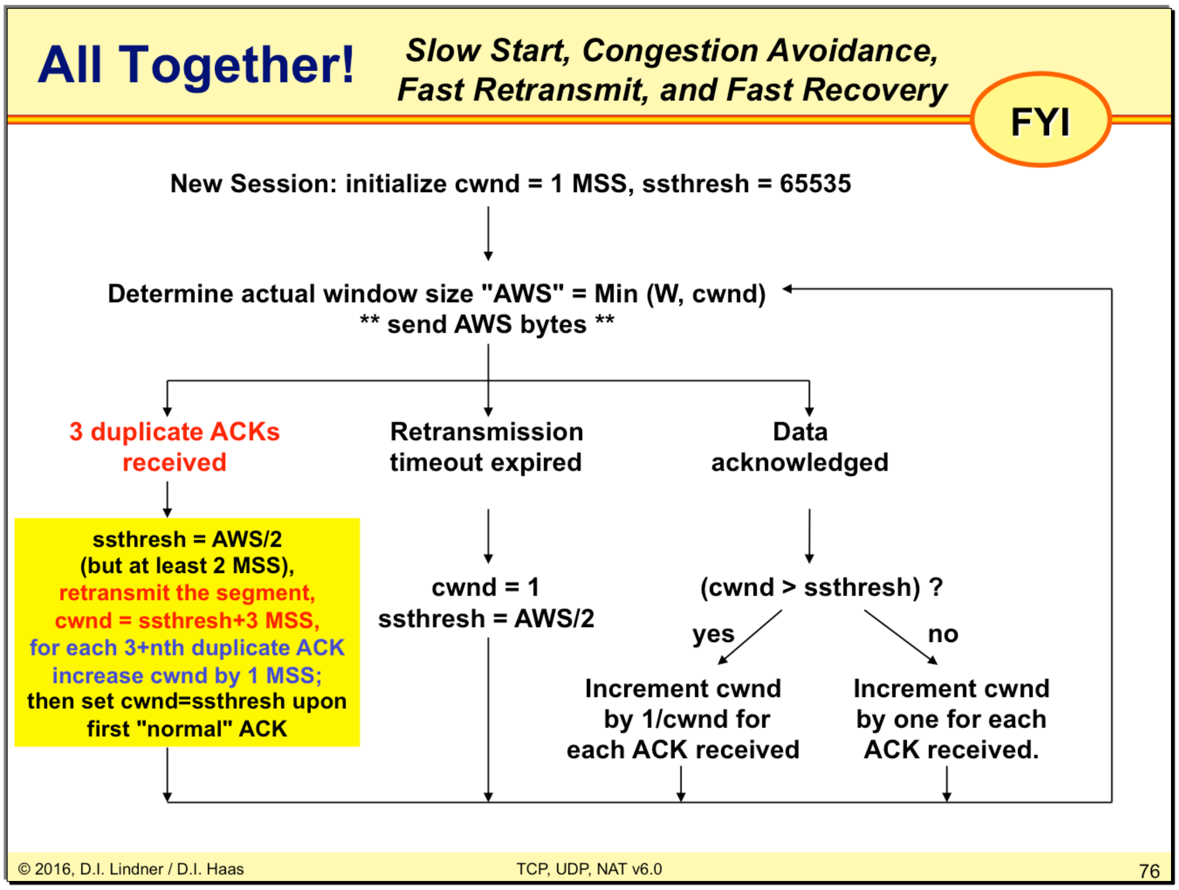
Hence Congestion avoidance, but not slow start should be performed. The receiver could only generate a duplicate ACK when another segment is received. That is there are <u>still segments flowing</u> through the network! Slow start would reduce this flow abruptly!

After that for each additional duplicate ACK the sender increases cwnd by 1 segment size. Upon receiving a normal ACK cwnd is set to ssthresh and sender resumes normal congestion avoidance mode.

Fast Recovery allows the sender to maintain the ack-clocked data rate for new data while the single segment loss repair is being undertaken. Note: if send window would be closed more abruptly the synchronization via duplicate ACKs would be lost. Still the single segment loss indicates congestion and back off to normal congestion avoidance mode must be done after that repair.

Fast Retransmit and Fast Recovery





All Together! *Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery*

FYI

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

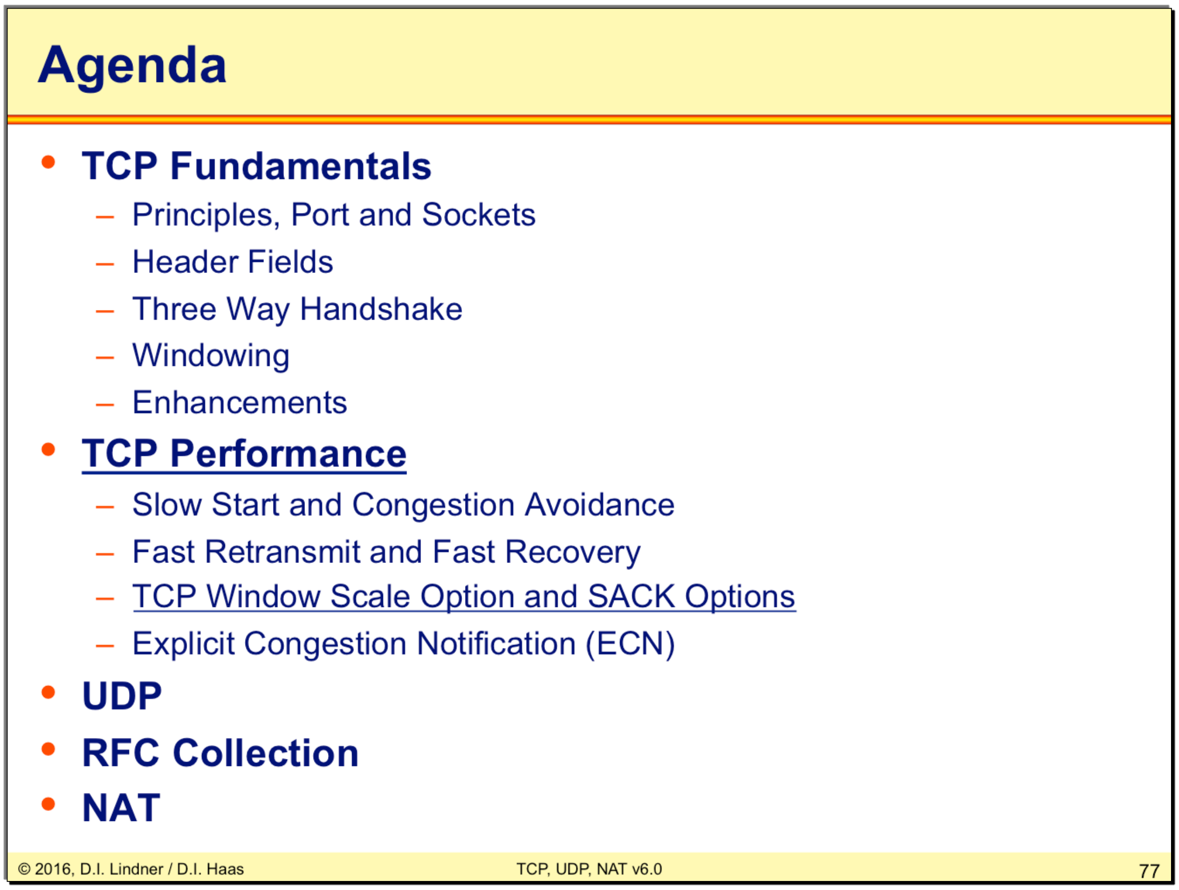
Determine actual window size "AWS" = Min (W, cwnd)
** send AWS bytes **

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

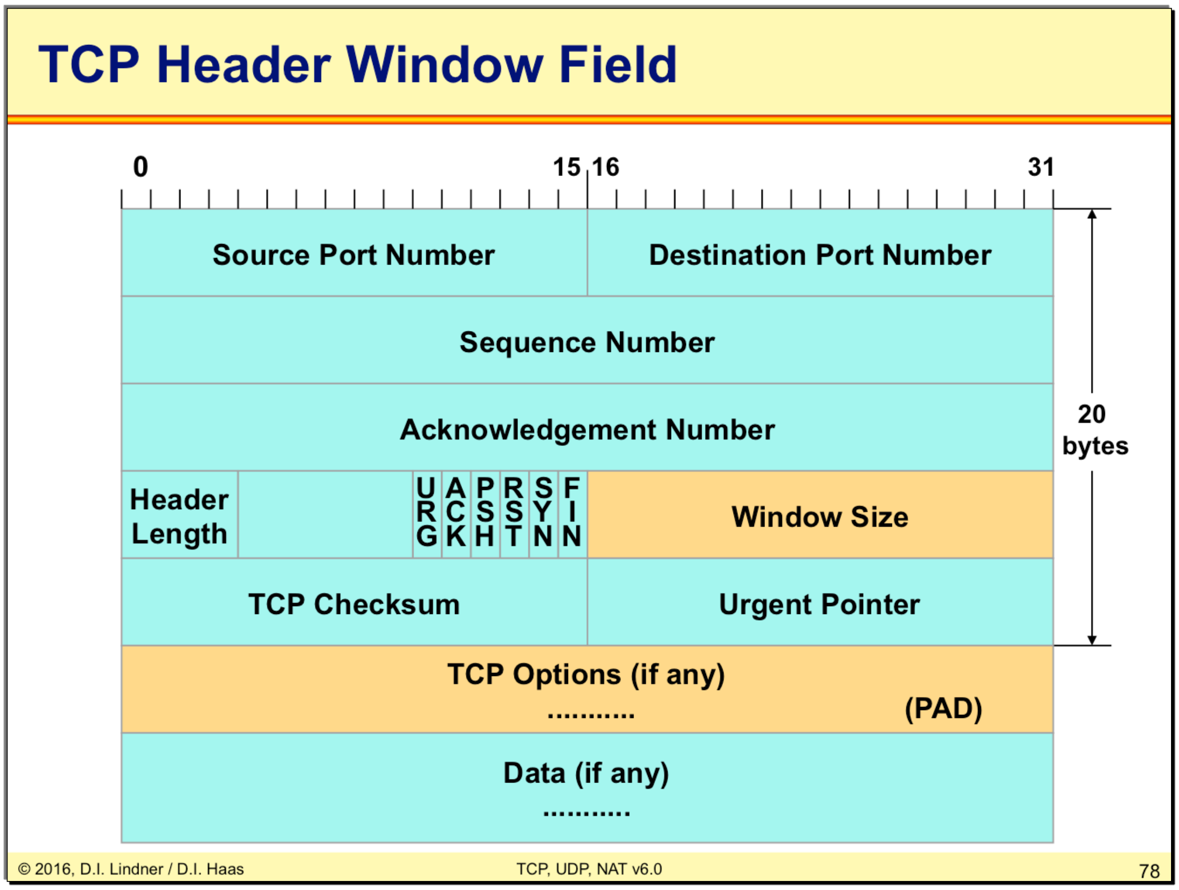
Retransmission timeout expired → cwnd = 1, ssthresh = AWS/2

Data acknowledged → (cwnd > ssthresh) ?
- yes → Increment cwnd by 1/cwnd for each ACK received
- no → Increment cwnd by one for each ACK received.

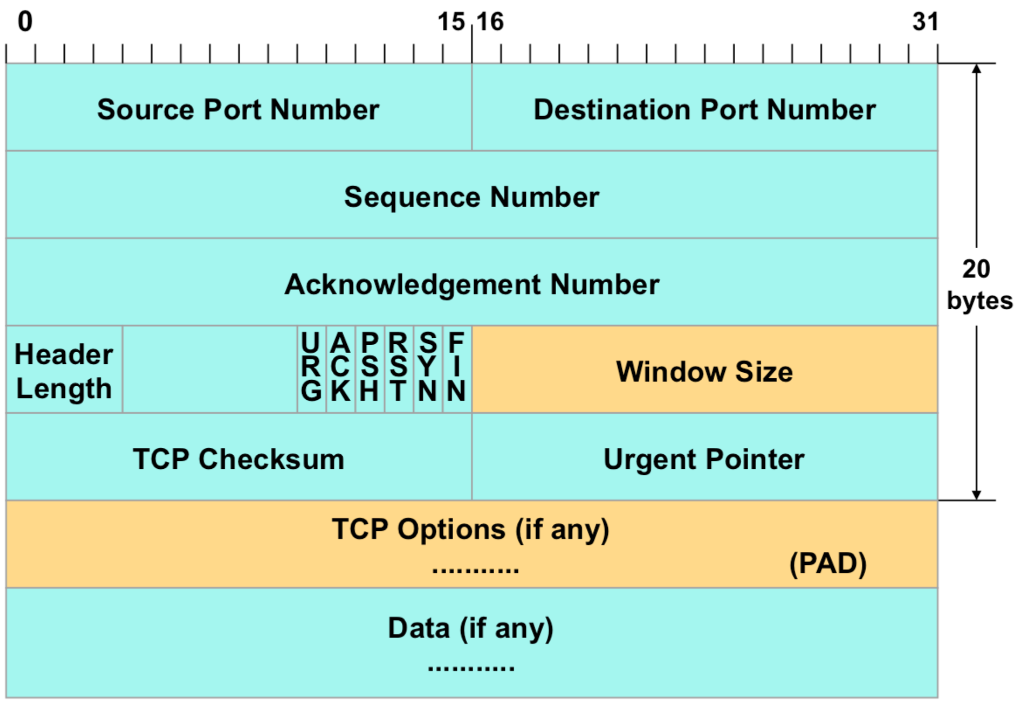
When one or two duplicate ACKs are received, TCP does not react because packet reorder is probable. Upon the third duplicate ACK TCP assumes that the segment (for which the duplicate ACK is meant) is really lost. TCP now immediately retransmit the packet (i. e. it does not wait for any timer expiration), 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 continuous sending new segments—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).



Agenda



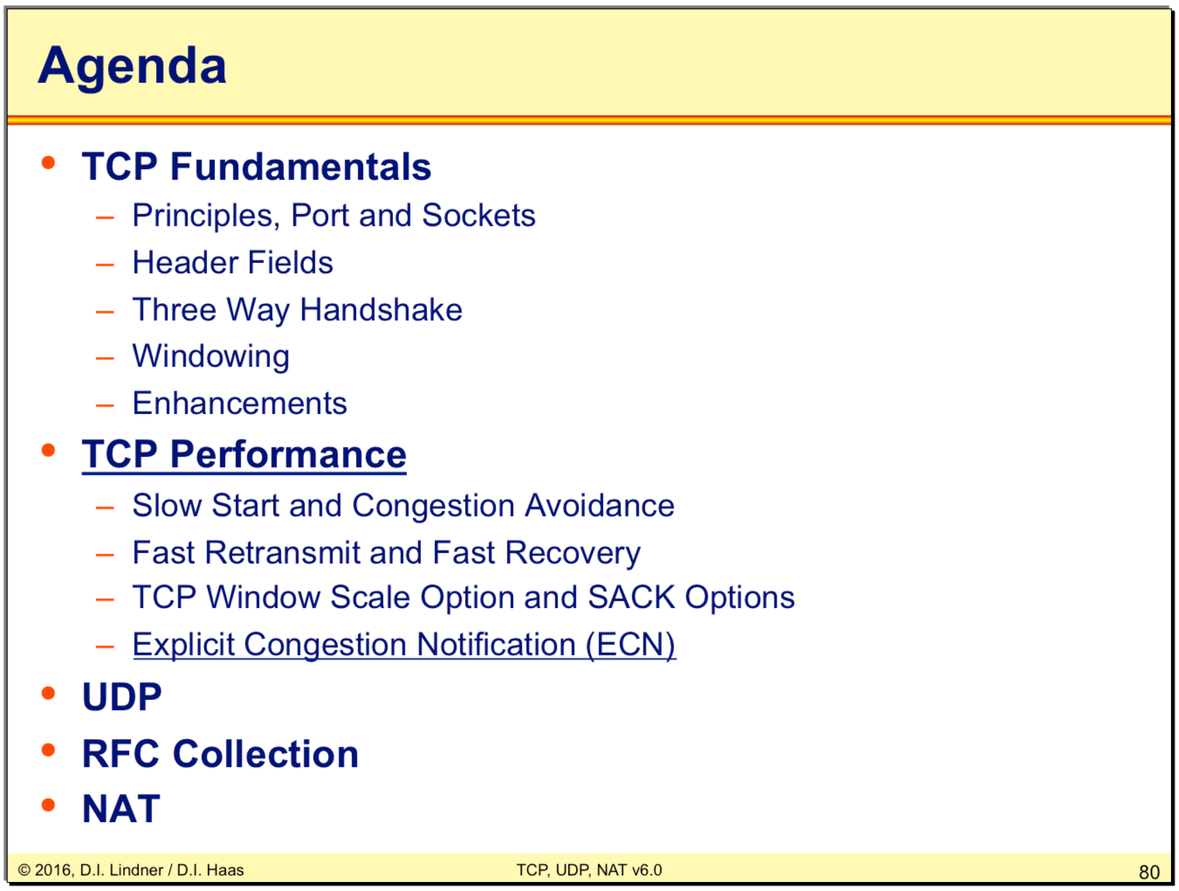
TCP Header Window Field



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

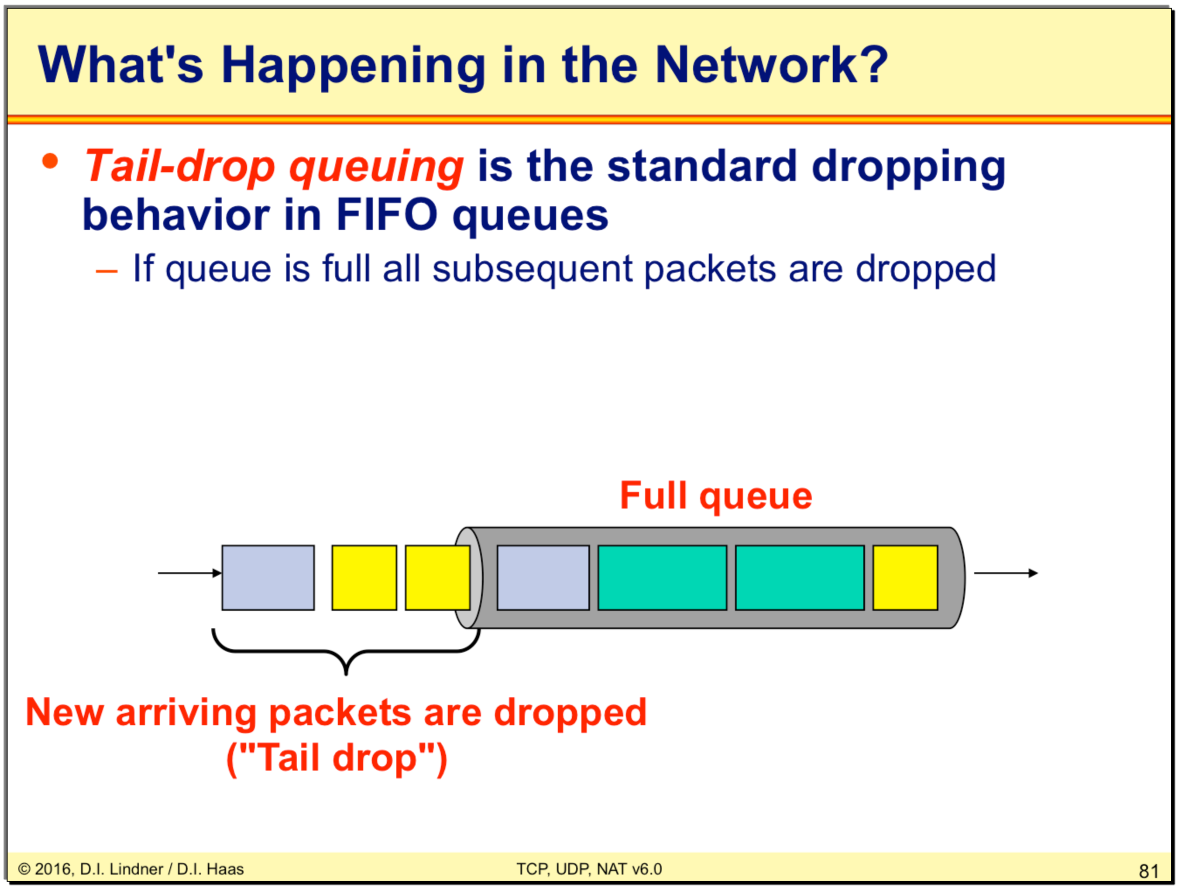
- **SACK (Selective Acknowledgement)**
  - if the SACK-permitted option is set during connection establishment, the receiver may selectively acknowledge already received data even if there is a gap in the TCP stream (Ack-based synchronization maintained)



Agenda

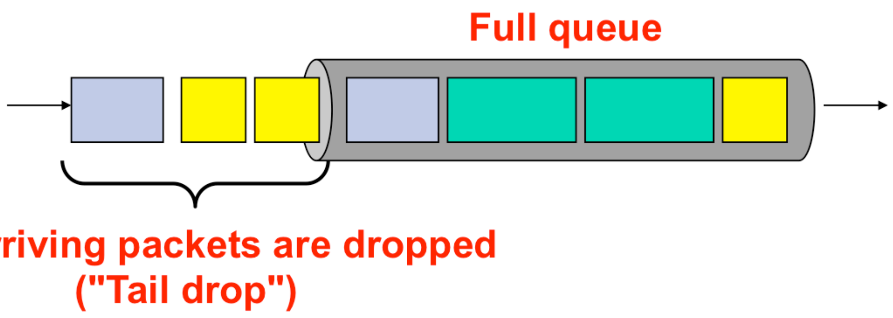
- **TCP Fundamentals**
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **UDP**
- **RFC Collection**
- **NAT**

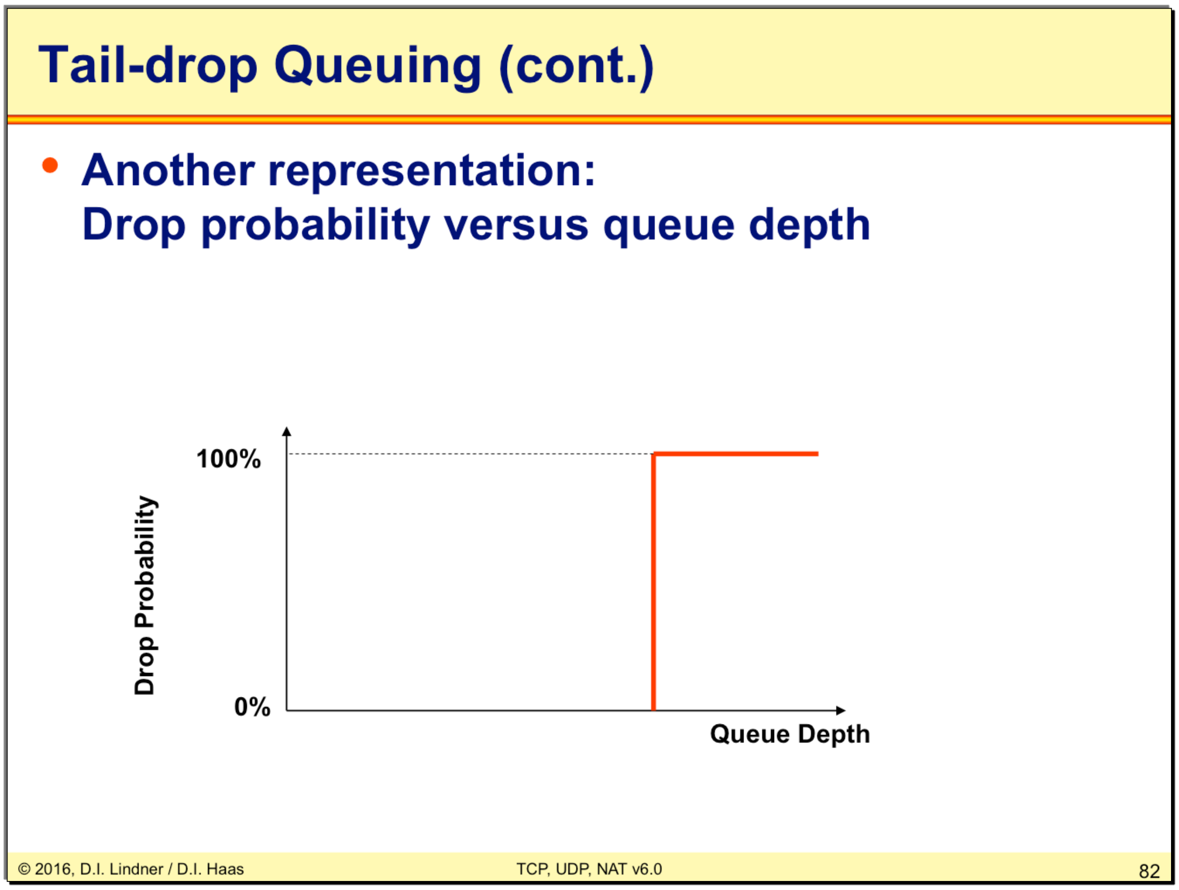
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 80



What's Happening in the Network?

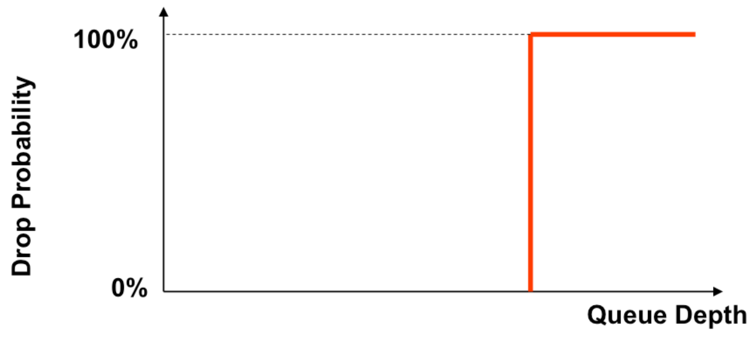
- ***Tail-drop queuing*** **is the standard dropping behavior in FIFO queues**
  - If queue is full all subsequent packets are dropped



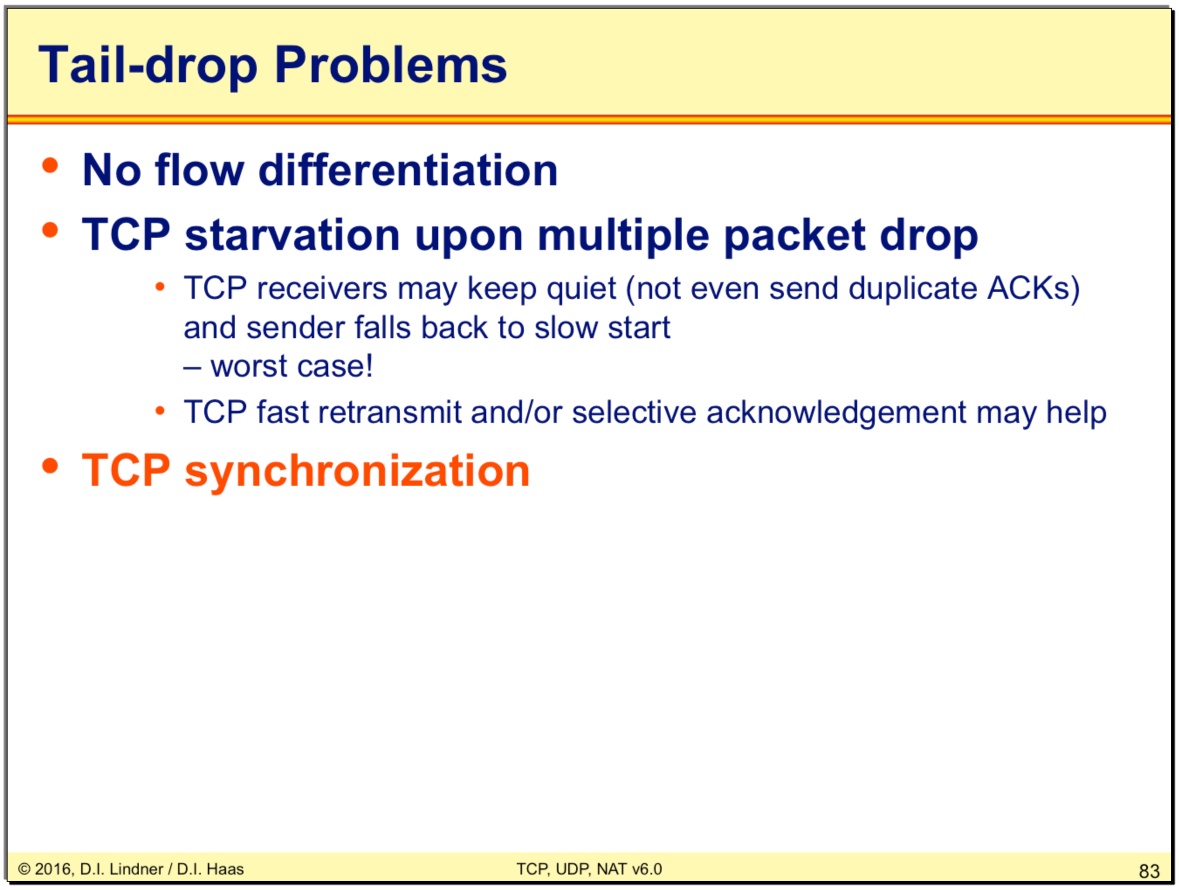


Tail-drop Queuing (cont.)

- 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.)

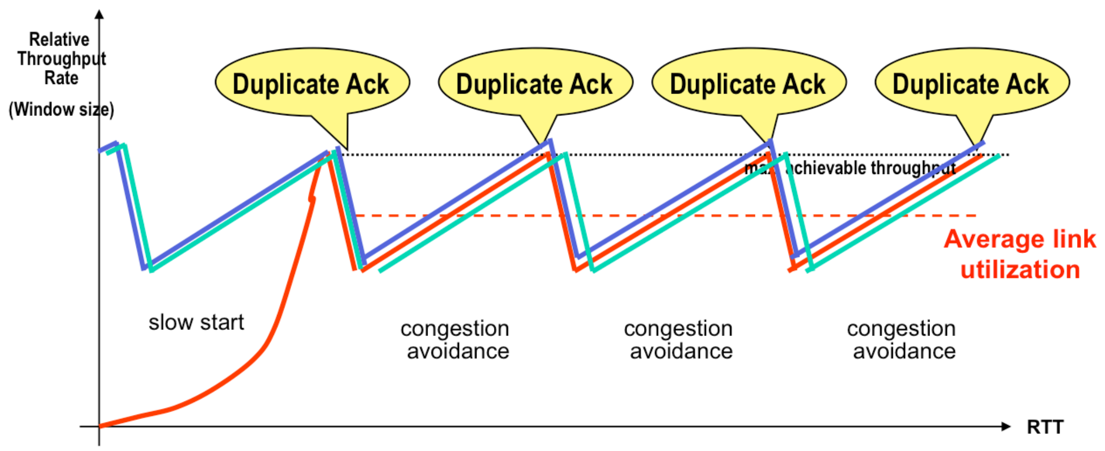


Tail-drop Problems

- 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

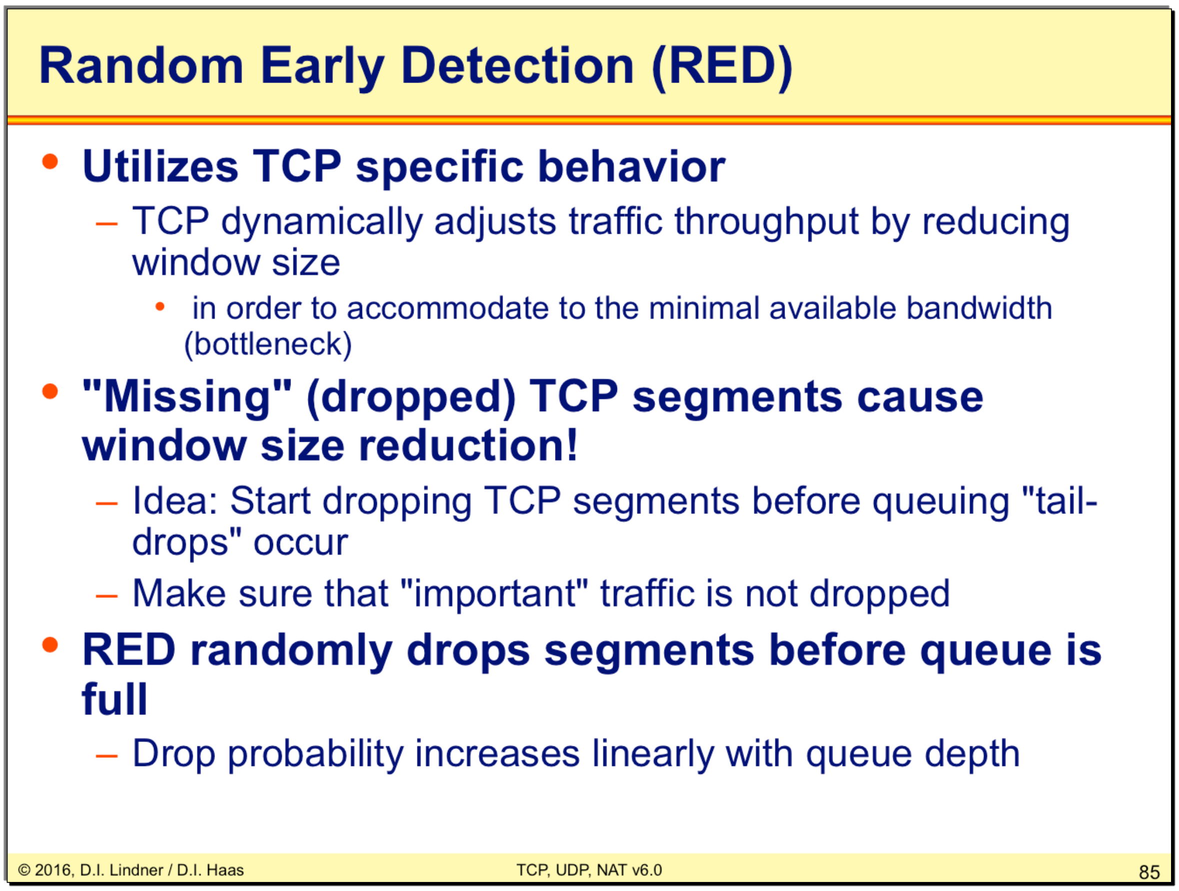
TCP Synchronization

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



TCP, UDP, NAT v6.0 84

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 Detection (RED)

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

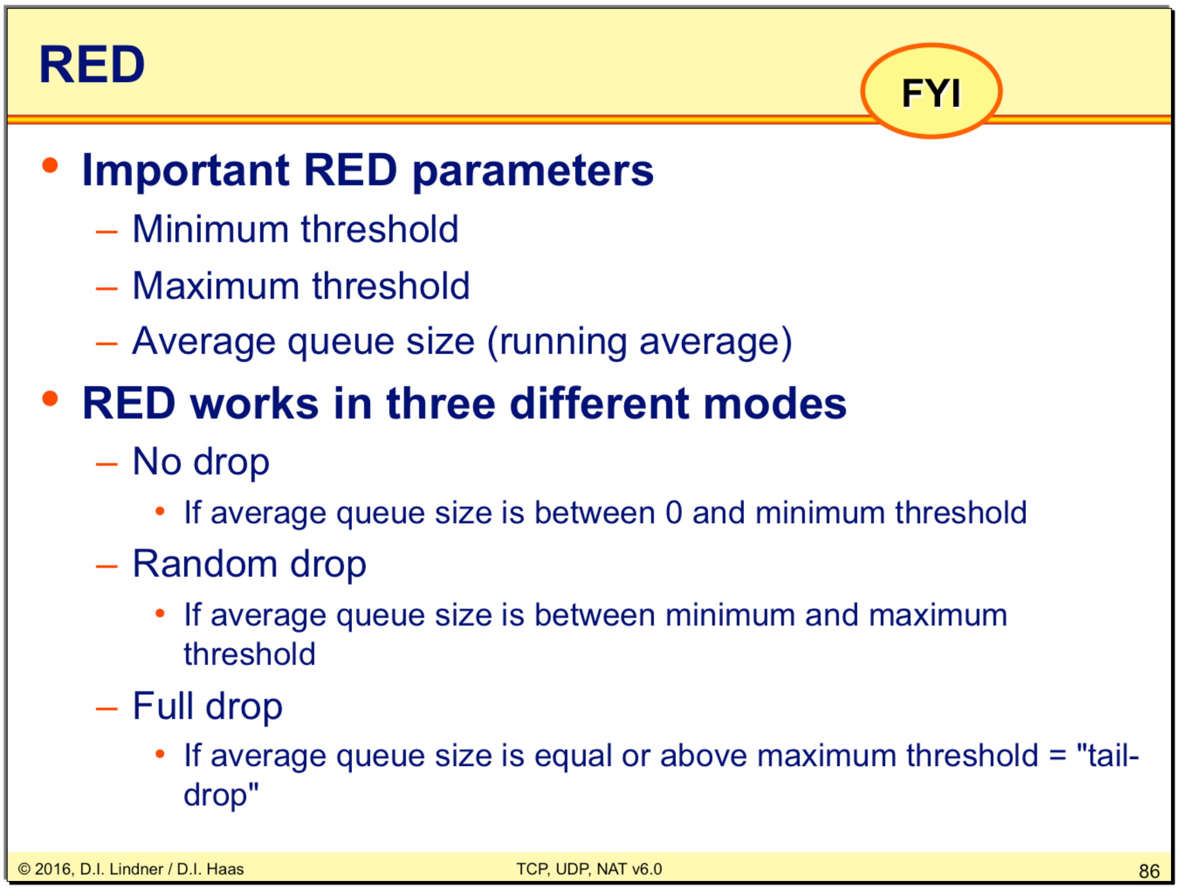
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 85

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.

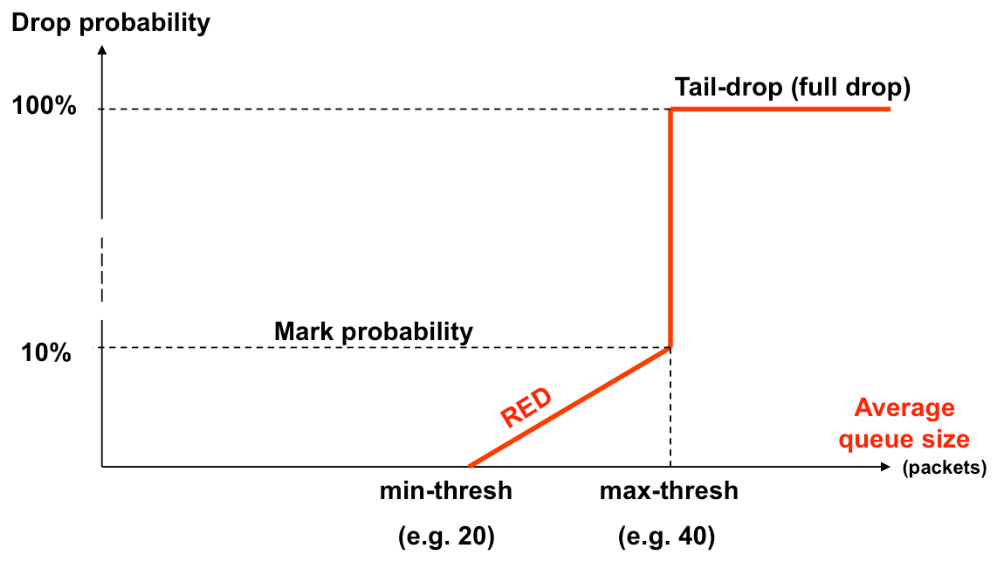


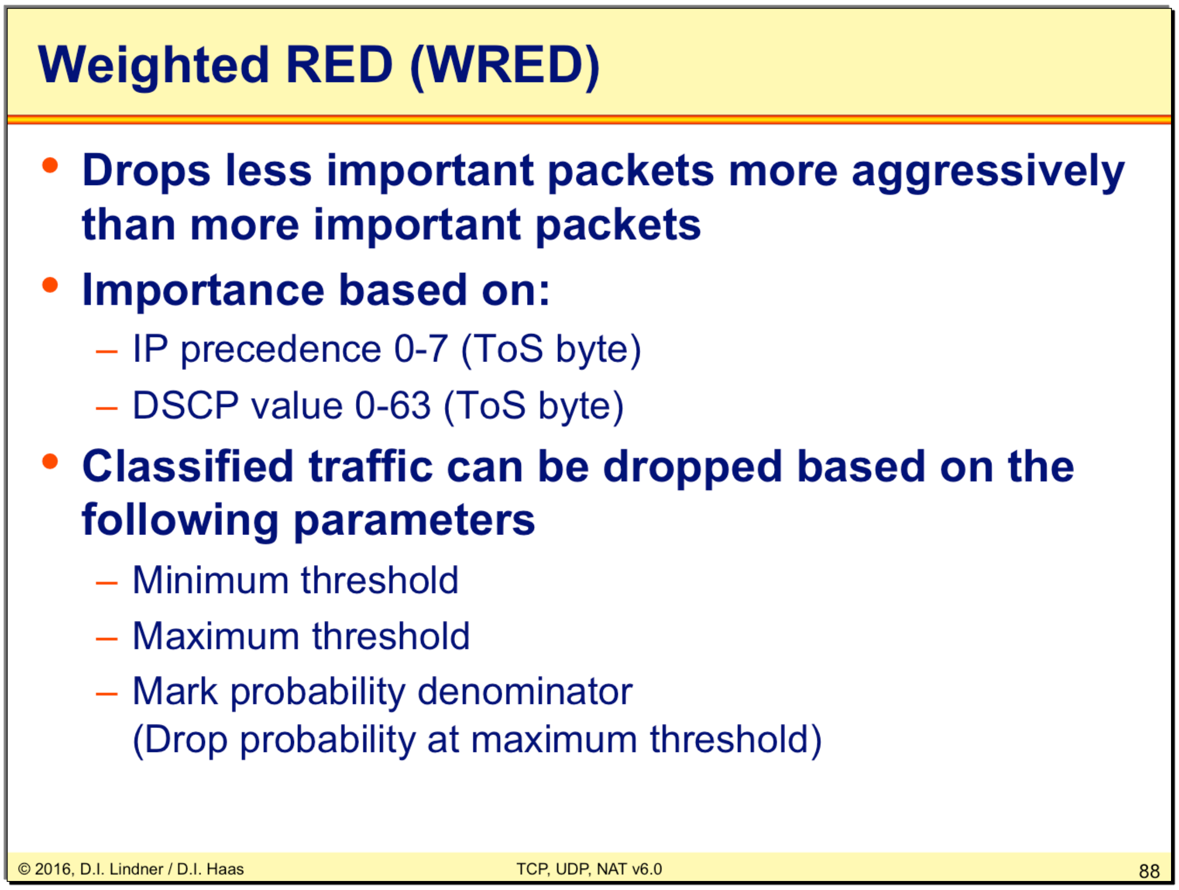
RED

FYI

- **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"

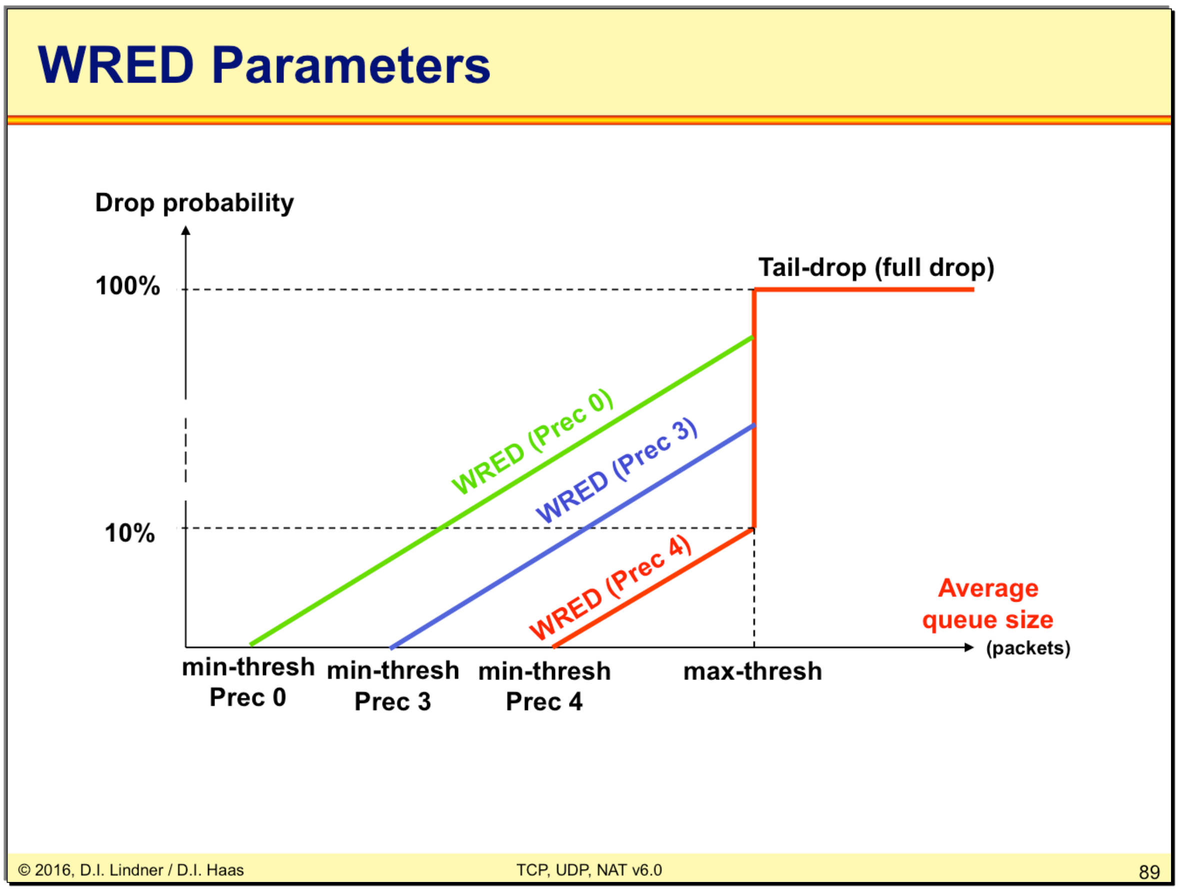
RED Parameters



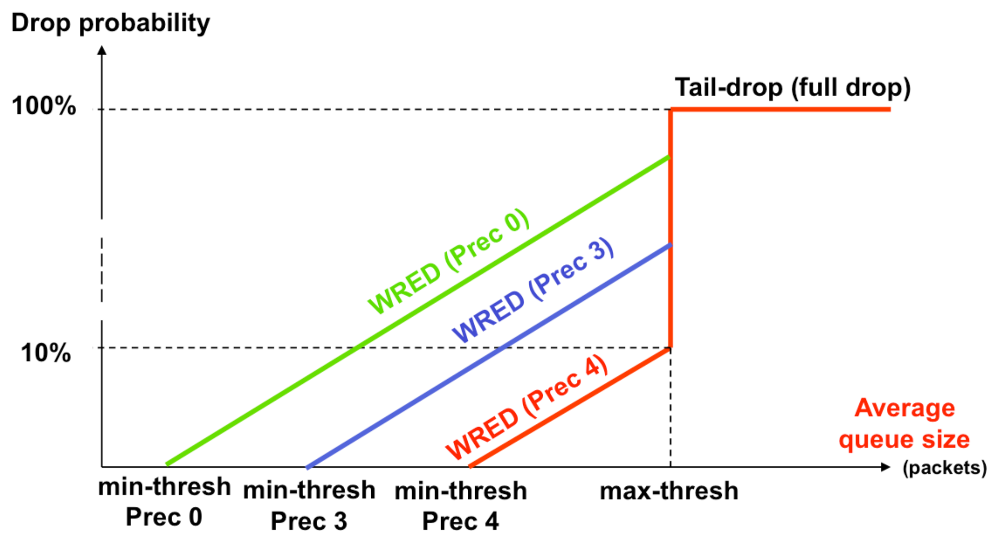


Weighted RED (WRED)

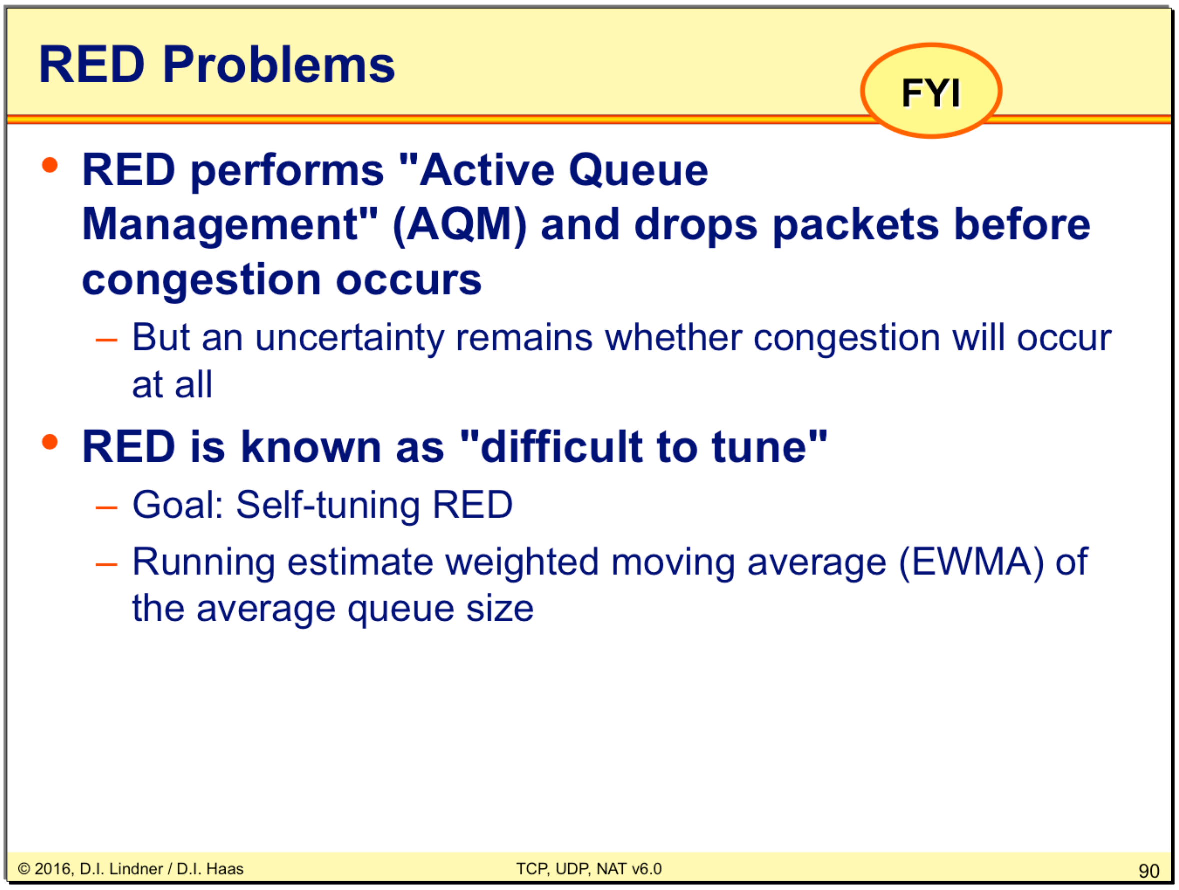
- **Drops less important packets more aggressively than more important packets**
- **Importance based on:**
  - IP precedence 0-7 (ToS byte)
  - DSCP value 0-63 (ToS byte)
- **Classified traffic can be dropped based on the following parameters**
  - Minimum threshold
  - Maximum threshold
  - Mark probability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



WRED Parameters



TCP, UDP, NAT v6.0 89



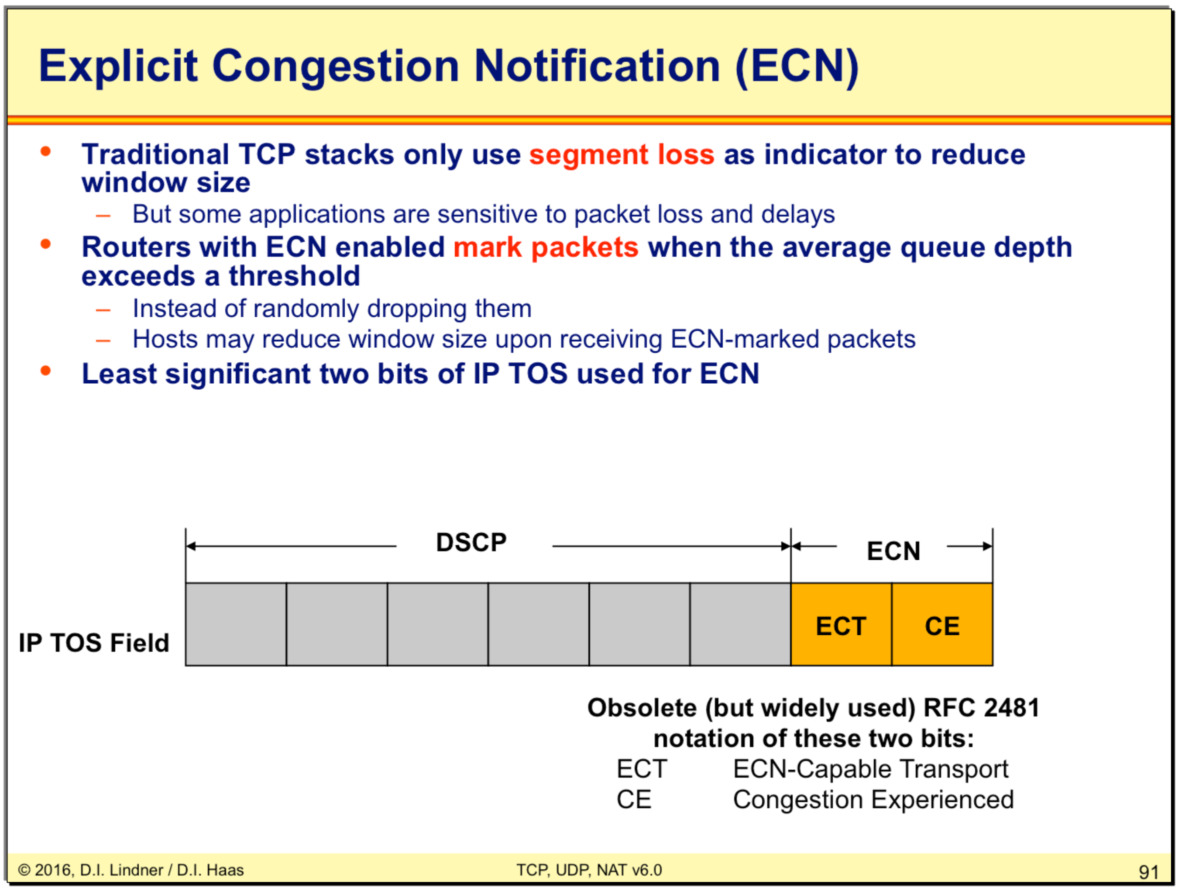
RED Problems

FYI

- **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

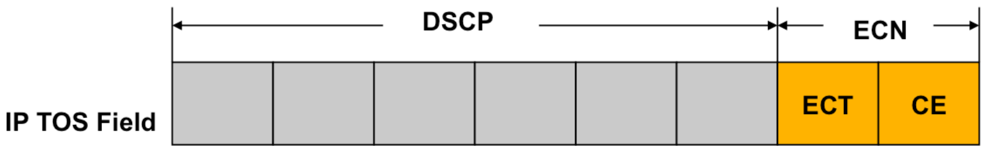
TCP, UDP, NAT v6.0 90

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



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

ECT	ECN-Capable Transport
CE	Congestion Experienced

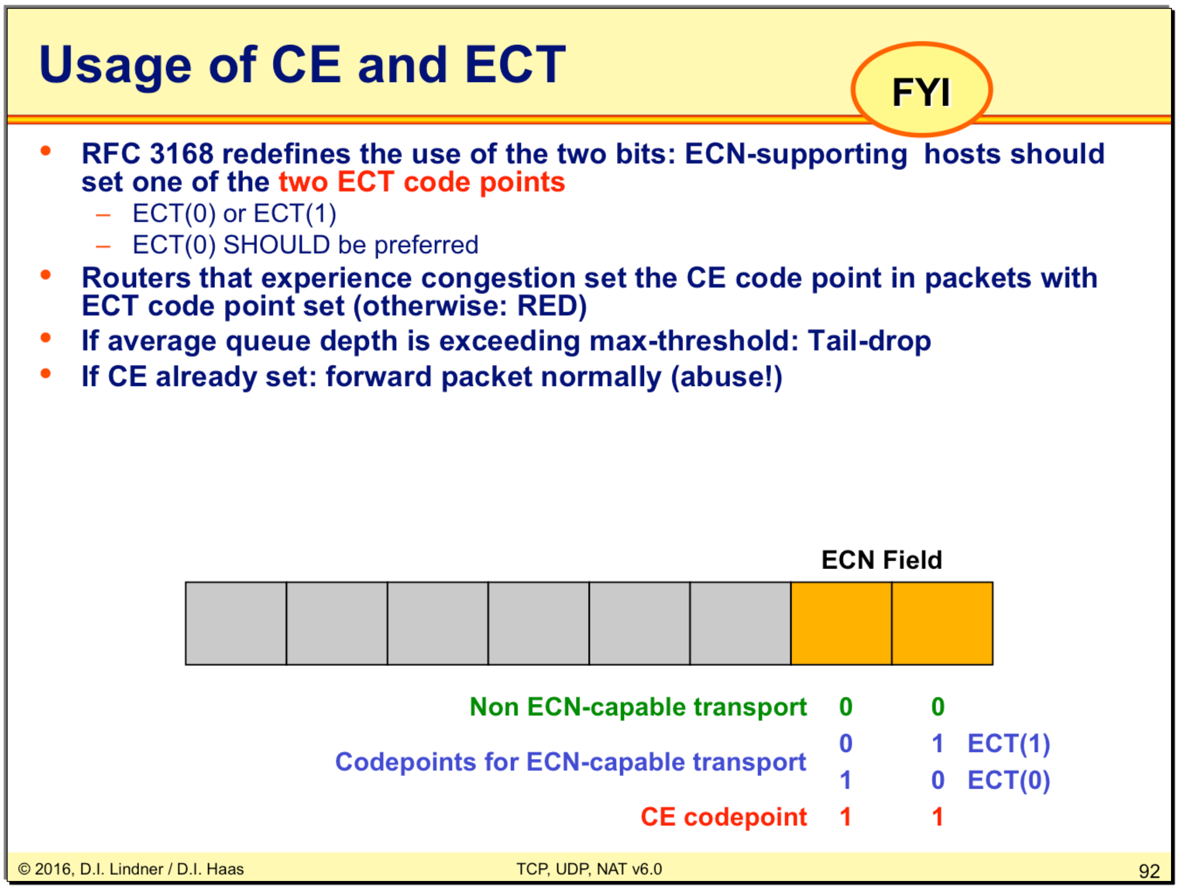
TCP, UDP, NAT v6.0

The limits of interpreting symptoms only:

Slow start and congestion avoidance try to maximize the traffic throughput without inclusion of network information. It is a host-based congestion control. Original IP idea: "Keep the network simple !" Slow start and congestion avoidance suspects congestion only by observing symptoms of the network.

Further improvements require an active inclusion of the intermediate network. This led to the introduction of an Explicit Congestion Notification mechanism which requires the help from routers that are expecting congestion (similar to the FECN seen in Frame Relay and EFCI in ATM)

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 it now.



Usage of CE and ECT

FYI

- **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!)**

ECN Field

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

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 92

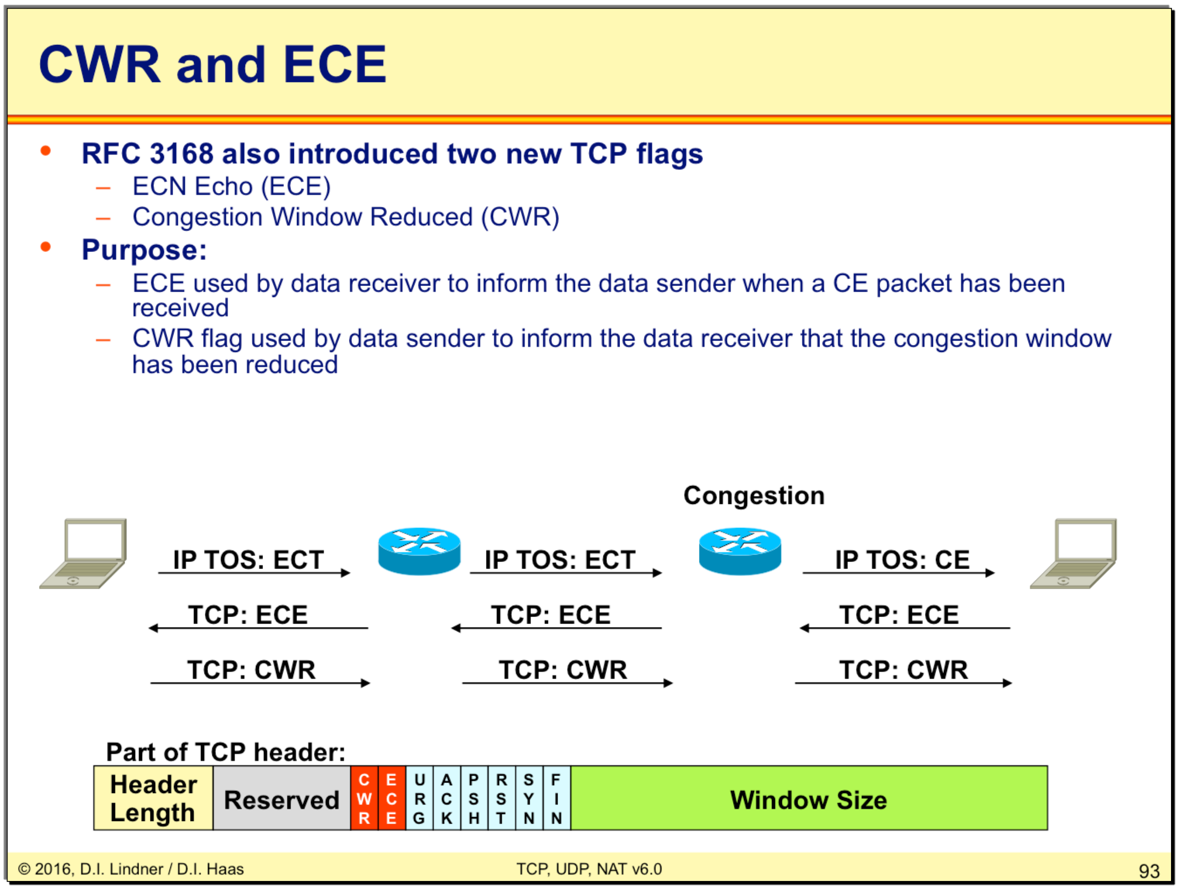
RFC 3168 - The Addition of Explicit Congestion Notification (ECN) to IP

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 hosts 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.

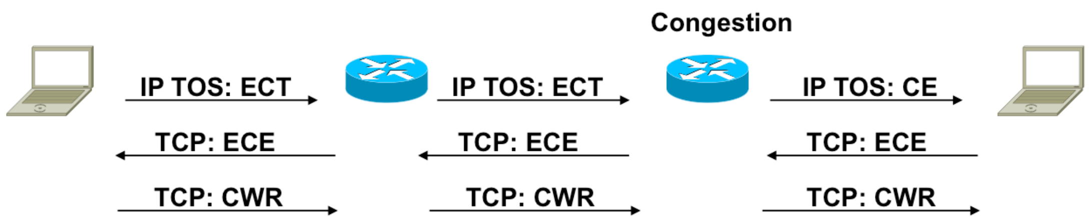
If you do not fully understand this please read the RFCs and search in the WWW – there a lots of debates about that.

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:

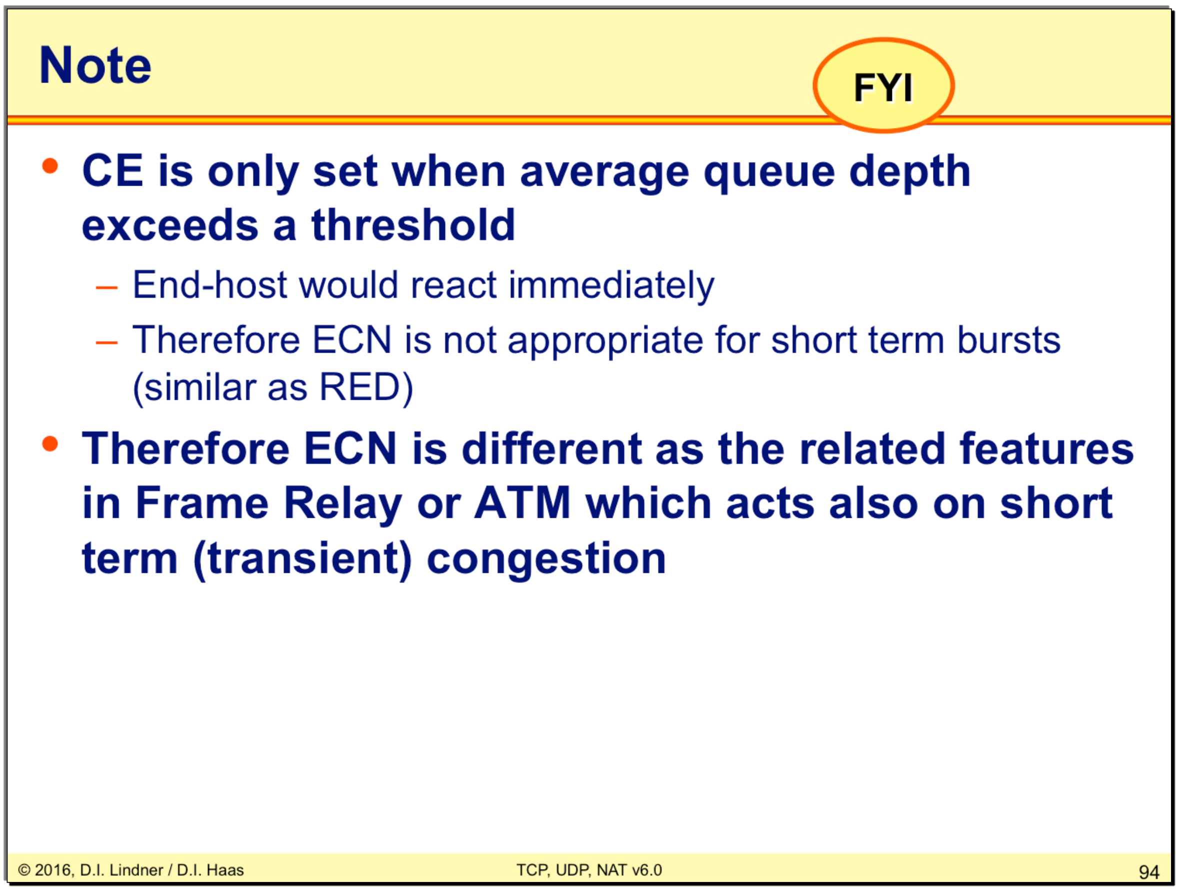
Header Length	Reserved	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 93

During TCP connection establishment, the ECN capability is negotiated. Additionally ECN requires the two TCP options "ECN-Echo" flag and "Congestion Window Reduced" (CWR) flag. 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.

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) and 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.

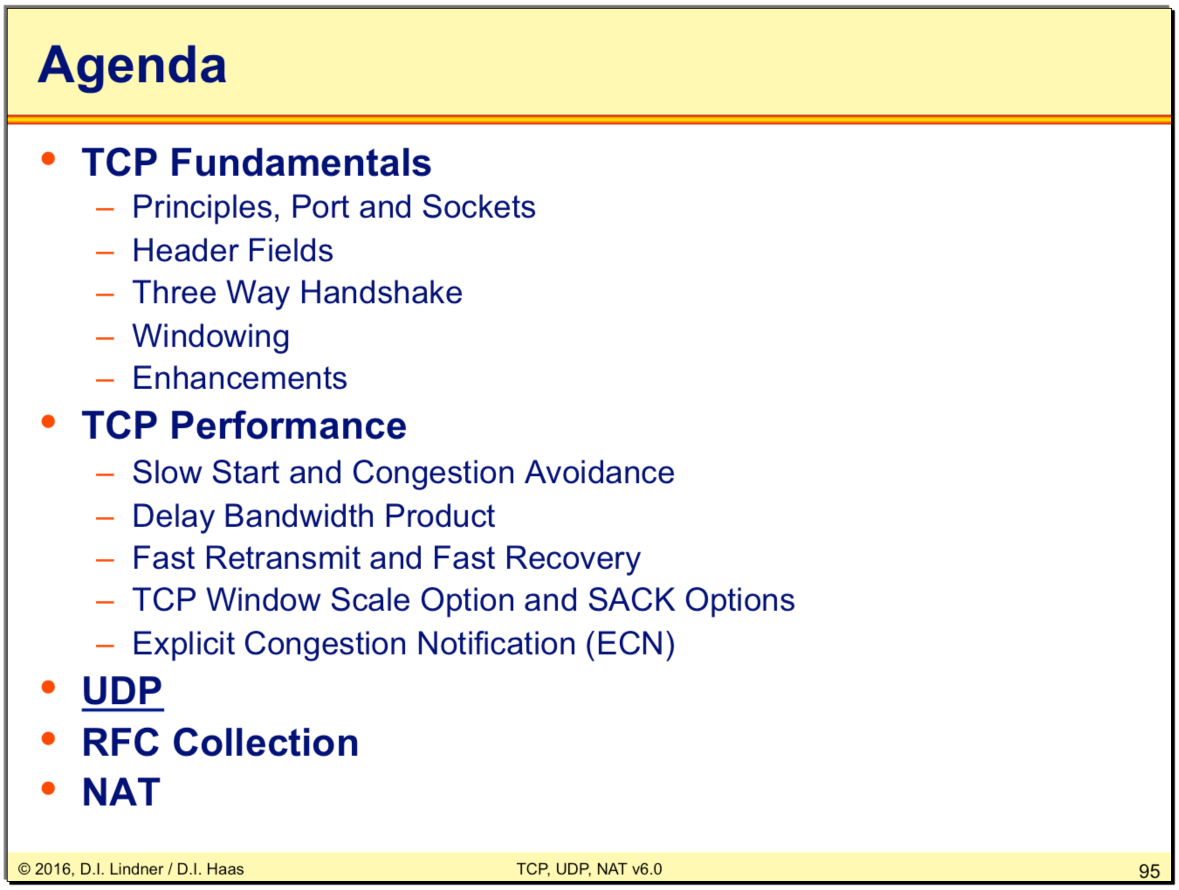
Main advantage: The sender does not have to wait for three duplicate ACKs to detect the congestion. He can react before dropping of segments will occur in the network by routers.



Note

FYI

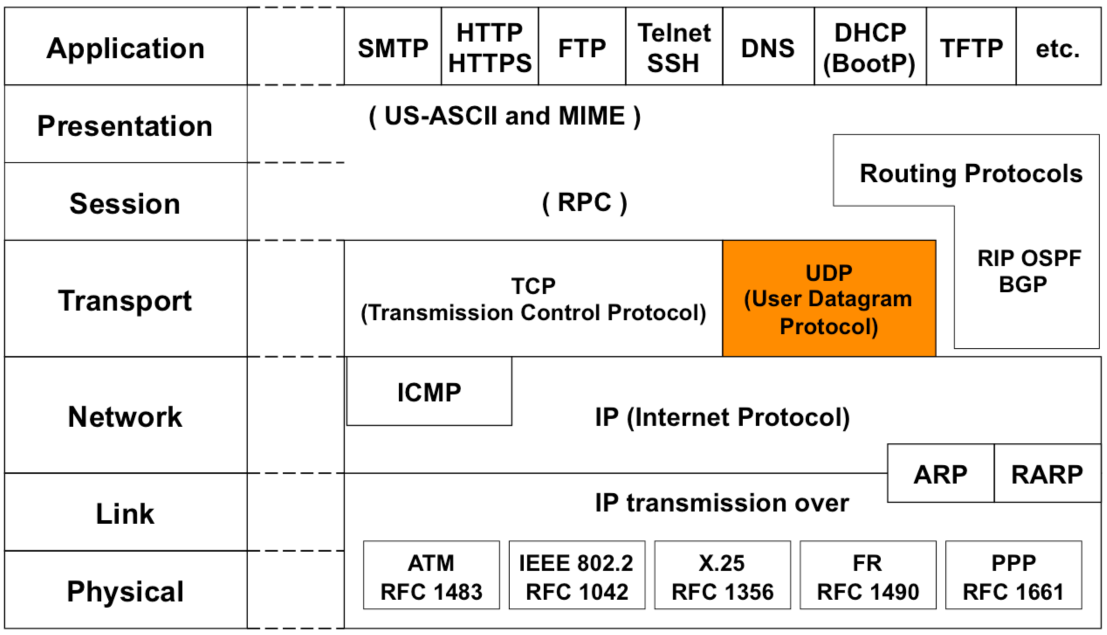
- **CE is only set when average queue depth exceeds a threshold**
  - End-host would react immediately
  - Therefore ECN is not appropriate for short term bursts (similar as RED)
- **Therefore ECN is different as the related features in Frame Relay or ATM which acts also on short term (transient) congestion**

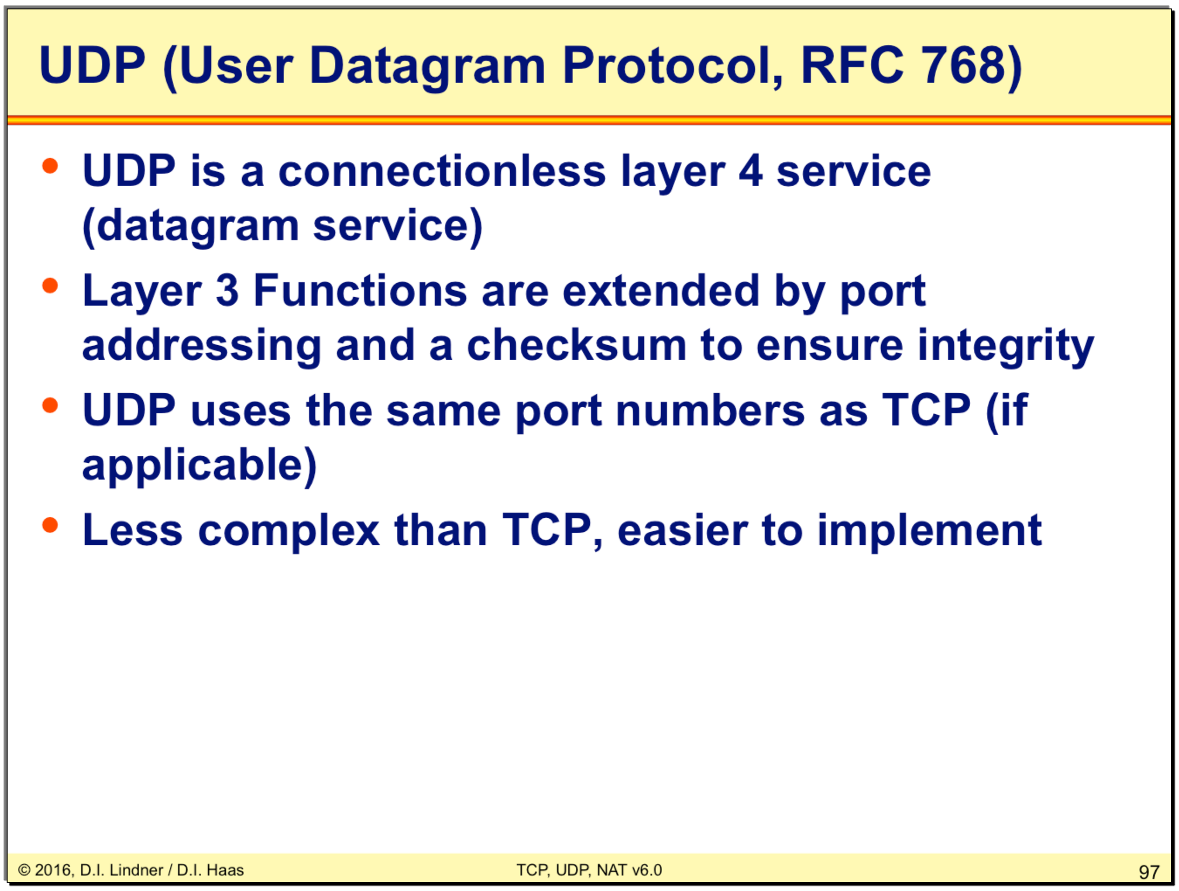


Agenda

- **TCP Fundamentals**
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Delay Bandwidth Product
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **UDP**
- **RFC Collection**
- **NAT**

TCP/IP Protocol Suite





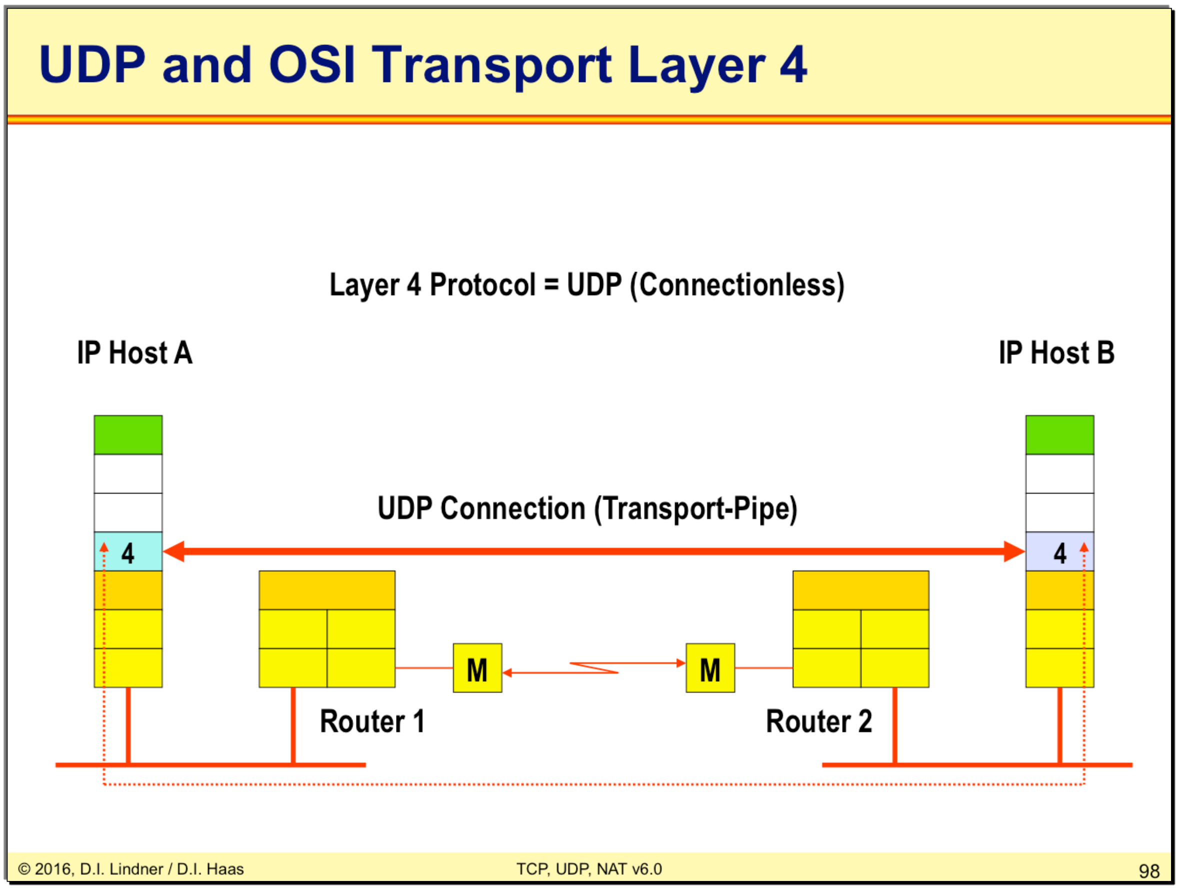
UDP (User Datagram Protocol, RFC 768)

- 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

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 97

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

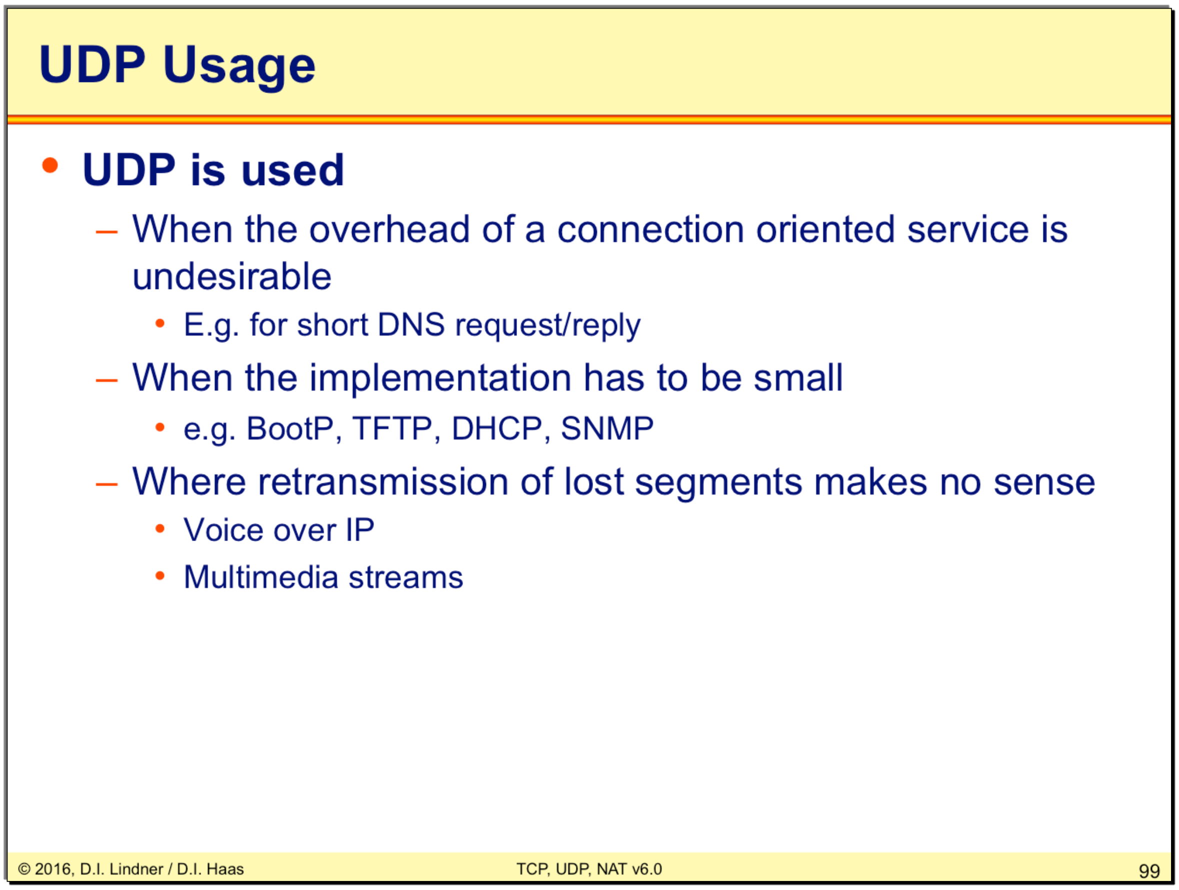
Typically applications that do not require error recovery but rely on speed use UDP, such as multimedia protocols.



UDP and OSI Transport Layer 4

Layer 4 Protocol = UDP (Connectionless)
IP Host A
IP Host B
UDP Connection (Transport-Pipe)
4
4
M
M
Router 1
Router 2

Recognizes that even the IP hosts see a transport pipe, this pipe is unreliable.



UDP Usage

- **UDP is used**
  - When the overhead of a connection oriented service is undesirable
    - E.g. for short DNS request/reply
  - When the implementation has to be small
    - e.g. BootP, TFTP, DHCP, SNMP
  - Where retransmission of lost segments makes no sense
    - Voice over IP
    - Multimedia streams

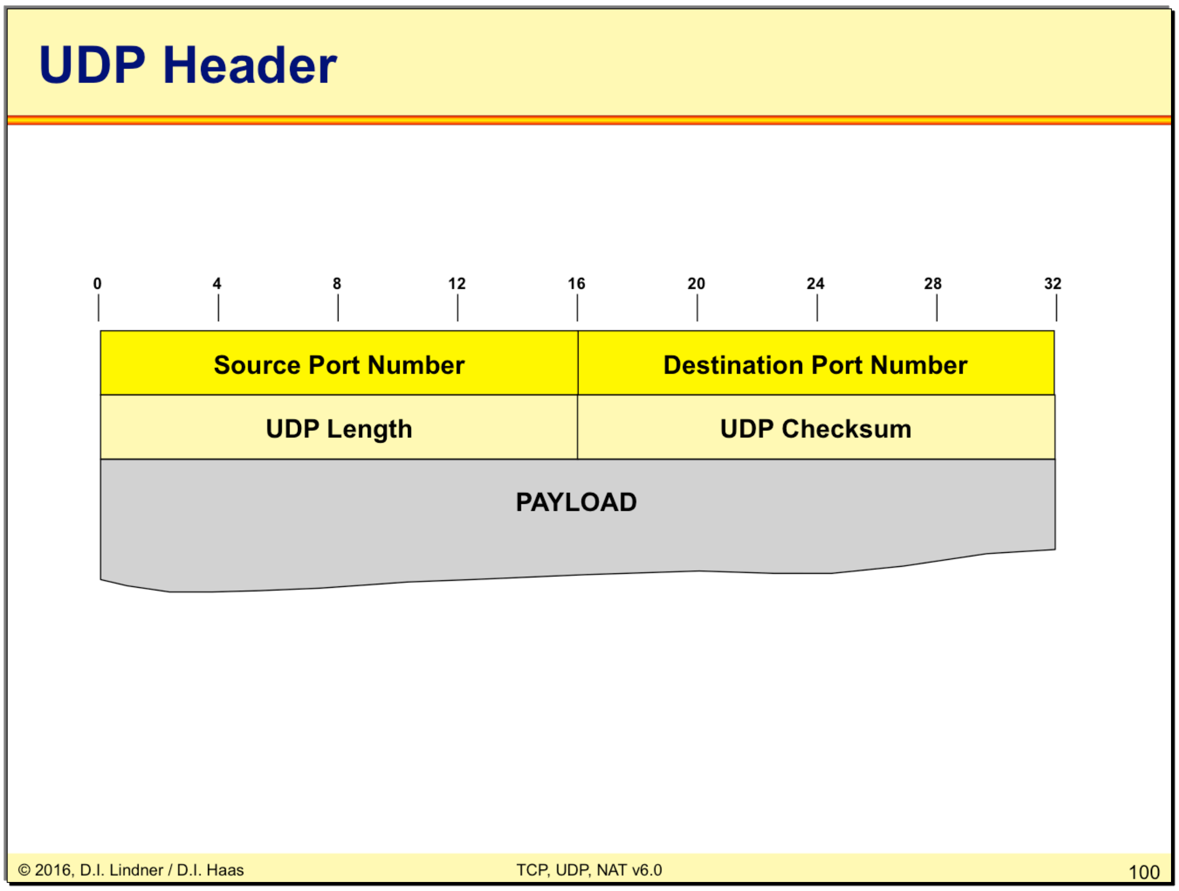
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 99

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

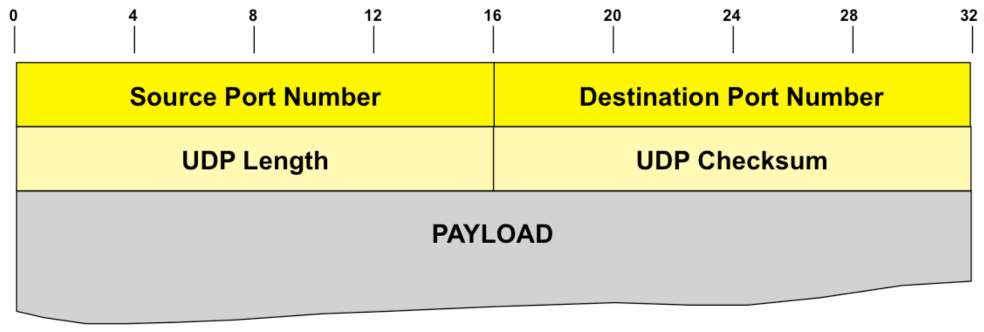
Note: Digitized voice is critical concerning delay but not against loss.

Voice is encapsulated in RTP (Real-time Transport Protocol) and RTP is encapsulated in UDP.

RTCP (RTP Control Protocol) propagates control information in the opposite direction. RTCP again is encapsulated in UDP.



UDP Header



TCP, UDP, NAT v6.0 100

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

Basically UDP adds just process addressing capabilities by usage of port numbers to best-effort service offered by IP.

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).

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 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.

Note that the checksum is often not calculated,

Important UDP Port Numbers

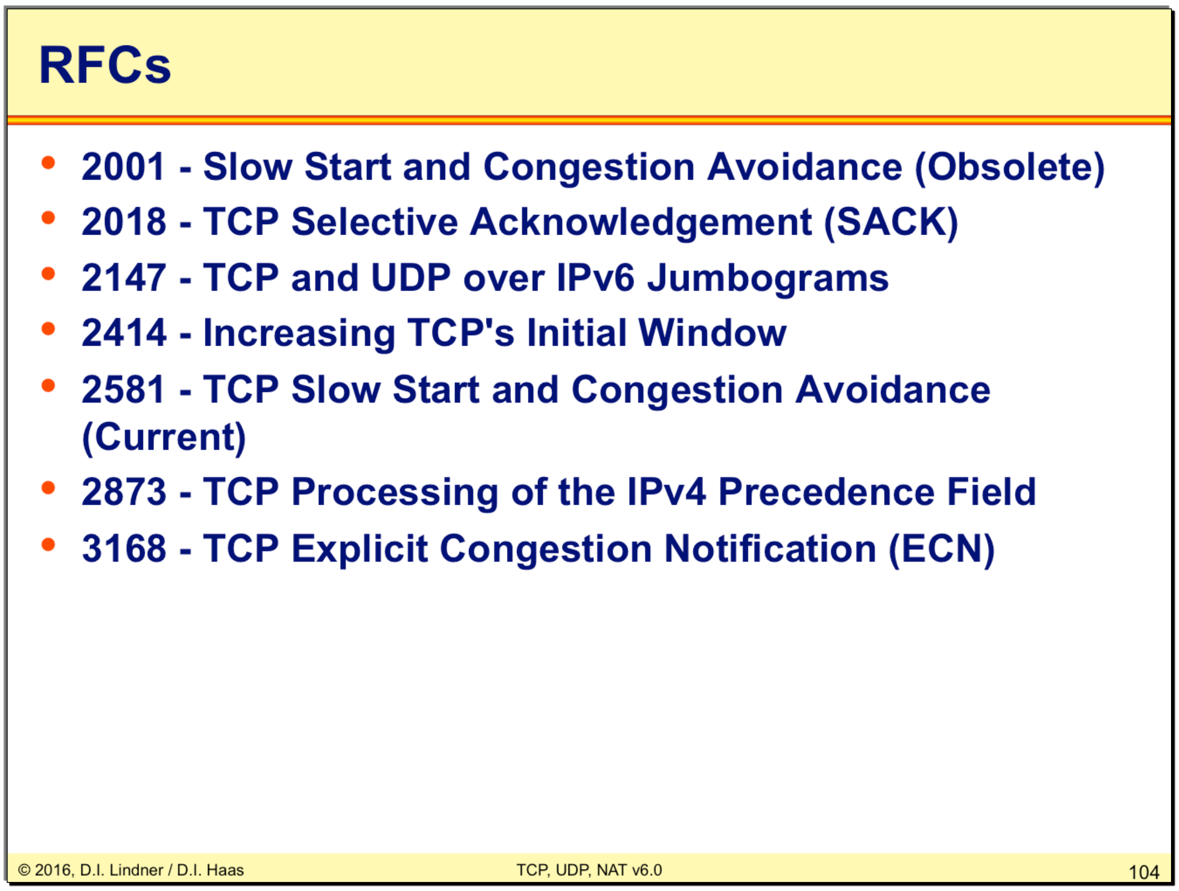
– 7	Echo
– 53	DOMAIN, Domain Name Server
– 67	BOOTPS, Bootstrap Protocol Server
– 68	BOOTPC, Bootstrap Protocol Client
– 69	TFTP, Trivial File Transfer Protocol
– 79	Finger
– 111	SUN RPC, Sun Remote Procedure Call
– 137	NetBIOS Name Service
– 138	NetBIOS Datagram Service
– 161	SNMP, Simple Network Management 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**
  - Principles, Port and Sockets
  - Header Fields
  - Three Way Handshake
  - Windowing
  - Enhancements
- **TCP Performance**
  - Slow Start and Congestion Avoidance
  - Delay Bandwidth Product
  - Fast Retransmit and Fast Recovery
  - TCP Window Scale Option and SACK Options
  - Explicit Congestion Notification (ECN)
- **UDP**
- **RFC Collection**
- **NAT**

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)



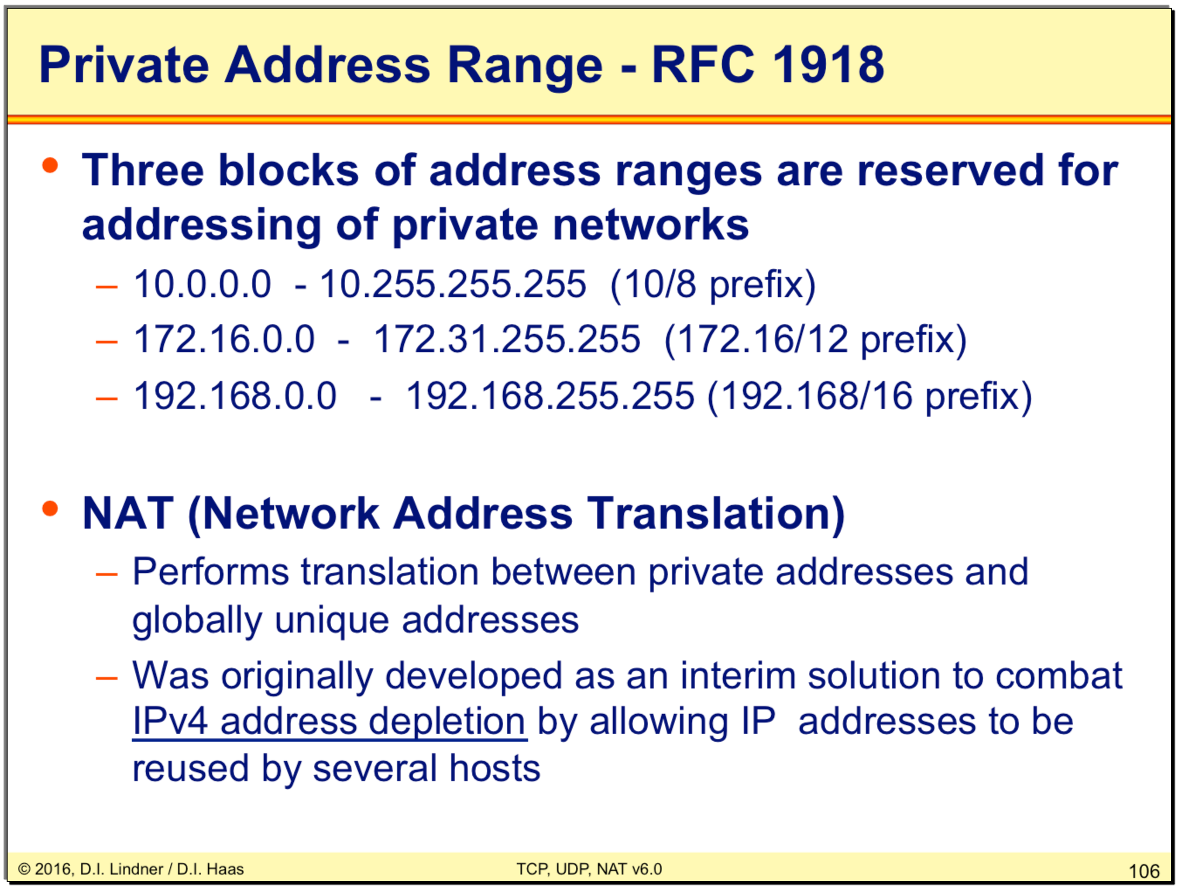
RFCs

- 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)

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 104

Agenda

- **TCP Fundamentals**
- **TCP Performance**
- **UDP**
- **RFC Collection**
- **NAT**
  - NAT Basics
  - NAPT
  - Virtual Server
  - Complex NAT
  - DNS Aspects
  - Load Balancing
  - RFCs



Private Address Range - RFC 1918

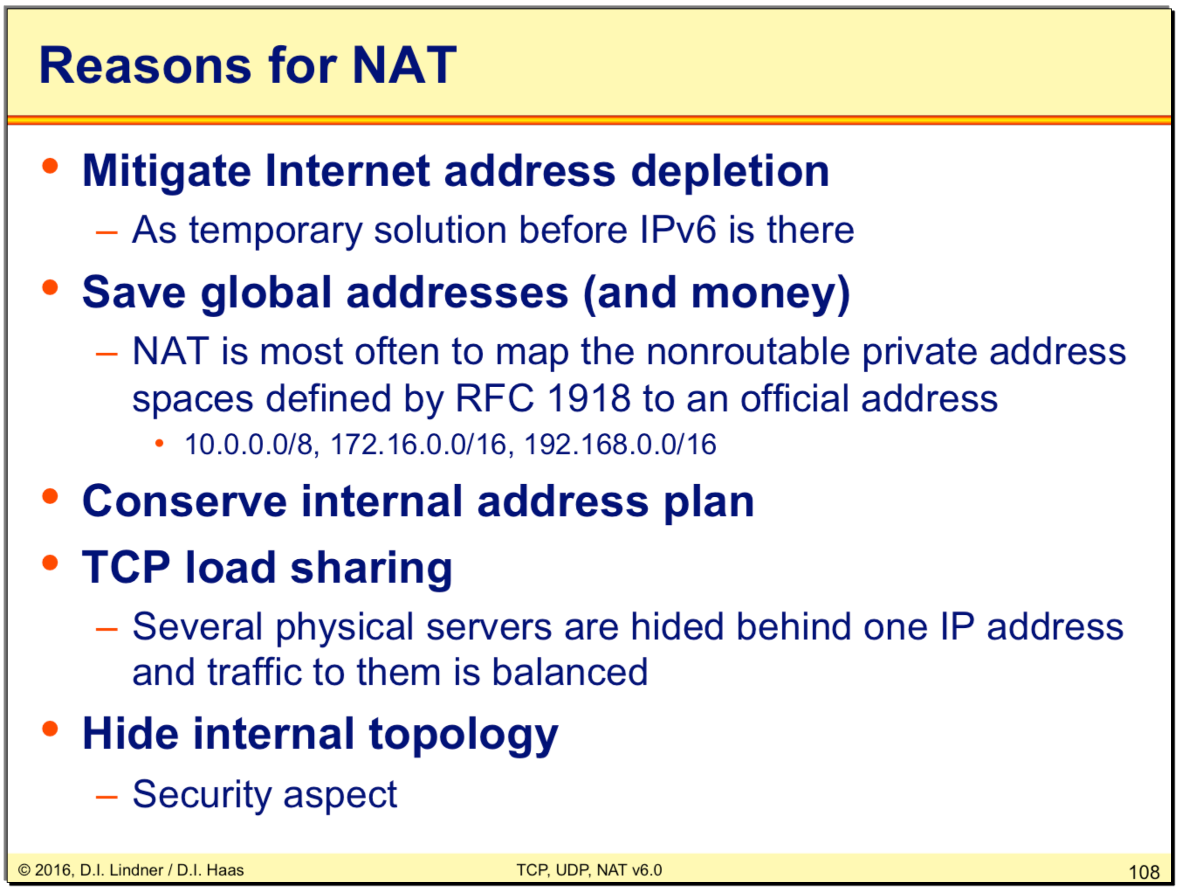
- **Three blocks of address ranges are reserved for addressing of private networks**
  - 10.0.0.0 - 10.255.255.255 (10/8 prefix)
  - 172.16.0.0 - 172.31.255.255 (172.16/12 prefix)
  - 192.168.0.0 - 192.168.255.255 (192.168/16 prefix)
- **NAT (Network Address Translation)**
  - Performs translation between private addresses and globally unique addresses
  - Was originally developed as an interim solution to combat IPv4 address depletion by allowing IP addresses to be reused by several hosts

TCP, UDP, NAT v6.0 106

In this chapter we discuss the idea of Network Address Translation and special issues associated to it. Invented in 1994, NAT became a quite popular technique to save official network addresses and to hide the own network topology from the Internet

Network Address Translation (NAT)

- **NAT**
  - First explained in RFC 1631
    - The address reuse solution is to place Network Address Translators (NAT) at the borders of stub domains
    - Each NAT box has a table consisting of pairs of local IP addresses and globally unique addresses performing address translation when passing IP Datagram's between a stub domain and the Internet and vice versa
    - The IP addresses inside the stub domain are not globally unique, they are reused in other domains, thus solving the address depletion problem
    - In most cases private addresses (RFC 1918) are used inside the stub domain (10.0.0.0/8, 172.16.0.0/16, 192.168.0.0/16)



Reasons for NAT

- **Mitigate Internet address depletion**
  - As temporary solution before IPv6 is there
- **Save global addresses (and money)**
  - NAT is most often to map the nonroutable private address spaces defined by RFC 1918 to an official address
    - 10.0.0.0/8, 172.16.0.0/16, 192.168.0.0/16
- **Conserve internal address plan**
- **TCP load sharing**
  - Several physical servers are hided behind one IP address and traffic to them is balanced
- **Hide internal topology**
  - Security aspect

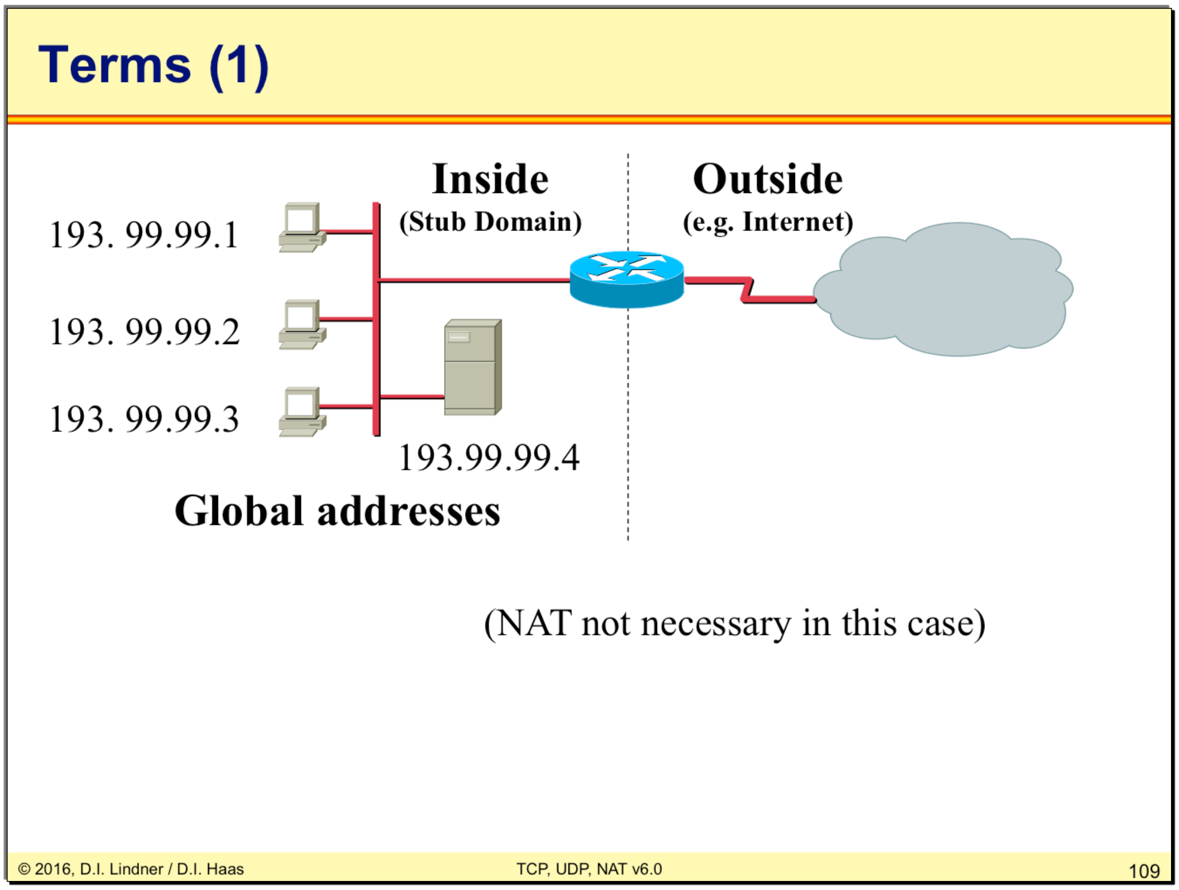
NAT allows a router to swap packet addresses. The initial idea was to mitigate IP address depletion by masquerading internal IP addresses with (perhaps a smaller number of) official addresses. We will discuss this later on.

The first and the second point reflect the same thing, but the first statement comes from the ISP while the second point is an argument for the customer.

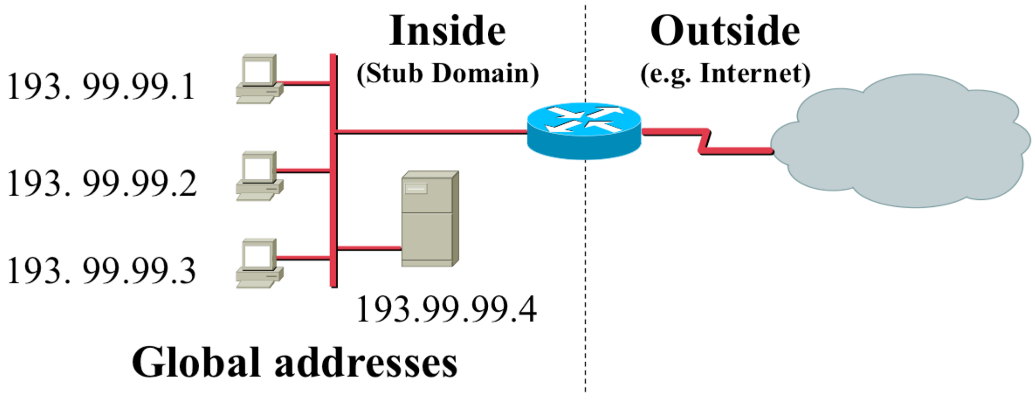
The third point means that the customer does not need to change her address plan when she switches to another ISP.

As stated in the fourth point, NAT additionally allows for TCP load sharing. Assume a bunch of servers represented by a single IP address to the outside.

Finally, NAT improves network security by hiding the actual host addresses. Frequently NAT boxes are combined with proxy and firewalling functions.



Terms (1)

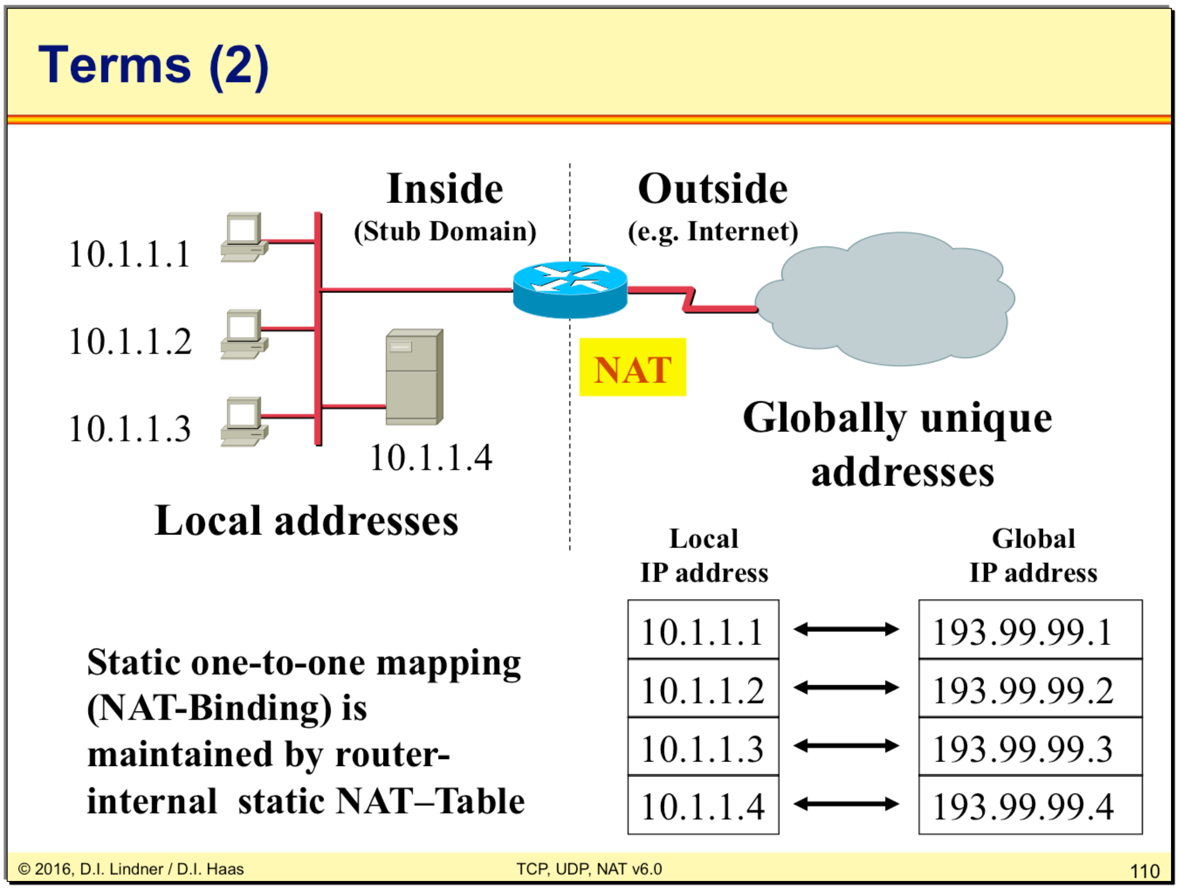


(NAT not necessary in this case)

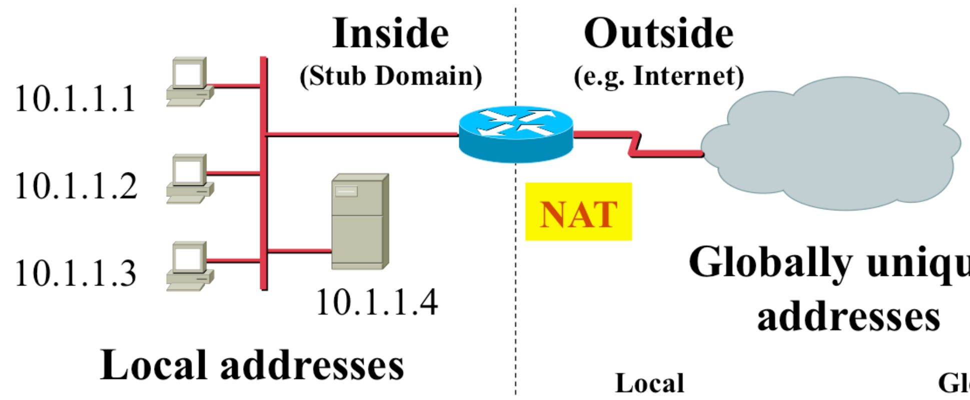
To understand standard documents such as RFCs or vendor documents such as Cisco white papers or similar, it is very important to understand four terms.

Firstly we have to distinguish the inside from the outside world. Inside is our own network (which we want to hide using a NAT-enabled router later on). Outside is the rest of the world, especially the Internet.

Secondly, suppose we do not use NAT. Therefore we use global addresses everywhere. That is, we use addresses that are registered by the NIC and can be seen from outside.



Terms (2)

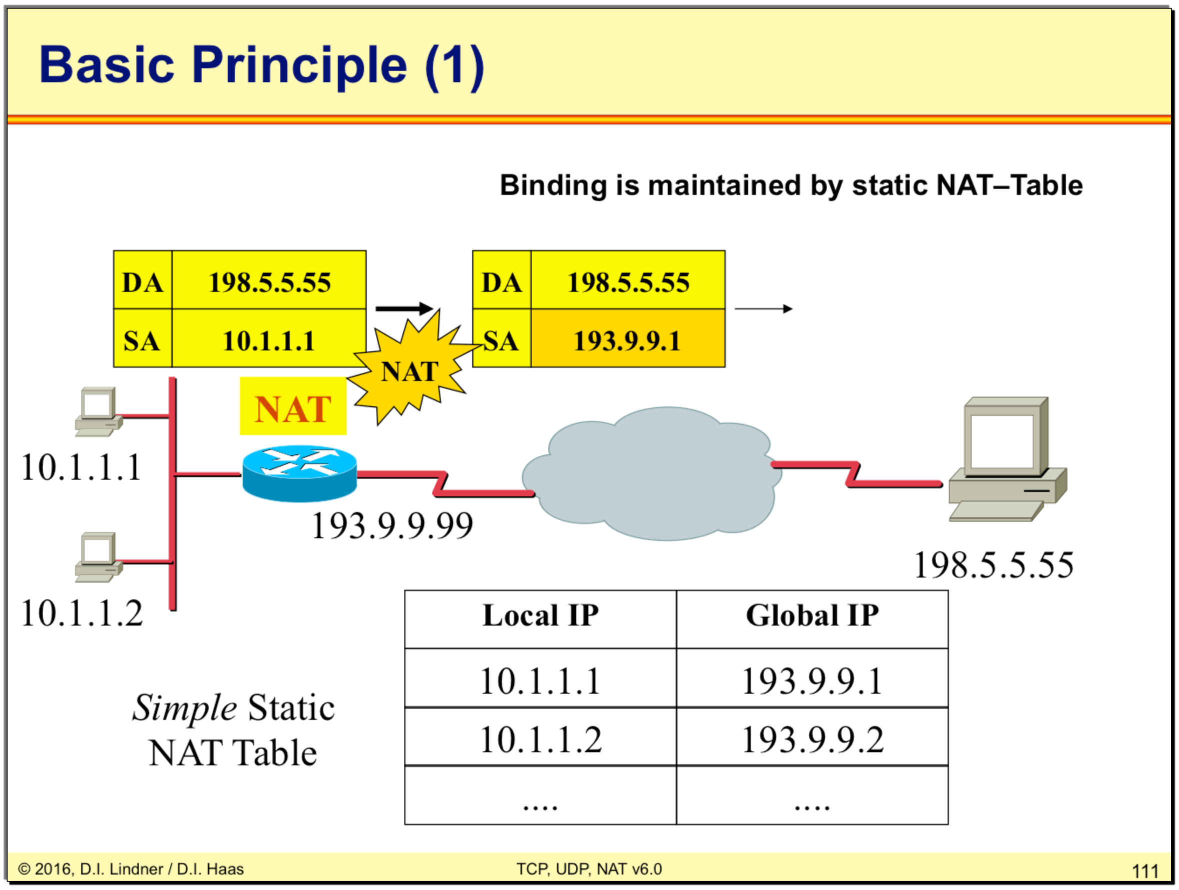


Static one-to-one mapping (NAT-Binding) is maintained by router-internal static NAT–Table

Local IP address		Global IP address
10.1.1.1	⟷	193.99.99.1
10.1.1.2	⟷	193.99.99.2
10.1.1.3	⟷	193.99.99.3
10.1.1.4	⟷	193.99.99.4

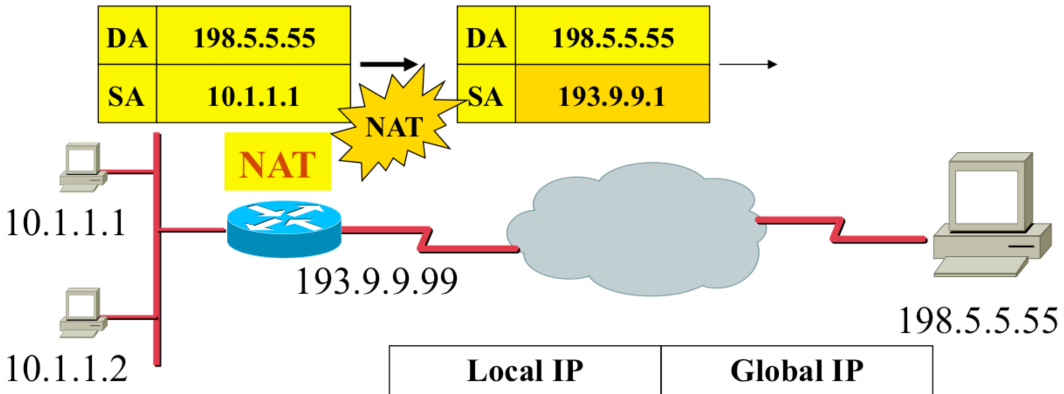
© 2016, D.I. Lindner / D.I. Haas   TCP, UDP, NAT v6.0   110

Using a NAT enabled router we can use inside local addresses which are not unique in the world. This addresses are not registered and must be translated to outside global addresses.



Basic Principle (1)

Binding is maintained by static NAT–Table

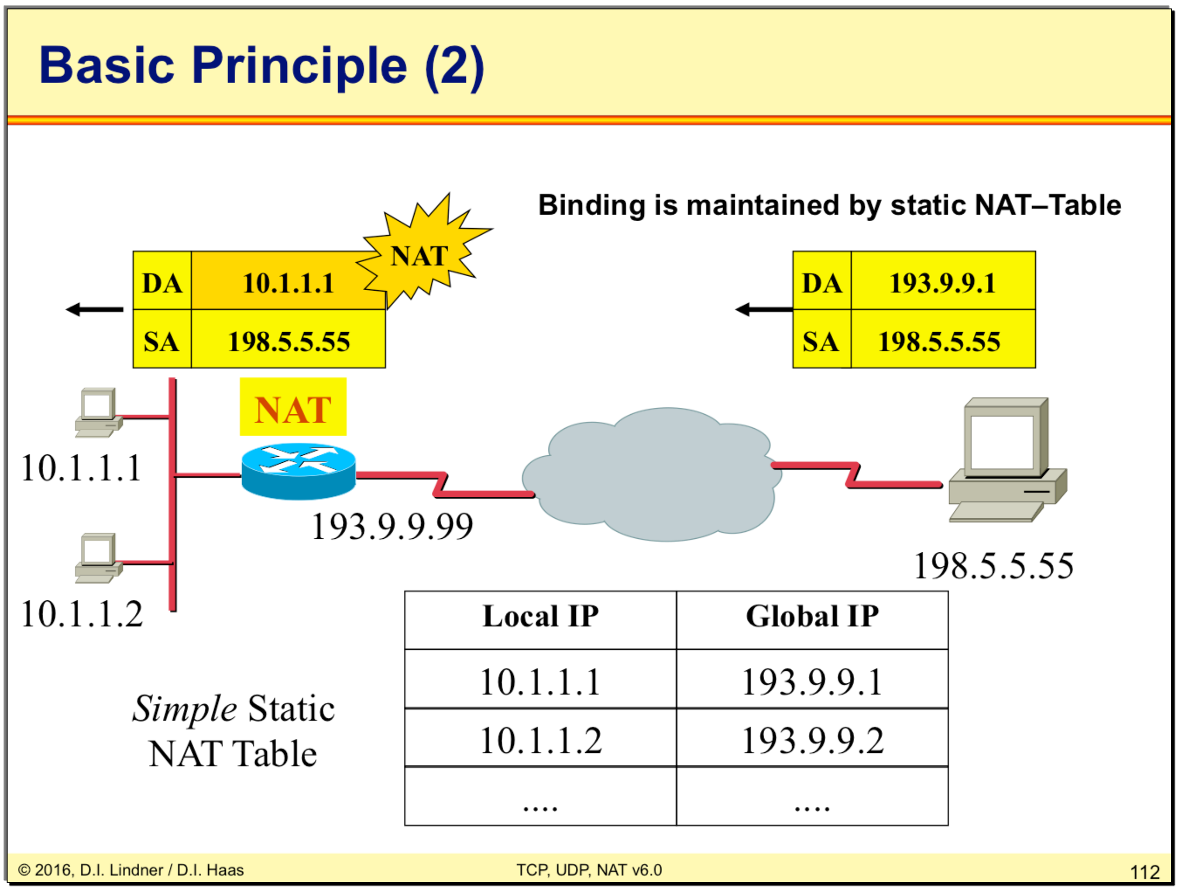


Simple Static NAT Table

Local IP	Global IP
10.1.1.1	193.9.9.1
10.1.1.2	193.9.9.2
....	....

TCP, UDP, NAT v6.0 111

1) Suppose the user at host 10.1.1.1 opens a connection to host 198.5.5.55.
2) The first packet that the router receives from host 10.1.1.1 causes the router to check its NAT table.
3) The router replaces the source address with the global address found in the NAT table.



Basic Principle (2)

Binding is maintained by static NAT–Table

Simple Static NAT Table

Local IP	Global IP
10.1.1.1	193.9.9.1
10.1.1.2	193.9.9.2
....	....

TCP, UDP, NAT v6.0 112

Host 198.5.5.55 responds to host 10.1.1.1 by using the global address 193.9.9.1 as destination address.

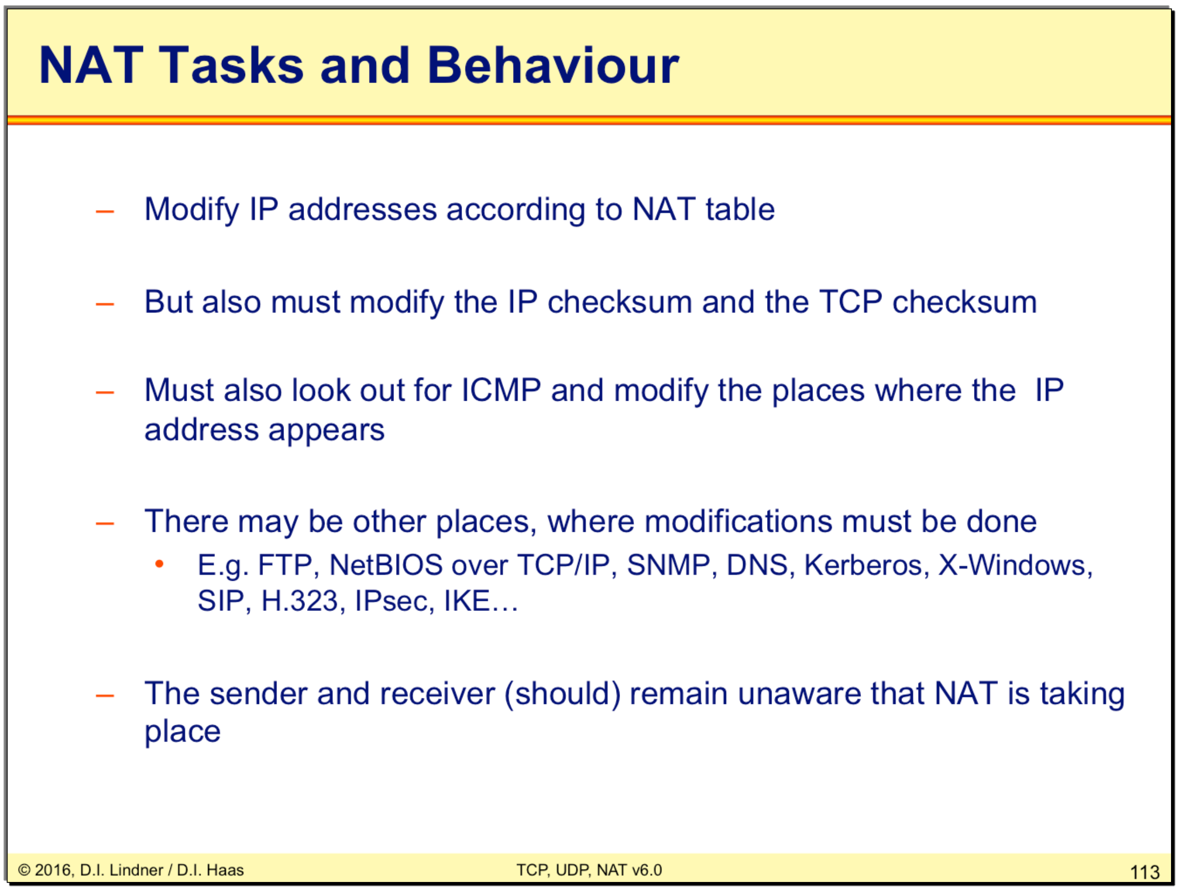
When the router receives a packet with the inside global address 193.9.9.1 it performs a NAT table lookup to determine the associated inside local address.

The router translate 193.9.9.1 to 10.1.1.1 and forwards the packet to host 10.1.1.1.

FYI:

Inside-to-outside translation occurs after routing

Outside-to-inside translation occurs before routing



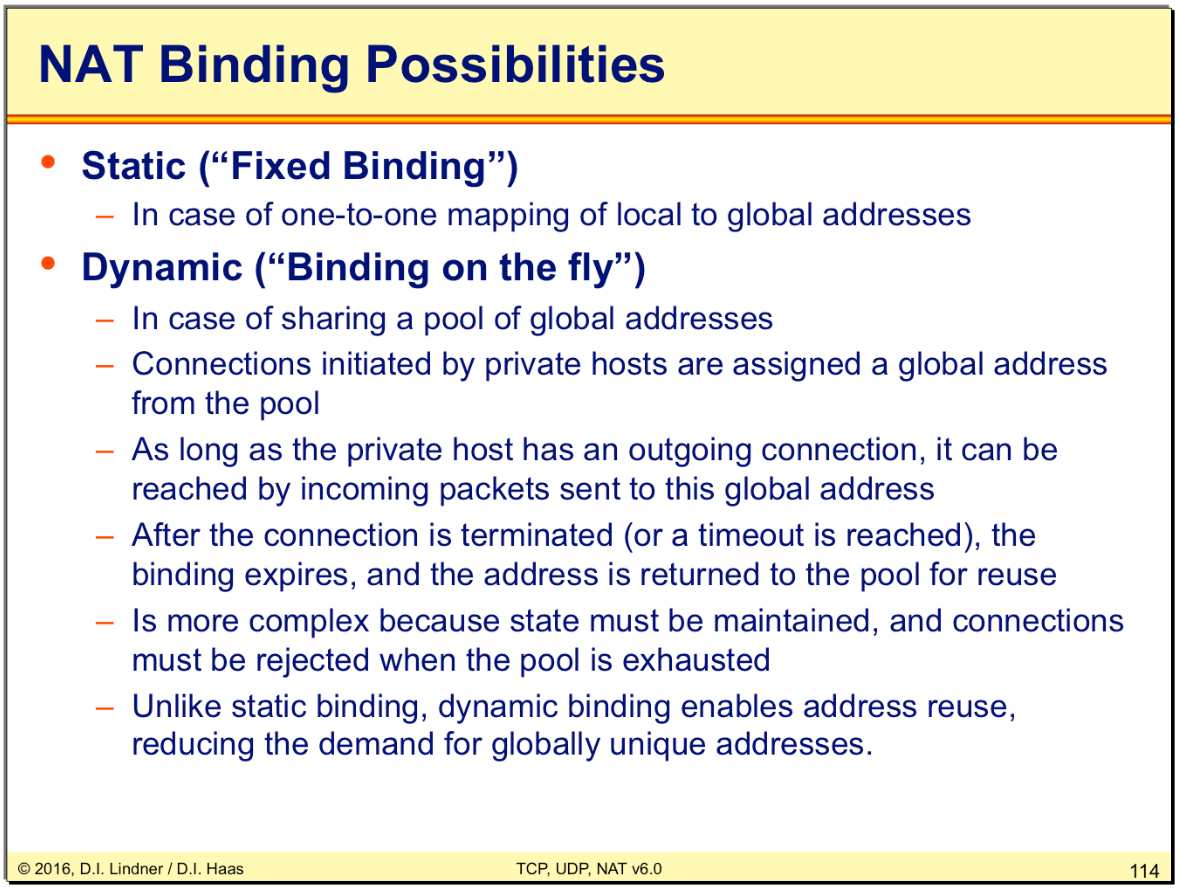
NAT Tasks and Behaviour

- Modify IP addresses according to NAT table
- But also must modify the IP checksum and the TCP checksum
- Must also look out for ICMP and modify the places where the IP address appears
- There may be other places, where modifications must be done
  - E.g. FTP, NetBIOS over TCP/IP, SNMP, DNS, Kerberos, X-Windows, SIP, H.323, IPsec, IKE…
- The sender and receiver (should) remain unaware that NAT is taking place

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 113

Note: TCP's checksum also covers a pseudo IP header which contains the source and destination IP addresses.

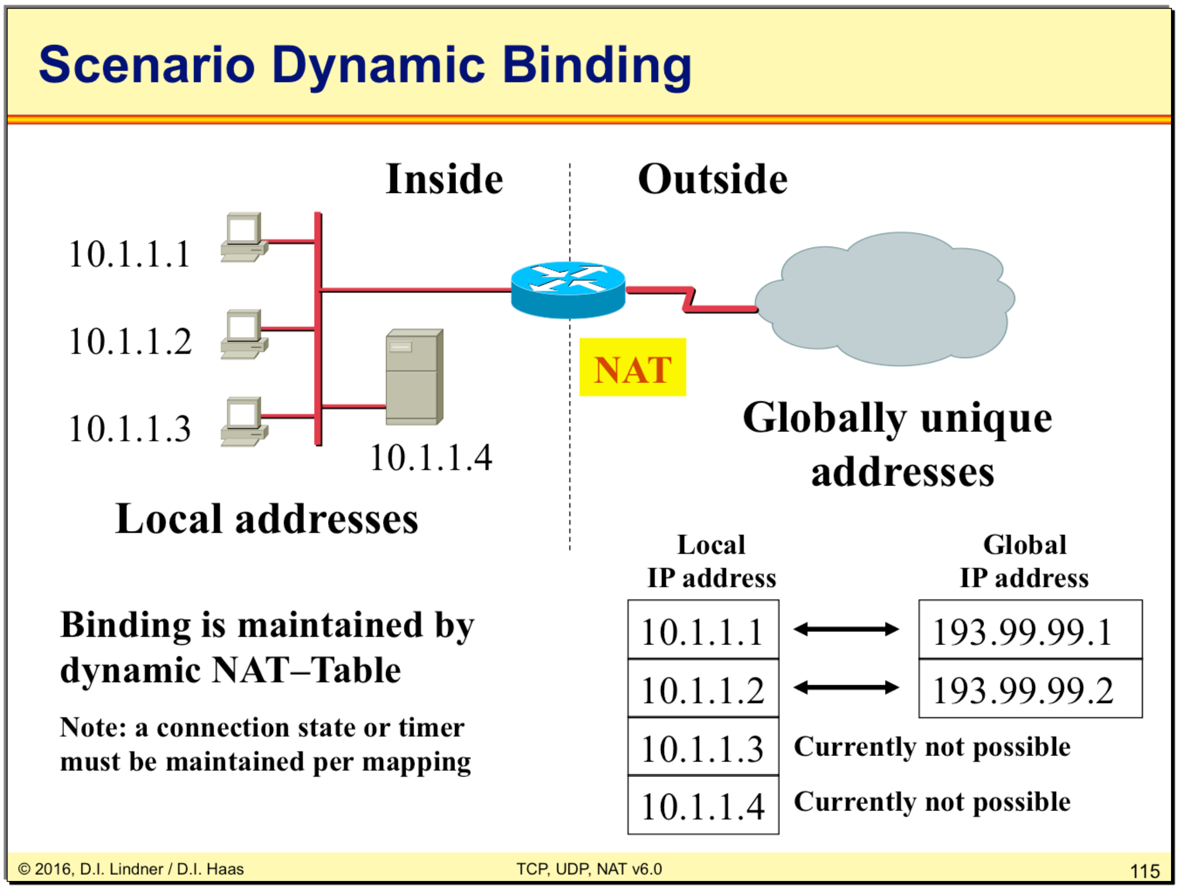
NAT devices were intended to be unmanaged devices that are transparent to end-to-end protocol interaction. Hence no specific interaction is required between the end systems and the NAT device.



NAT Binding Possibilities

- **Static ("Fixed Binding")**
  - In case of one-to-one mapping of local to global addresses
- **Dynamic ("Binding on the fly")**
  - In case of sharing a pool of global addresses
  - Connections initiated by private hosts are assigned a global address from the pool
  - As long as the private host has an outgoing connection, it can be reached by incoming packets sent to this global address
  - After the connection is terminated (or a timeout is reached), the binding expires, and the address is returned to the pool for reuse
  - Is more complex because state must be maintained, and connections must be rejected when the pool is exhausted
  - Unlike static binding, dynamic binding enables address reuse, reducing the demand for globally unique addresses.

© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 114



Scenario Dynamic Binding

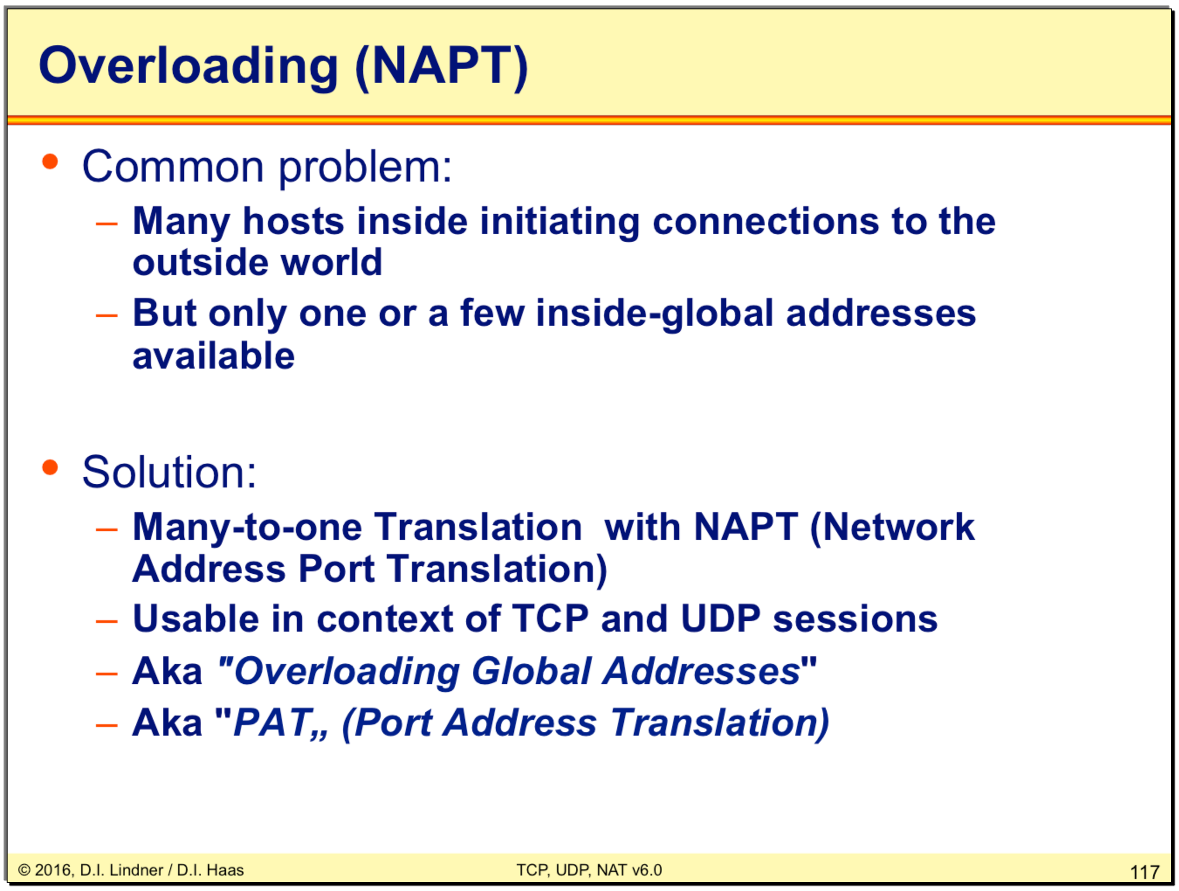
Binding is maintained by dynamic NAT–Table

Note: a connection state or timer must be maintained per mapping

Local IP address		Global IP address
10.1.1.1	⟷	193.99.99.1
10.1.1.2	⟷	193.99.99.2
10.1.1.3	Currently not possible	
10.1.1.4	Currently not possible	

If no translation entry exists, the router determines that the source address must be translated dynamically and selects a legal global address from the *predefined* dynamic *address pool* and creates a translation entry.

Agenda



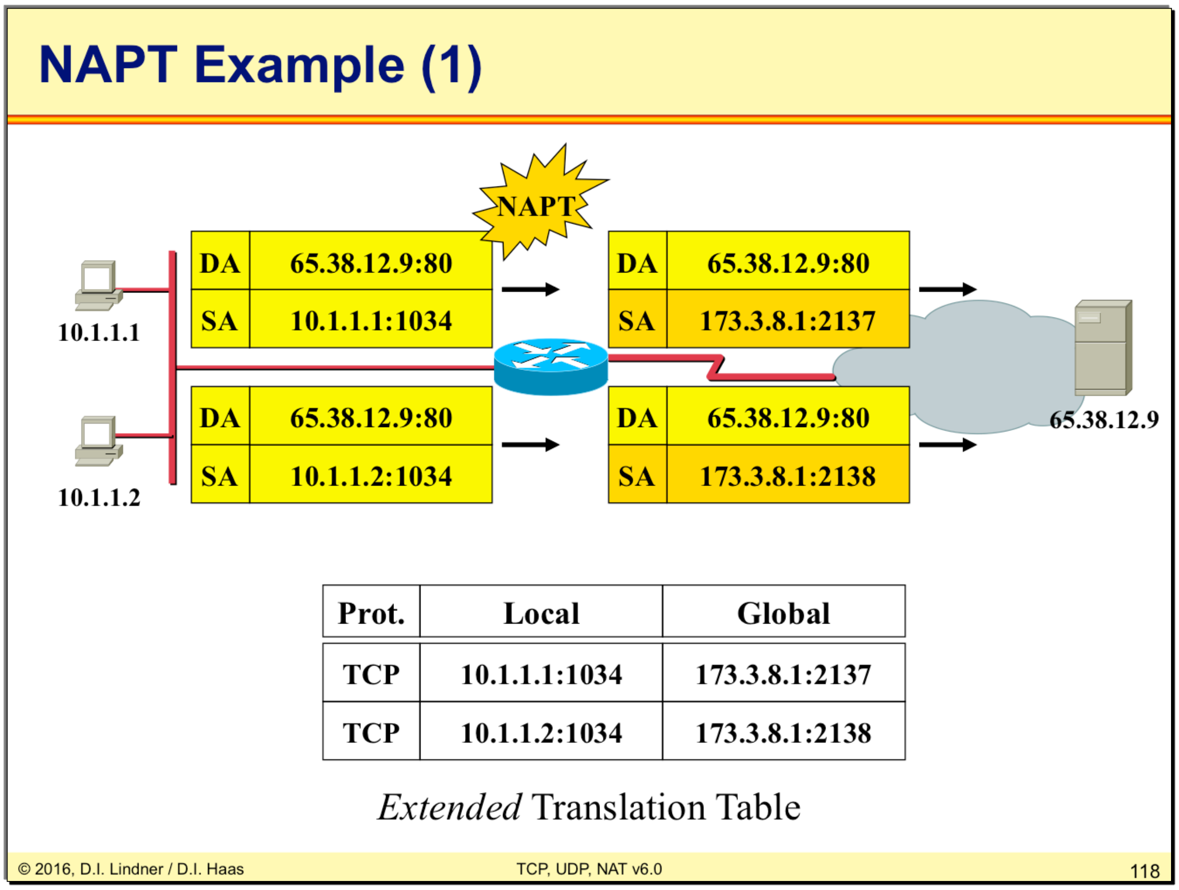
Overloading (NAPT)

- Common problem:
  - Many hosts inside initiating connections to the outside world
  - But only one or a few inside-global addresses available
- Solution:
  - Many-to-one Translation with NAPT (Network Address Port Translation)
  - Usable in context of TCP and UDP sessions
  - Aka "*Overloading Global Addresses*"
  - Aka "*PAT„ (Port Address Translation)*

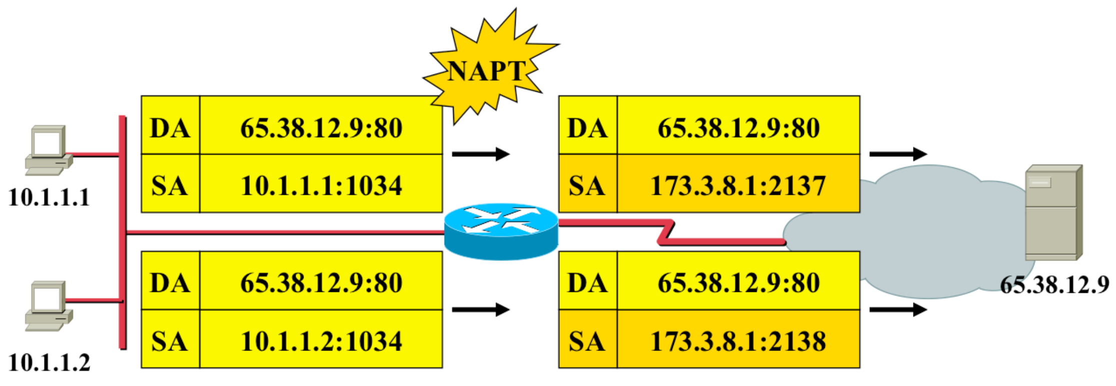
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 117

Many-to-one translation is accomplished by identifying each traffic according to the source port numbers. This method is commonly known as Port Address Translation (PAT). In the IETF documents you will also see the abbreviation NAPT. In the Linux world it is known as masquerading.

When N inside hosts use the same source port numbers, the PAT-routers will increase N-1 of these identical source port numbers to the next free values.



NAPT Example (1)



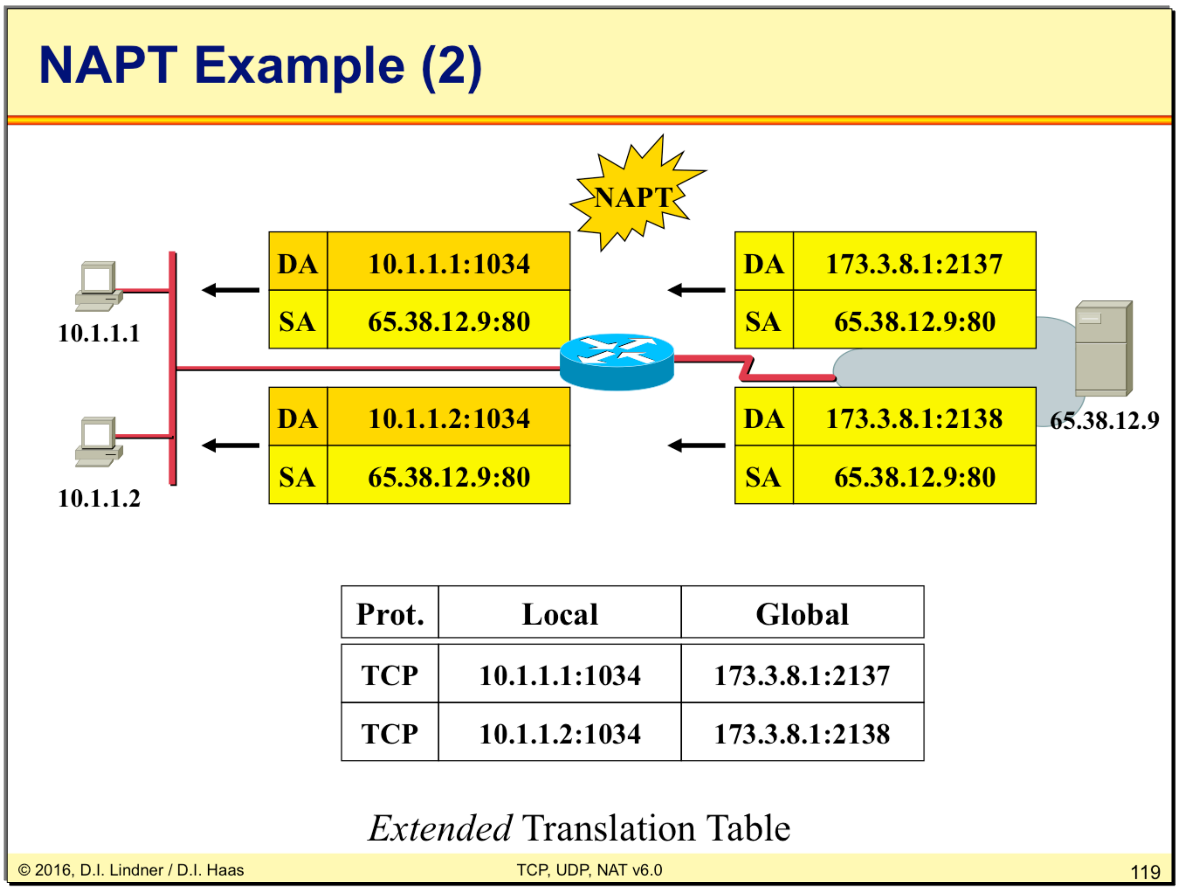
Prot.	Local	Global
TCP	10.1.1.1:1034	173.3.8.1:2137
TCP	10.1.1.2:1034	173.3.8.1:2138

Extended Translation Table

The port number is the differentiator. Note that the TCP and UDP port number range allows up to 65,536 number per IP address. This number is the upper limit for simultaneous transmissions per inside-global IP address.

If the port numbers run out, PAT will move to the next IP address and try to allocate the original source port again. This continues until all available ports and IP addresses are utilized. If a PAT router run out of addresses, it drops the packet and sends an ICMP Host Unreachable message.

Generally, NAT/PAT is only practical when relatively few hosts in a stub domain communicate outside of the domain at the same time. In this case, only a small subset of the IP addresses in the own domain must be translated into globally unique IP addresses.



NAPT Example (2)

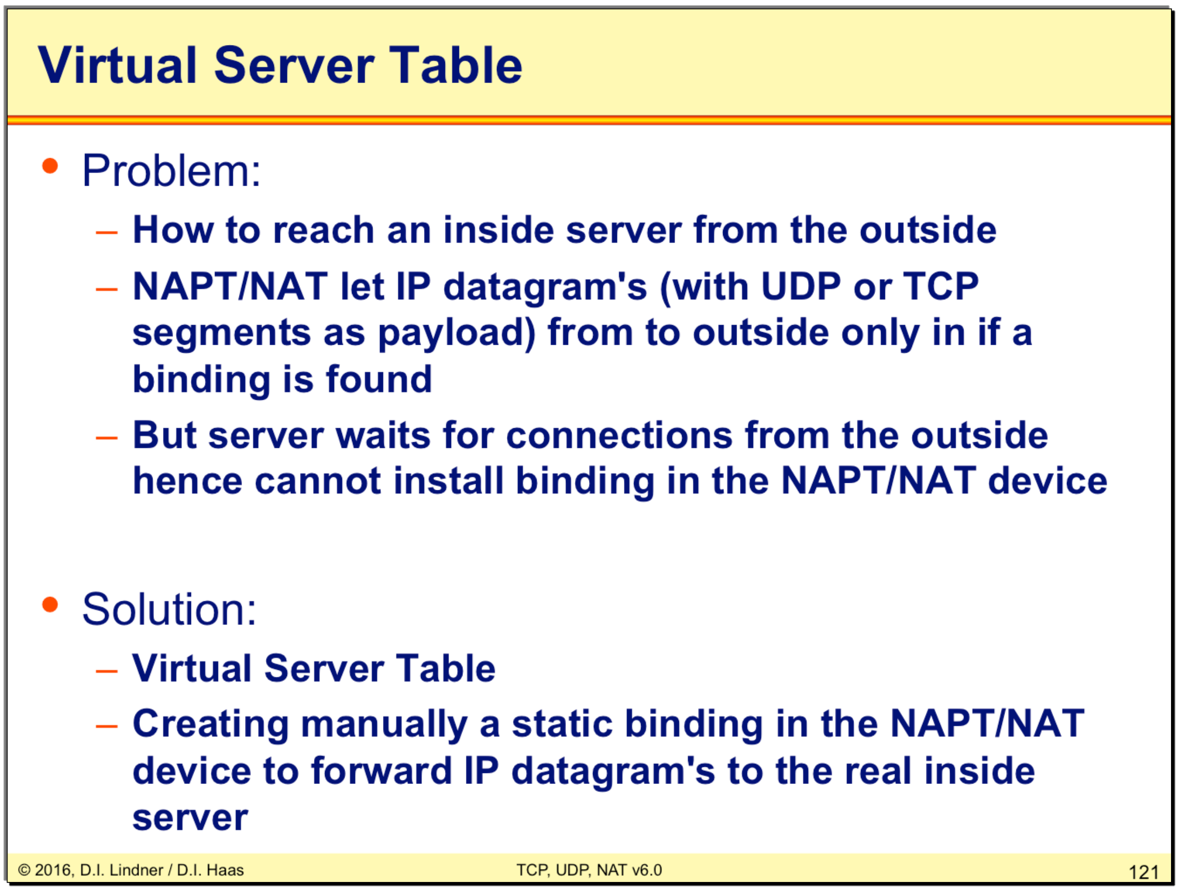
Prot.	Local	Global
TCP	10.1.1.1:1034	173.3.8.1:2137
TCP	10.1.1.2:1034	173.3.8.1:2138

Extended Translation Table

In this example both inside hosts (10.1.1.1 and 10.1.1.2) connect to the same outside webserver. The outside global addresses are identical. The destination port number is used to translate to the corresponding inside host.

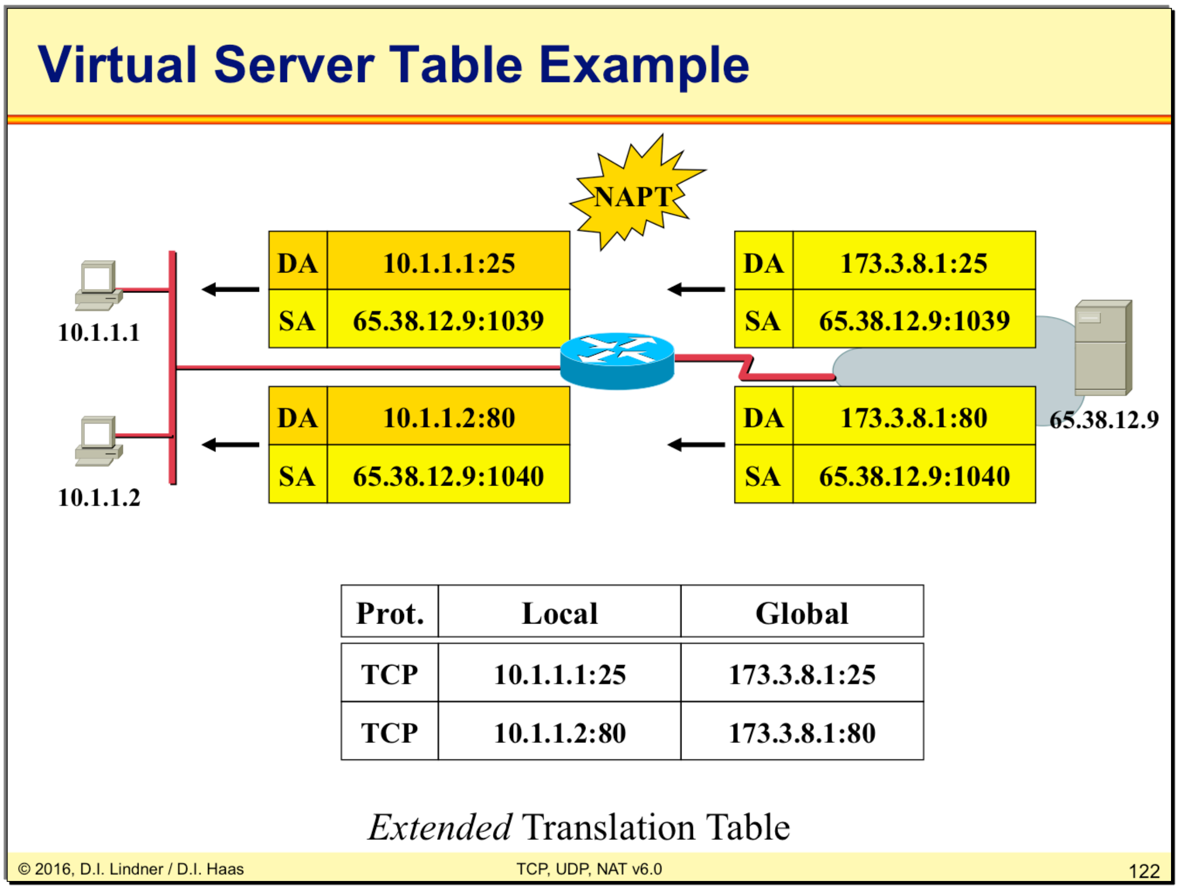
Agenda

- TCP Fundamentals
- TCP Performance
- UDP
- RFC Collection
- NAT
  - NAT Basics
  - NAPT
  - Virtual Server
  - Complex NAT
  - DNS Aspects
  - Load Balancing
  - RFCs

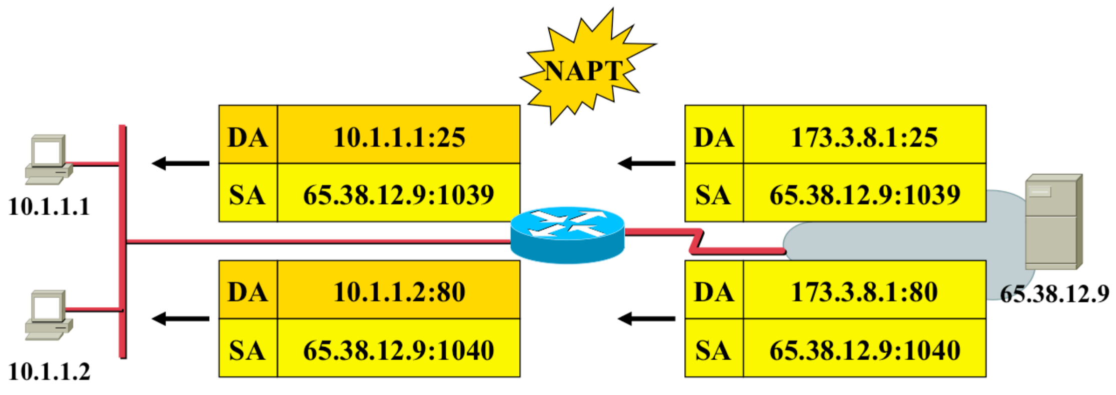


Virtual Server Table

- Problem:
  - **How to reach an inside server from the outside**
  - **NAPT/NAT let IP datagram's (with UDP or TCP segments as payload) from to outside only in if a binding is found**
  - **But server waits for connections from the outside hence cannot install binding in the NAPT/NAT device**
- Solution:
  - **Virtual Server Table**
  - **Creating manually a static binding in the NAPT/NAT device to forward IP datagram's to the real inside server**

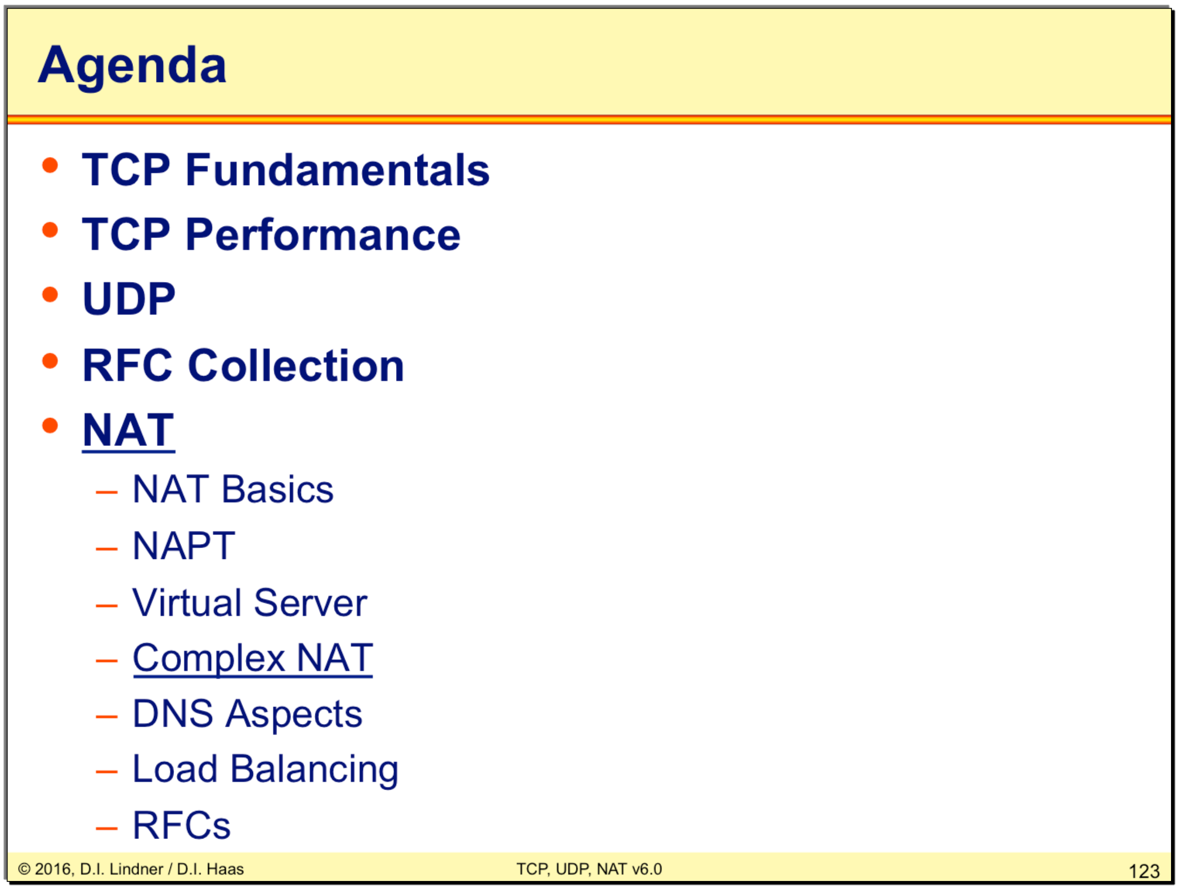


Virtual Server Table Example



Prot.	Local	Global
TCP	10.1.1.1:25	173.3.8.1:25
TCP	10.1.1.2:80	173.3.8.1:80

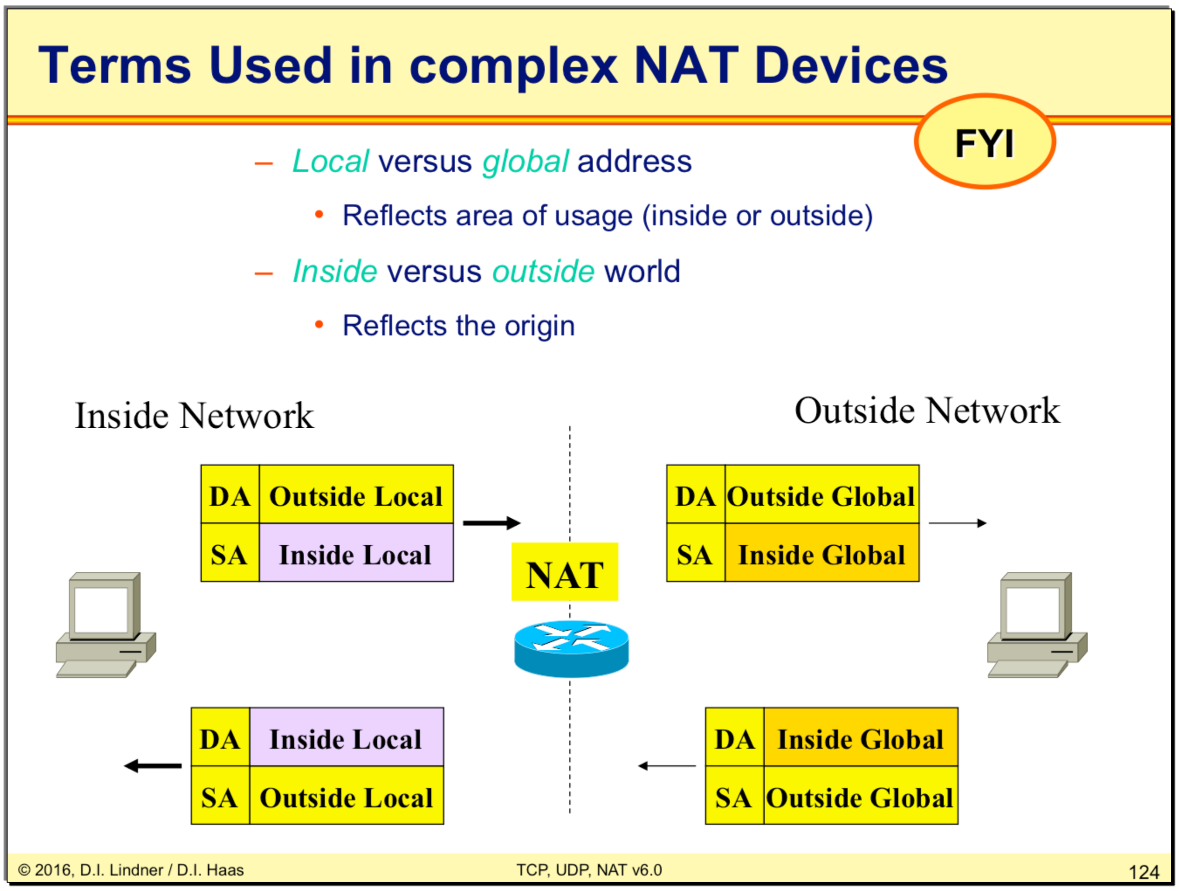
Extended Translation Table



Agenda

- **TCP Fundamentals**
- **TCP Performance**
- **UDP**
- **RFC Collection**
- **NAT**
  - NAT Basics
  - NAPT
  - Virtual Server
  - Complex NAT
  - DNS Aspects
  - Load Balancing
  - RFCs

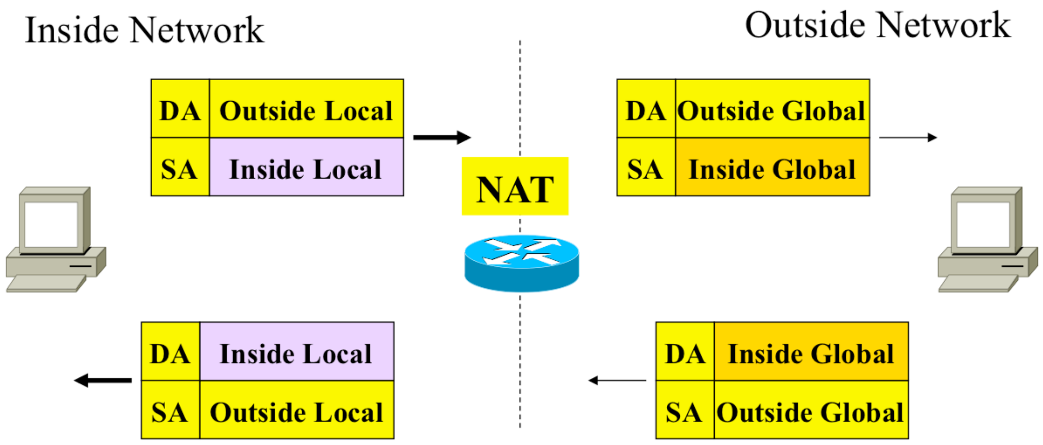
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 123



Terms Used in complex NAT Devices

FYI

- *Local* versus *global* address
  - Reflects area of usage (inside or outside)
- *Inside* versus *outside* world
  - Reflects the origin



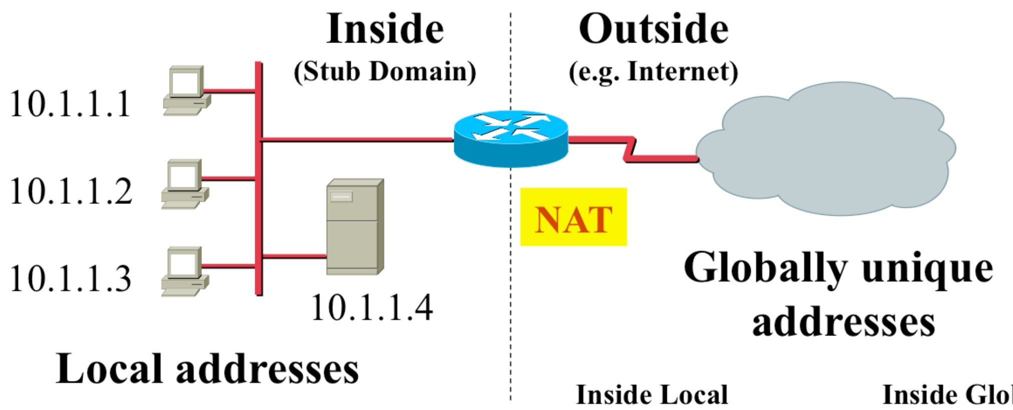
This slide summarizes all terms by showing packets flowing from inside to outside and from outside to inside. Local is what we can use inside our network. Inside local source addresses are always private addresses otherwise we won't use NAT.

Outside local addresses can be either private or registered. Mostly they are registered, but in certain cases we might want to present official registered addresses in incoming packets as being private addresses. See the slide "Outside Address Translation" for this special case. Typically the outside local address is mostly identical with the outside global address.

The inside global address is the official address of our hosts as seen in the Internet. What people mostly expect from NAT is to translate an inside local address to an inside global address. Both addresses belong to a host inside our network.

The outside global address is the official registered IP address of an Internet host. Mostly it is identical with our outside local address we use as destination address for outgoing packets. See the slide "Outside Address Translation" for exceptions.

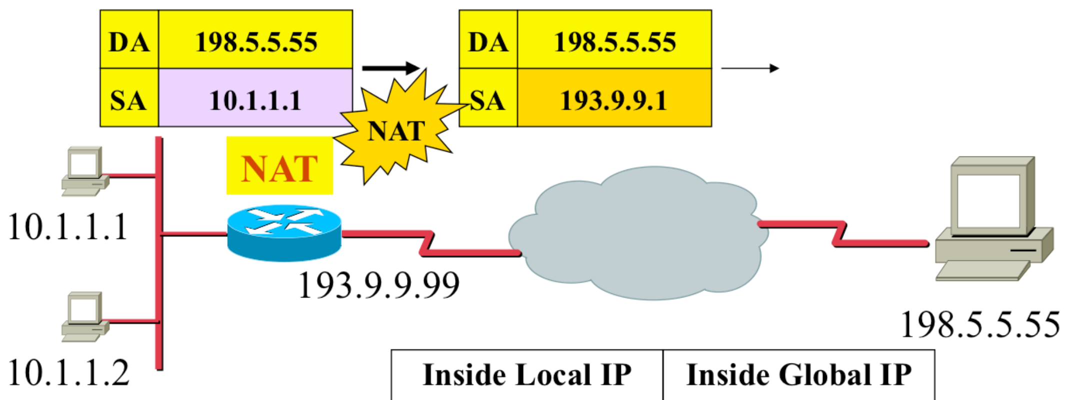
Static NAT Example with New Terms



Binding is maintained by static NAT–Table

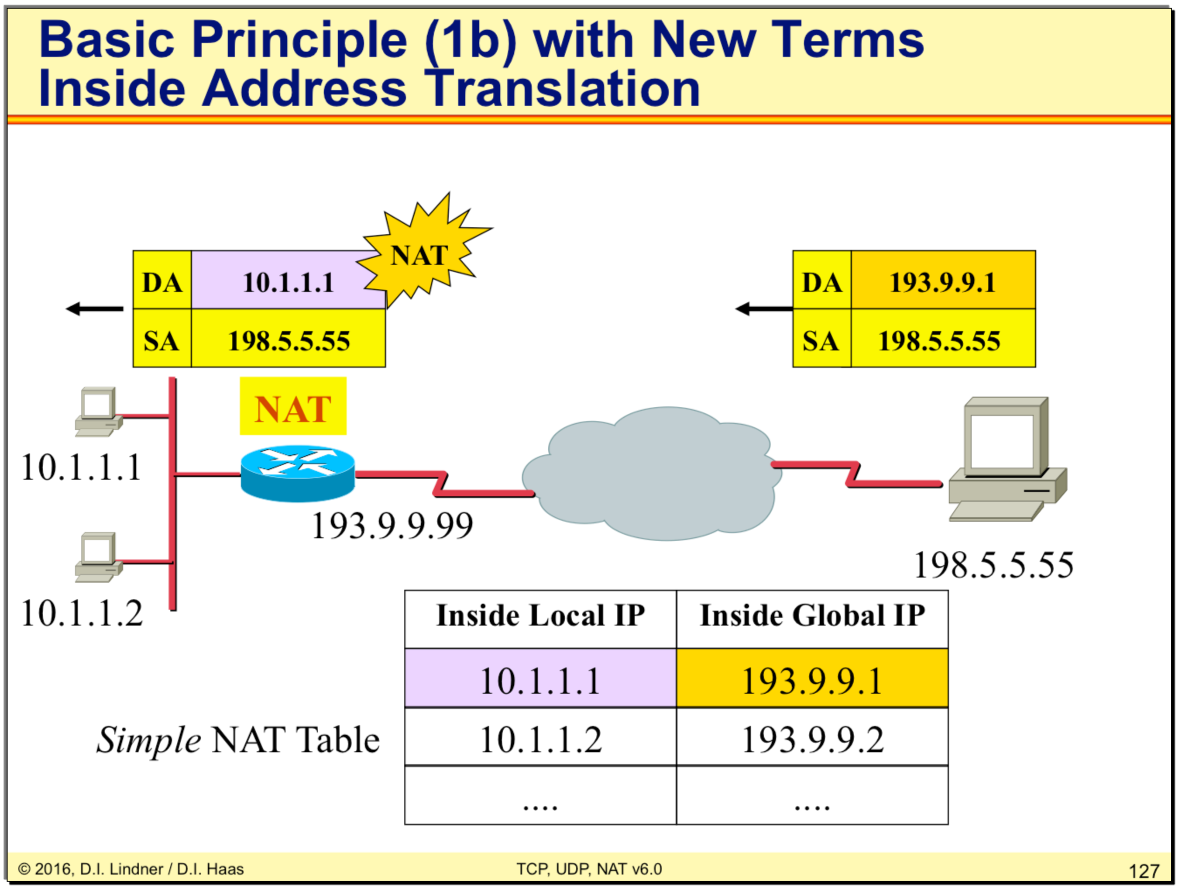
Inside Local IP address		Inside Global IP address
10.1.1.1	⟷	193.99.99.1
10.1.1.2	⟷	193.99.99.2
10.1.1.3	⟷	193.99.99.3
10.1.1.4	⟷	193.99.99.4

Basic Principle (1a) with New Terms Inside Address Translation



Simple NAT Table

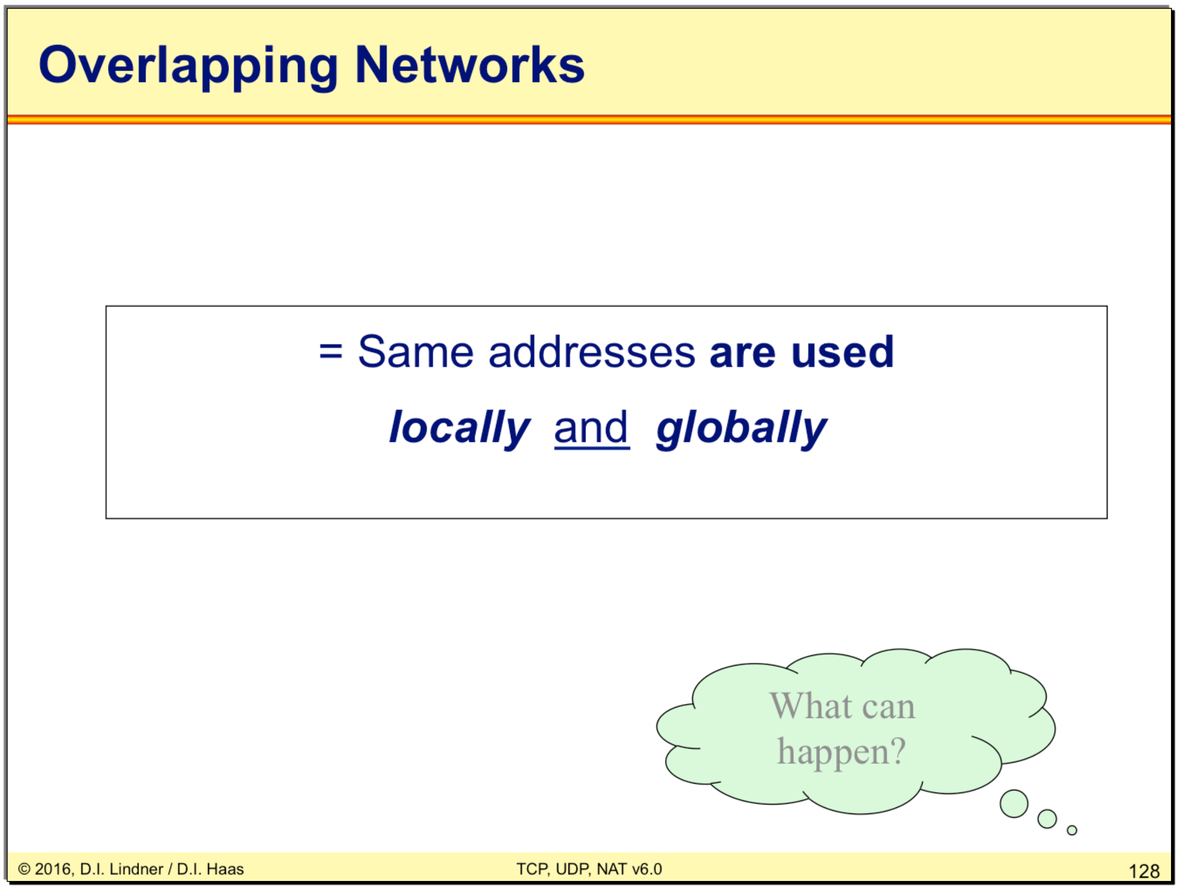
Inside Local IP	Inside Global IP
10.1.1.1	193.9.9.1
10.1.1.2	193.9.9.2
....	....



Basic Principle (1b) with New Terms Inside Address Translation

Simple NAT Table

Inside Local IP	Inside Global IP
10.1.1.1	193.9.9.1
10.1.1.2	193.9.9.2
....	....



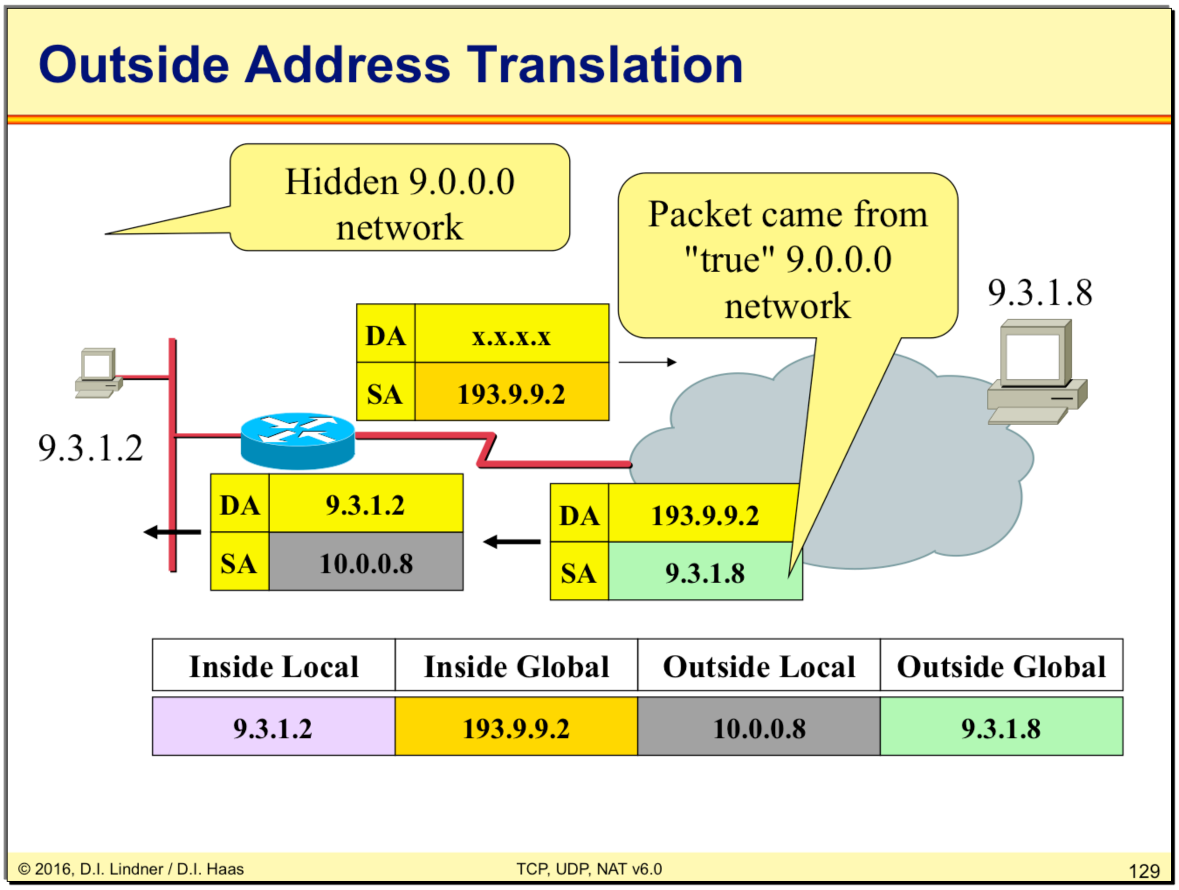
Overlapping Networks

= Same addresses **are used**

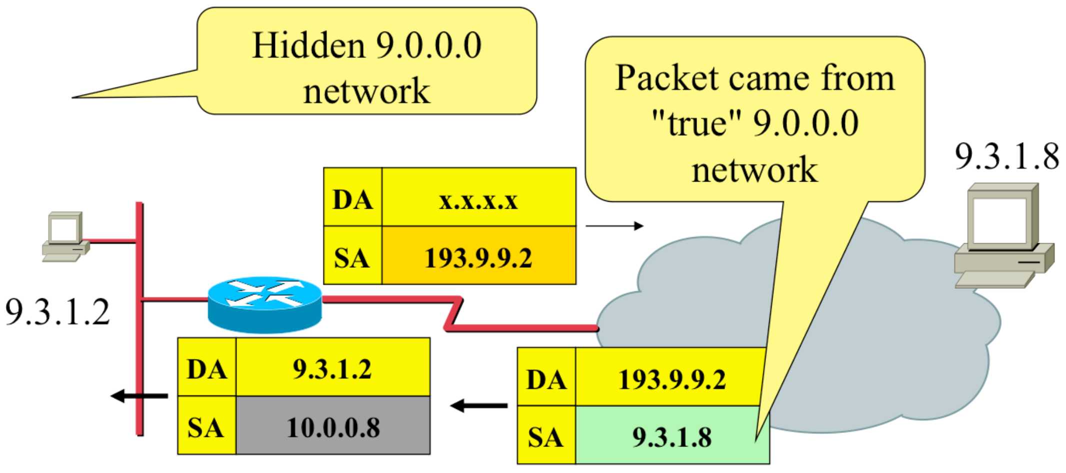
locally and ***globally***

What can happen?

Overlapping networks occur if we use non-legal (not officially assigned) IP addresses that officially belong to another network. We can do that if we use NAT to translate our internal addresses into global ones. However, if we want to communicate with the other network (that use our inside-local addresses as global ones) we must consider some special issues...



Outside Address Translation

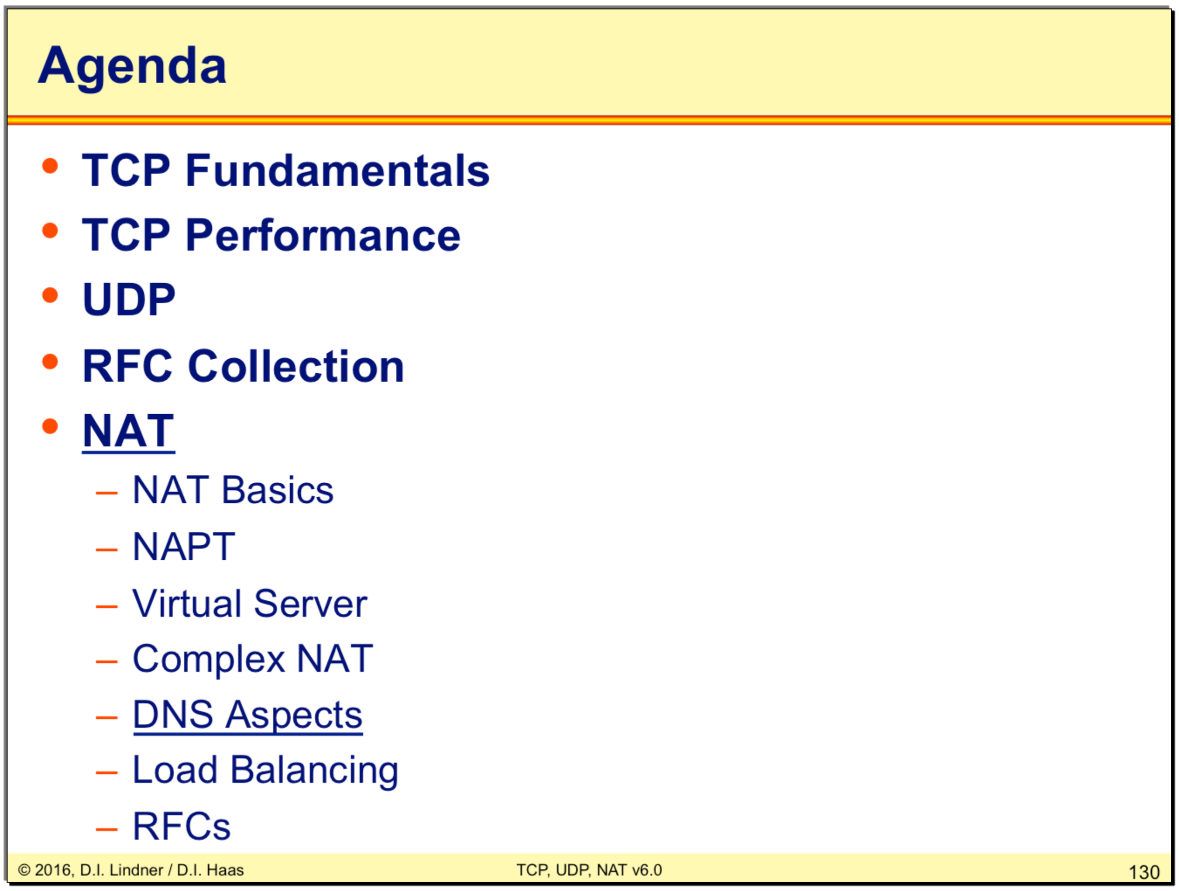


Inside Local	Inside Global	Outside Local	Outside Global
9.3.1.2	193.9.9.2	10.0.0.8	9.3.1.8

First we examine the simple case. Suppose we used a class A network 9.0.0.0 for several years and now we want to give it back to the world (thereby earning a lot of money from our ISP).

Now we will present our network through NAT to the outside world. Obviously the class A range we had given away will be used by other customers, so incoming packets might have the same source addresses as we still use for our devices. Clearly we should renumber our hosts with RFC1918 private addresses.

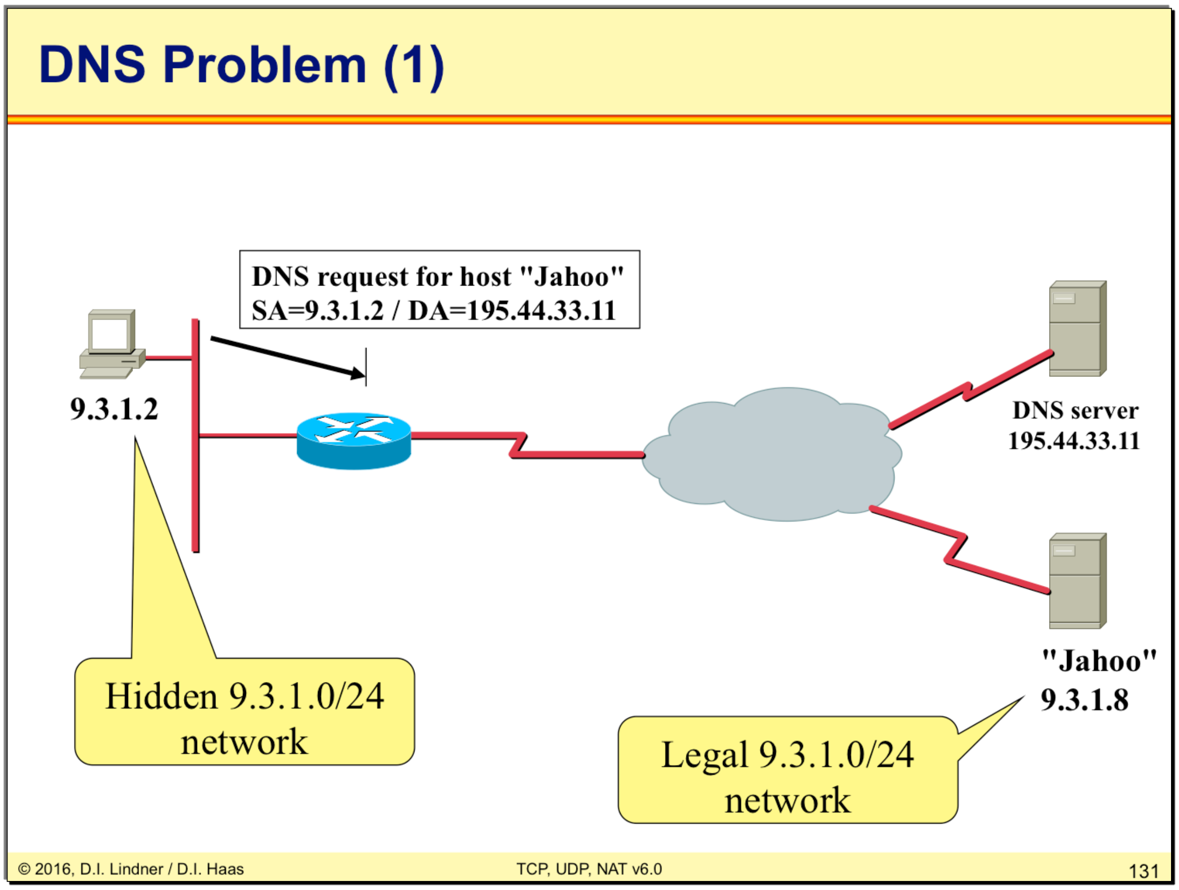
But if we had a big number of hosts we might not want to renumber all devices, instead we will translate the source addresses of incoming packets if they come from the true class-A network 9.0.0.0. By changing to an outside-local address, these packets can be routed outside.



Agenda

- **TCP Fundamentals**
- **TCP Performance**
- **UDP**
- **RFC Collection**
- **NAT**
  - NAT Basics
  - NAPT
  - Virtual Server
  - Complex NAT
  - DNS Aspects
  - Load Balancing
  - RFCs

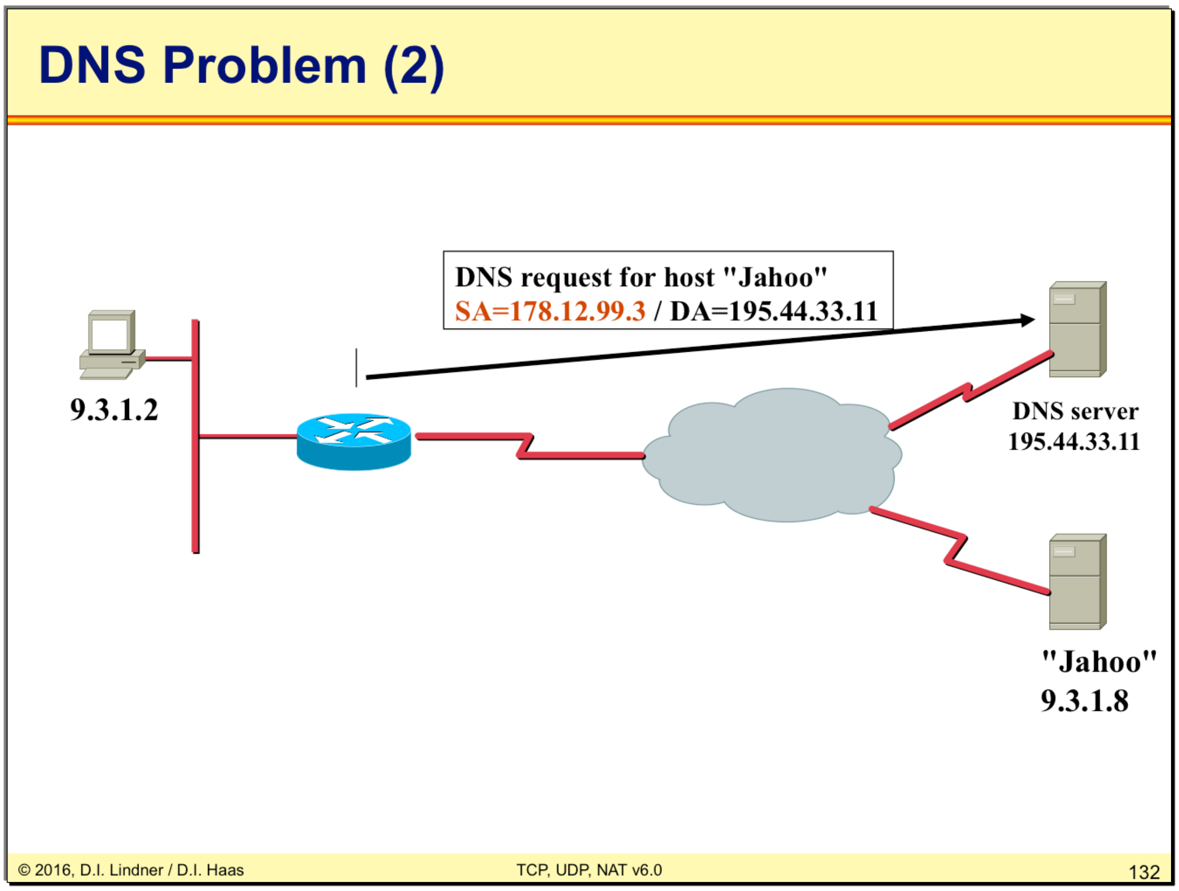
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 130



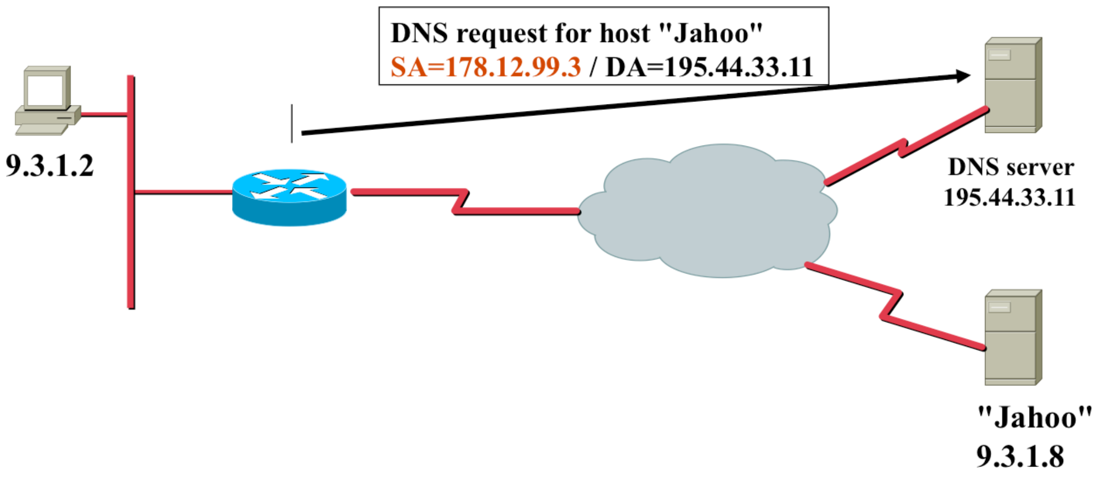
DNS Problem (1)

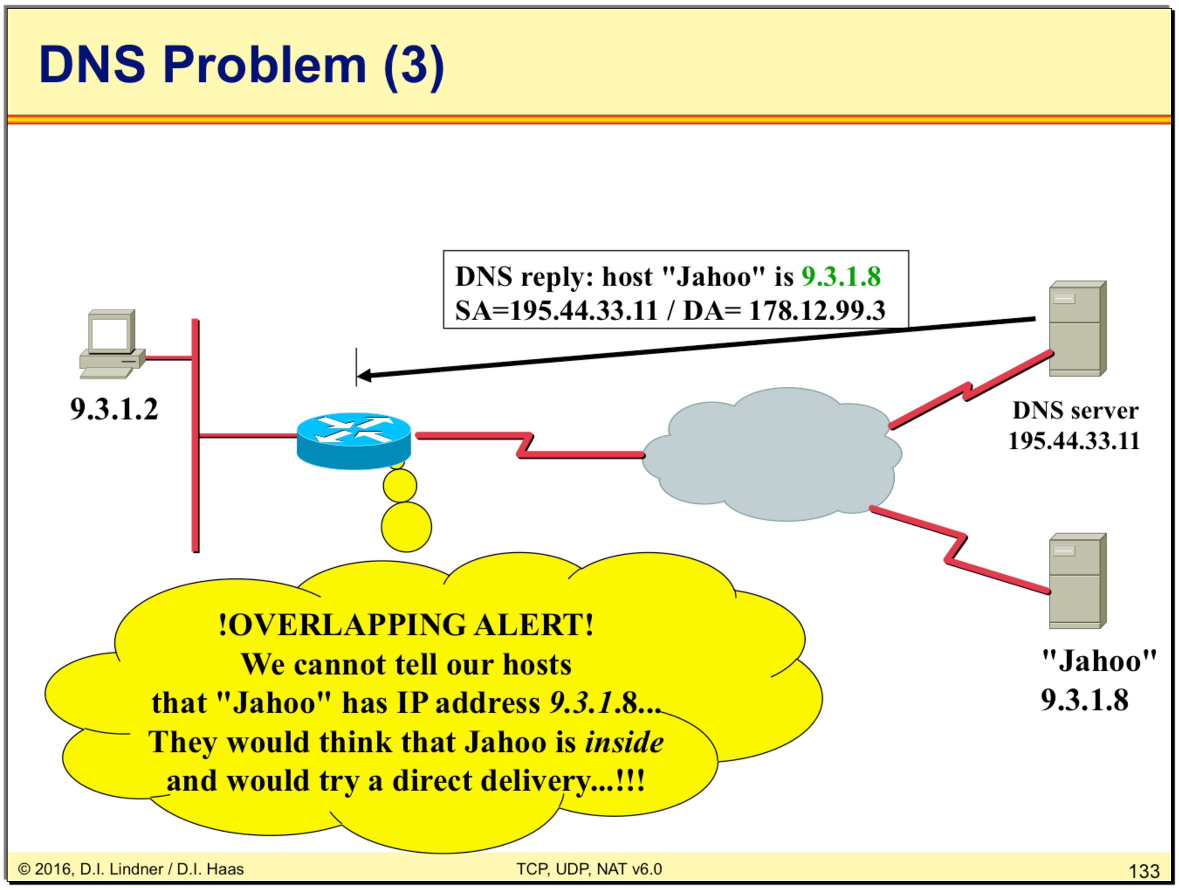
DNS request for host "Jahoo"
SA=9.3.1.2 / DA=195.44.33.11
9.3.1.2
DNS server
195.44.33.11
"Jahoo"
9.3.1.8
Hidden 9.3.1.0/24 network
Legal 9.3.1.0/24 network
© 2016, D.I. Lindner / D.I. Haas
TCP, UDP, NAT v6.0
131

This is a more tricky issue. Usually we do not know IP addresses of outside hosts, rather we ask a DNS server for name resolution.



DNS Problem (2)


DNS request for host "Jahoo"
SA=178.12.99.3 / DA=195.44.33.11
9.3.1.2
DNS server
195.44.33.11
"Jahoo"
9.3.1.8

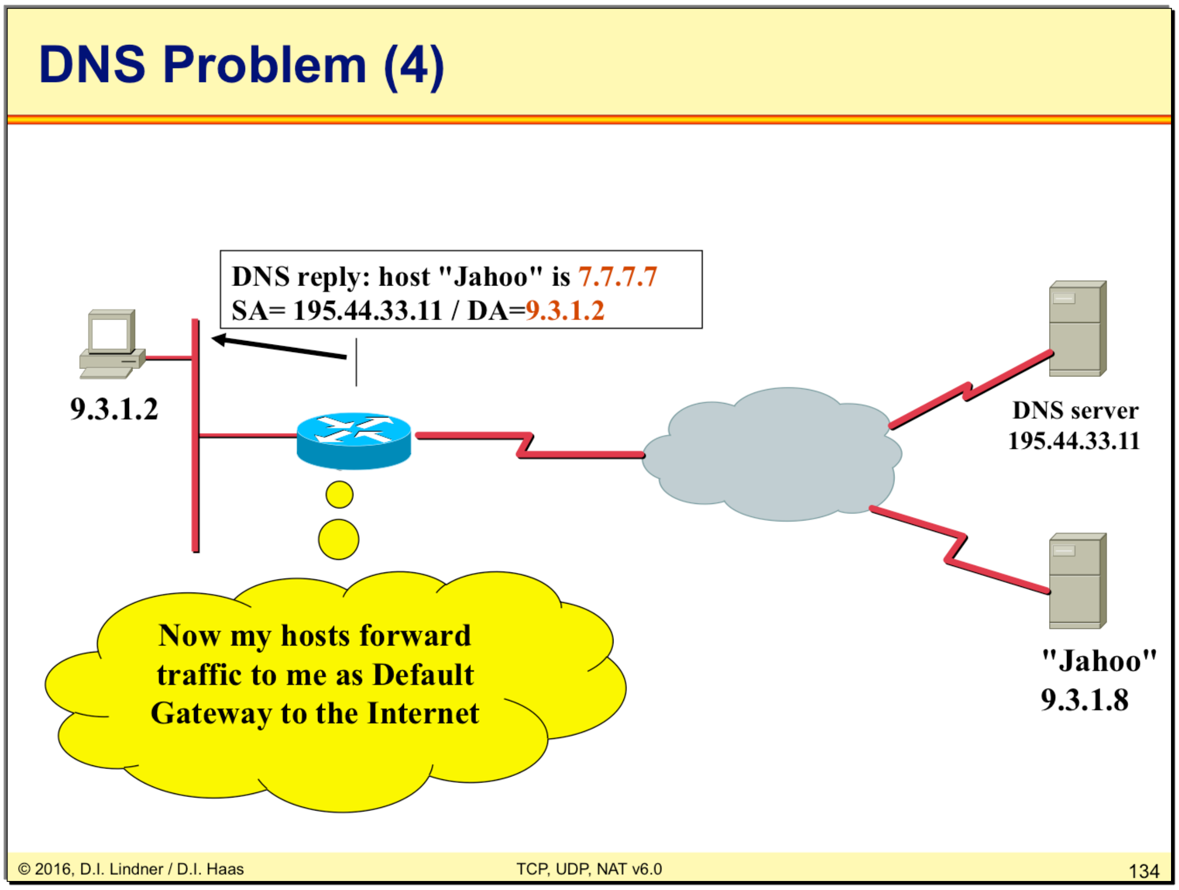


DNS Problem (3)

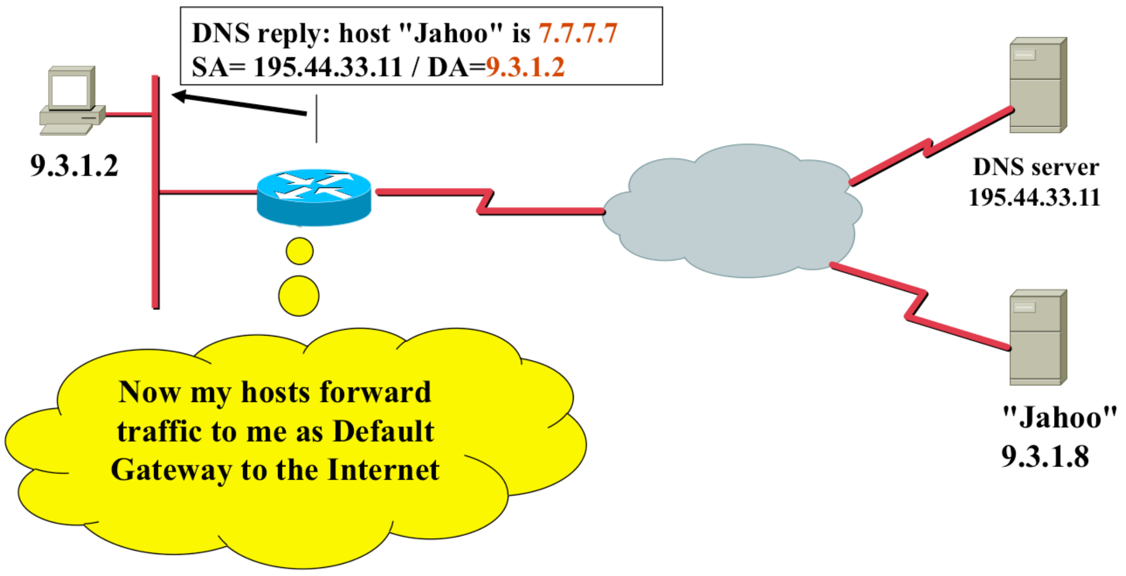
DNS reply: host "Jahoo" is 9.3.1.8
SA=195.44.33.11 / DA= 178.12.99.3
9.3.1.2
DNS server
195.44.33.11
!OVERLAPPING ALERT!
We cannot tell our hosts that "Jahoo" has IP address *9.3.1.8...* They would think that Jahoo is *inside* and would try a direct delivery...!!!
"Jahoo"
9.3.1.8

TCP, UDP, NAT v6.0 133

But what, if the DNS server replies an IP address which is supposed to be inside our own network? In this case the NAT router must manipulate the layer-7 DNS information and translate the global-outside addresses.



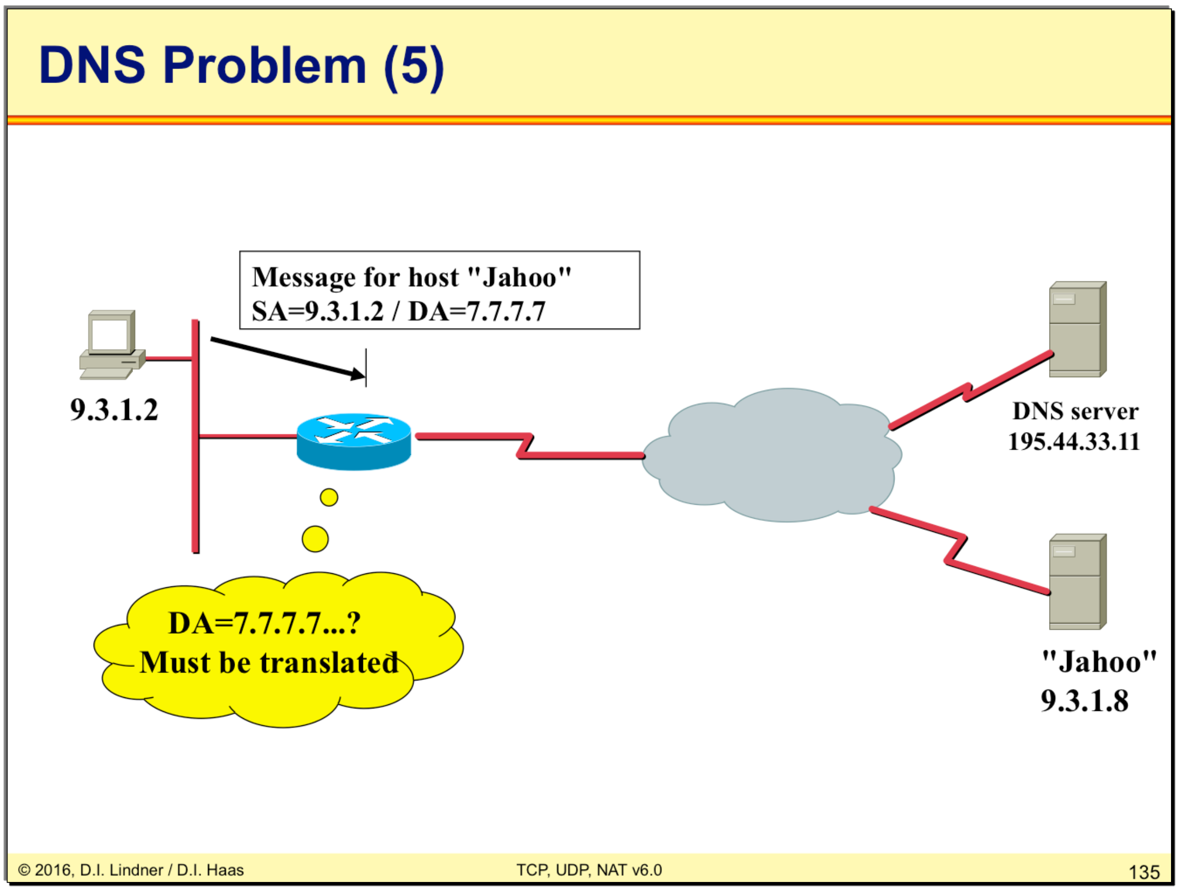
DNS Problem (4)


DNS reply: host "Jahoo" is 7.7.7.7
SA= 195.44.33.11 / DA=9.3.1.2
9.3.1.2
Now my hosts forward traffic to me as Default Gateway to the Internet
DNS server
195.44.33.11
"Jahoo"
9.3.1.8

The router examines every DNS reply, ensuring that the resolved address is not used inside. In such overlapping situations the router will translate the address.

Note:

Cisco NAT is able to inspect and perform address translation on A (Address) and PTR (Pointer) DNS Resource Records.

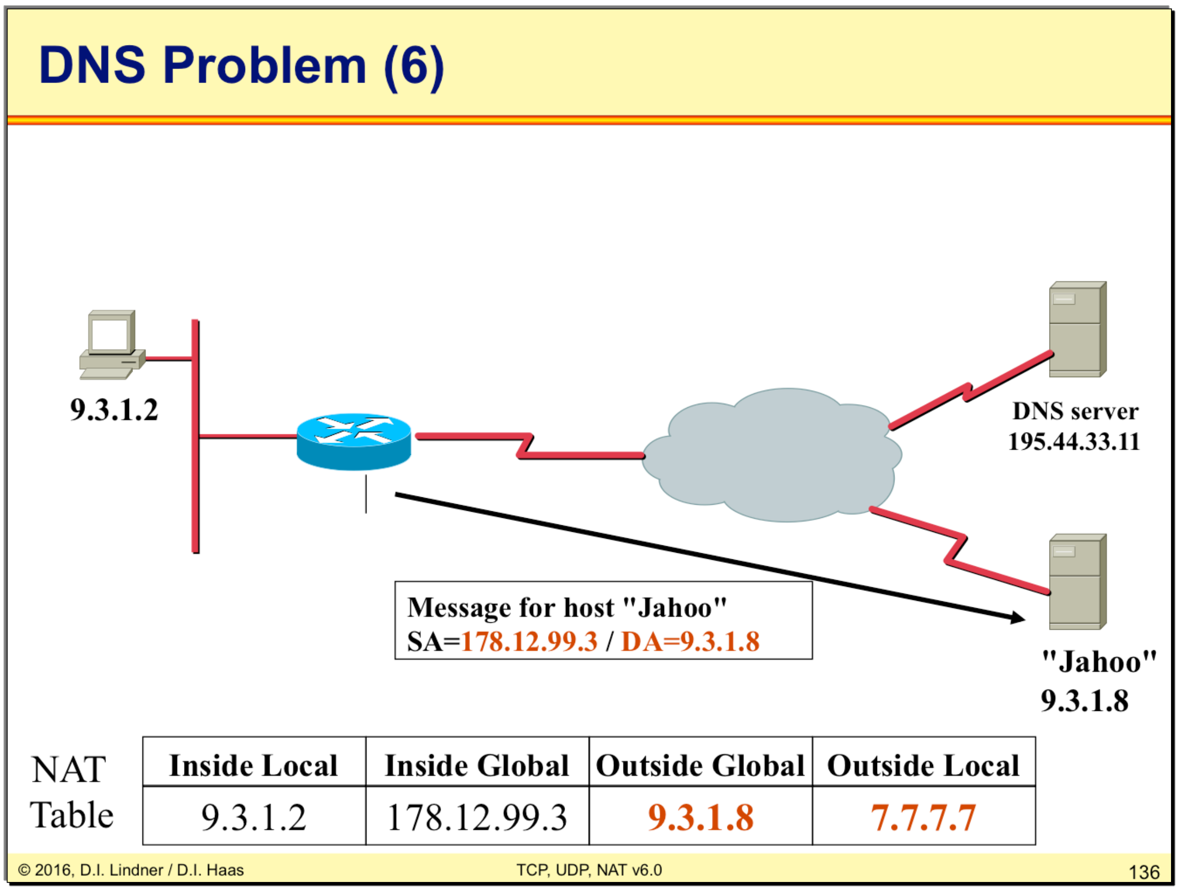


DNS Problem (5)

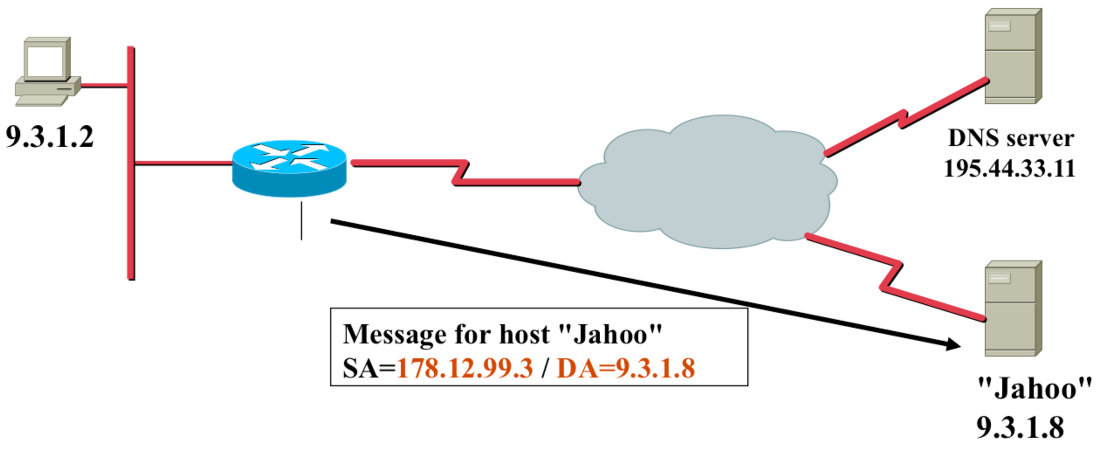
Of course if the destination address of outgoing packets match a previously introduced outside-local address, it must be translated into a outside-global address.

The same performance is done in a converse situation where the DNS server is inside and a DNS request is sent by an outside host. If the name resolution result in an inside local address the NAT router has to translate this address.

NOTE: Cisco IOS does not translate addresses inside DNS zone transfers.

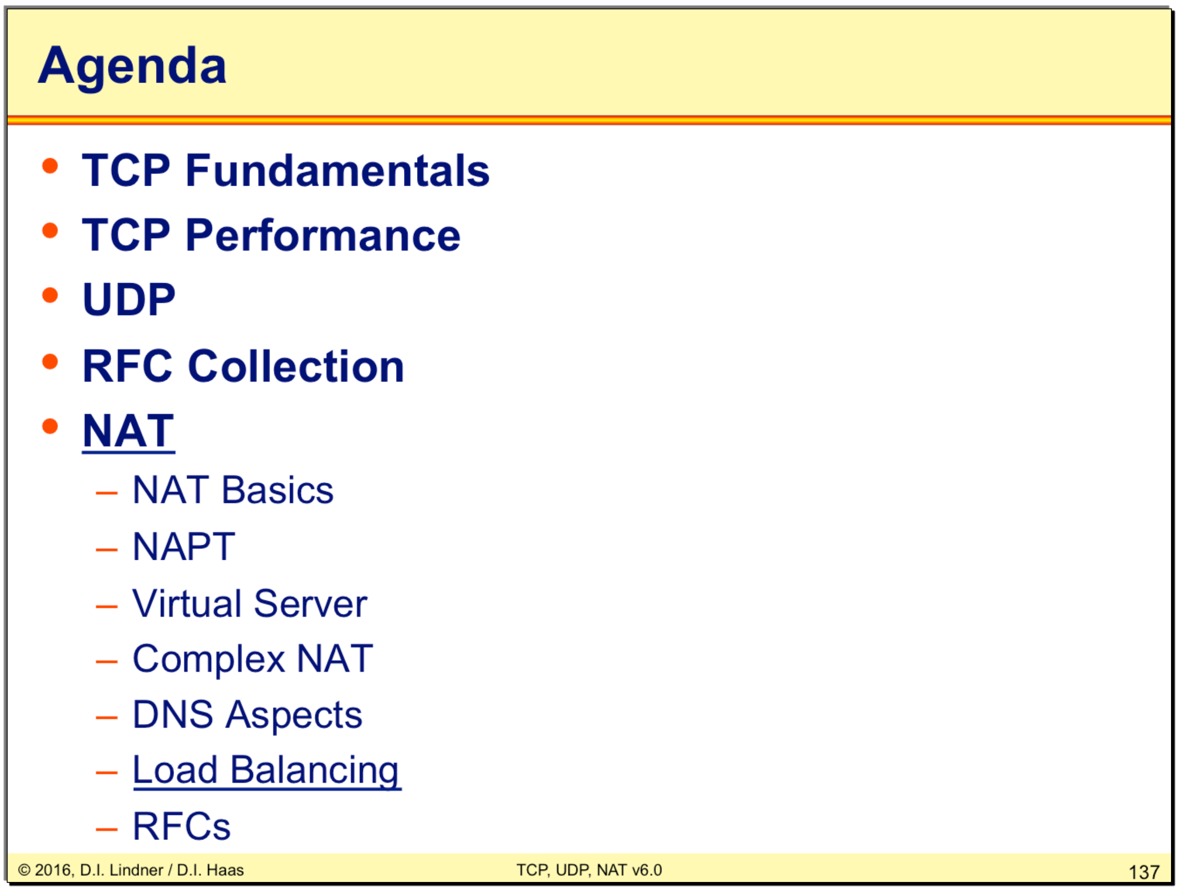


DNS Problem (6)



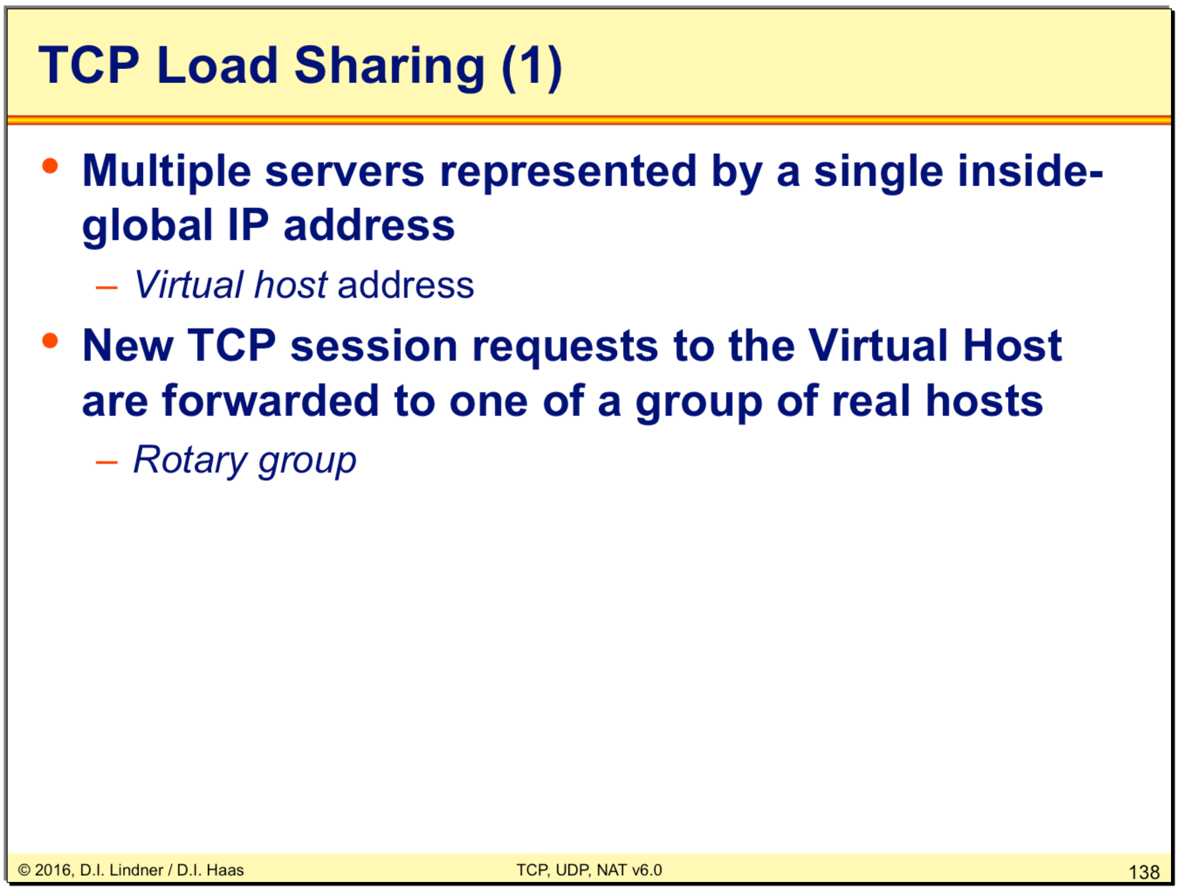
Message for host "Jahoo"
SA=178.12.99.3 / DA=9.3.1.8

NAT Table	Inside Local	Inside Global	Outside Global	Outside Local
	9.3.1.2	178.12.99.3	9.3.1.8	7.7.7.7



Agenda

- TCP Fundamentals
- TCP Performance
- UDP
- RFC Collection
- NAT
  - NAT Basics
  - NAPT
  - Virtual Server
  - Complex NAT
  - DNS Aspects
  - Load Balancing
  - RFCs



TCP Load Sharing (1)

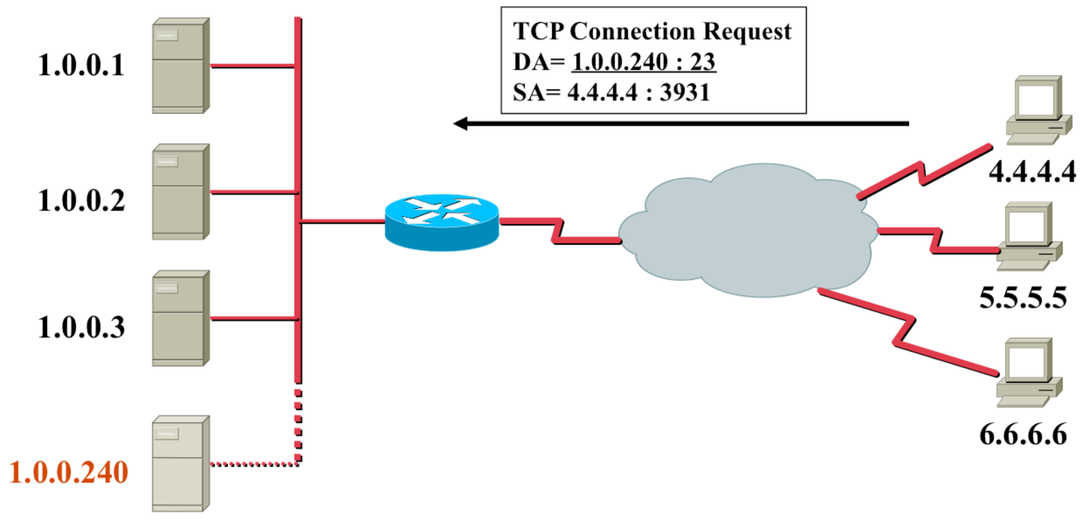
- **Multiple servers represented by a single inside-global IP address**
  - *Virtual host* address
- **New TCP session requests to the Virtual Host are forwarded to one of a group of real hosts**
  - *Rotary group*

TCP, UDP, NAT v6.0 138

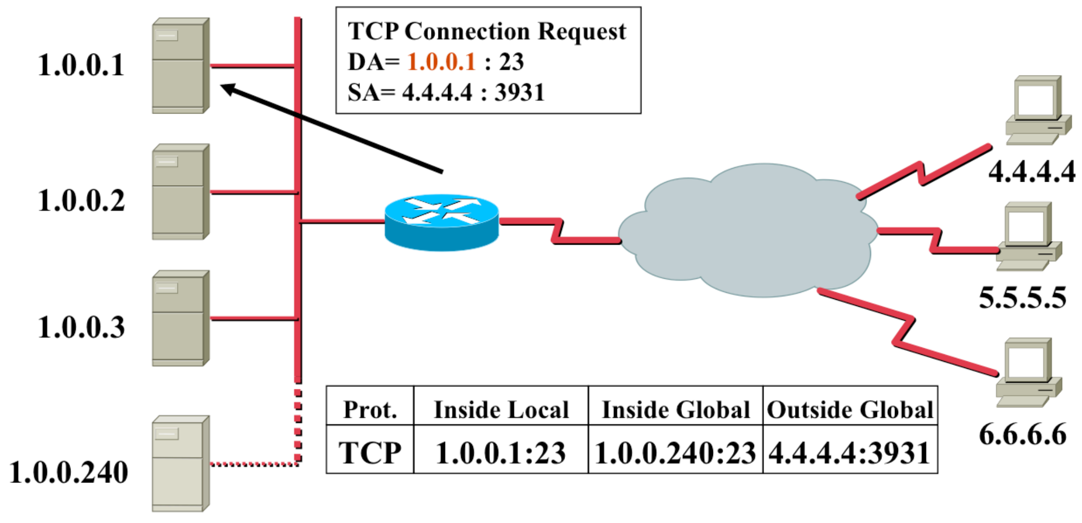
TCP load sharing is an enhanced NAT feature and is used inside the Intranet because this has nothing to do with private address translation. If we want to offer a highly loaded specific service to users, we can employ a NAT router to map a single inside-global address (the virtual host address which is known to the users) to multiple inside-local addresses, each assigned to a real host. Everytime a user connects to the virtual host and wants to establish a session, this session is mapped to one of the real hosts in a round-robin manner. That is why the group of real hosts is called "rotary group".

Note that the NAT router has no idea of the load distribution. Neither the service availability is known to the router!

TCP Load Sharing (2)

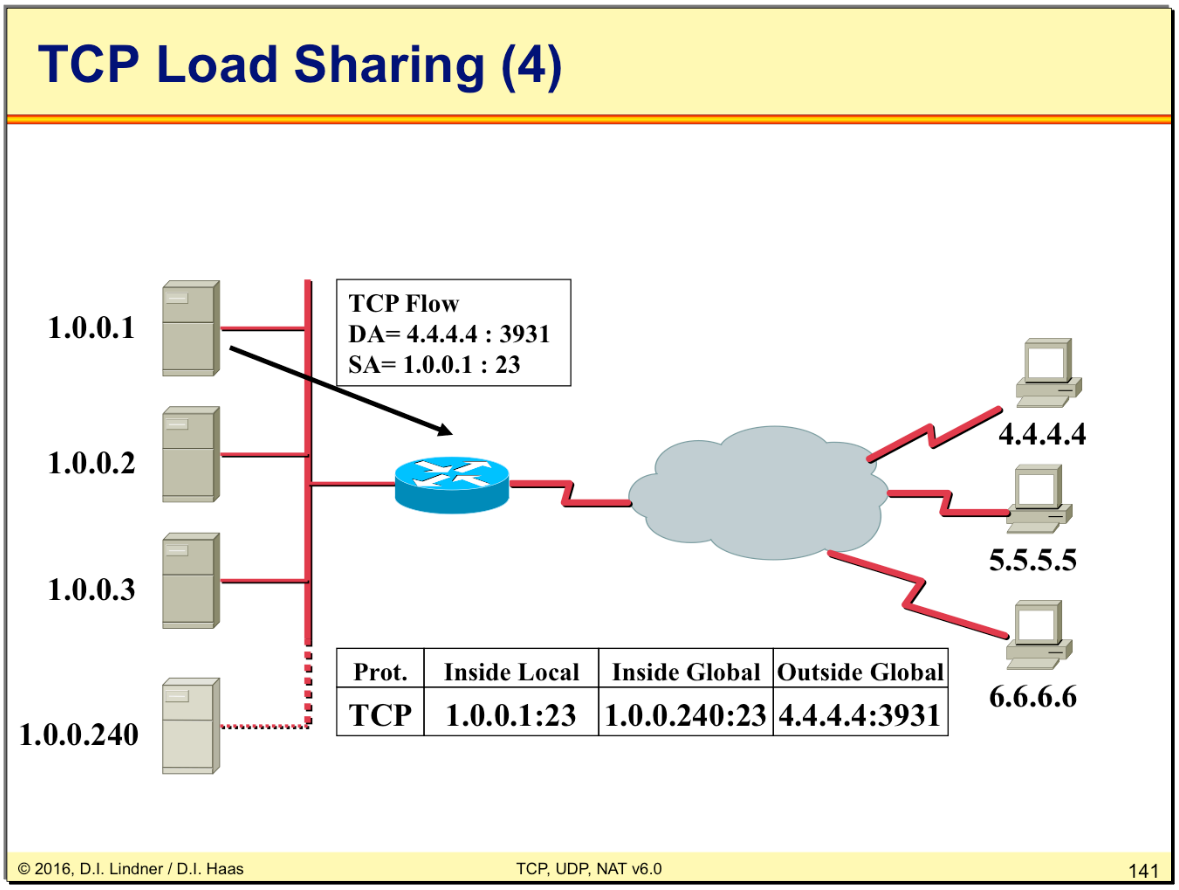


TCP Load Sharing (3)

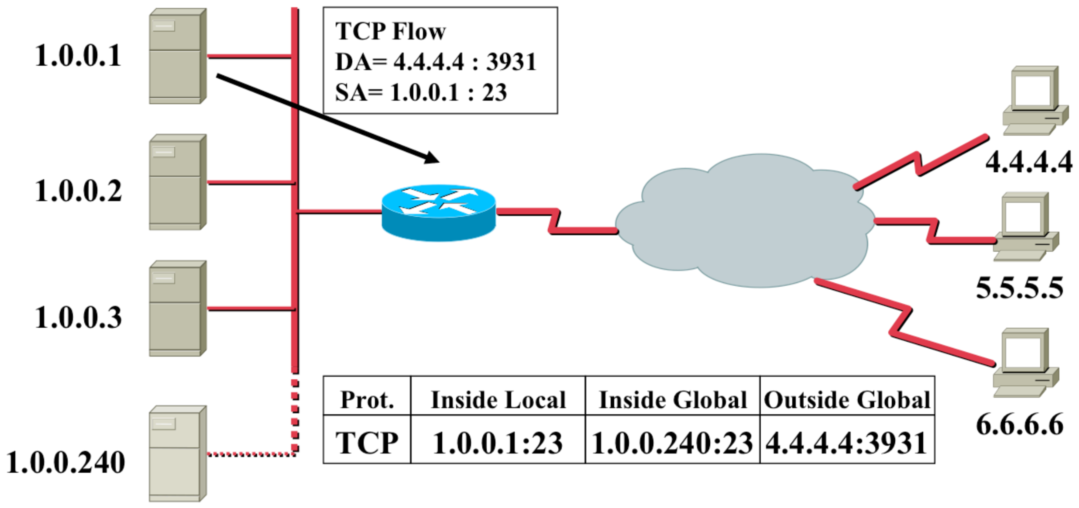


TCP Connection Request
DA= 1.0.0.1 : 23
SA= 4.4.4.4 : 3931

Prot.	Inside Local	Inside Global	Outside Global
TCP	1.0.0.1:23	1.0.0.240:23	4.4.4.4:3931

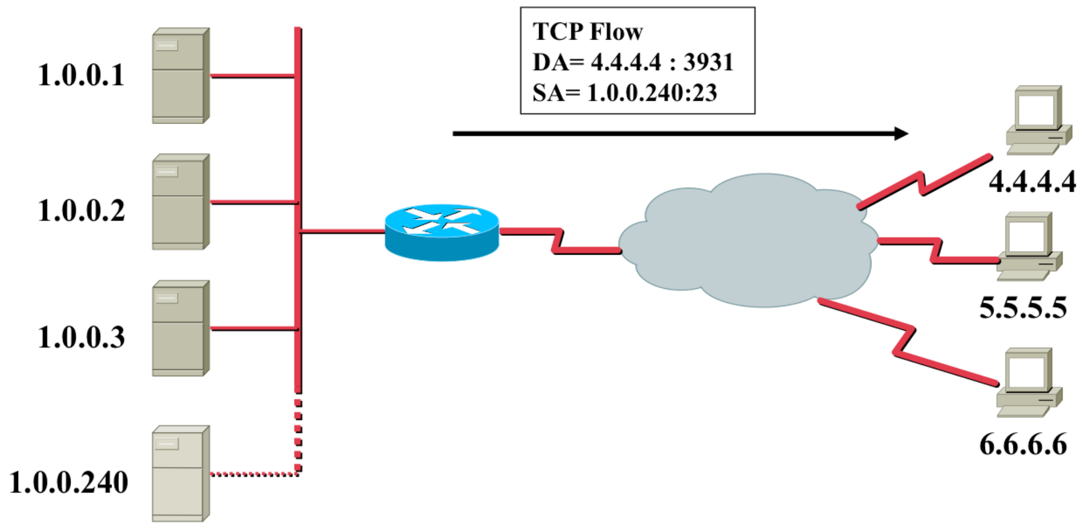


TCP Load Sharing (4)

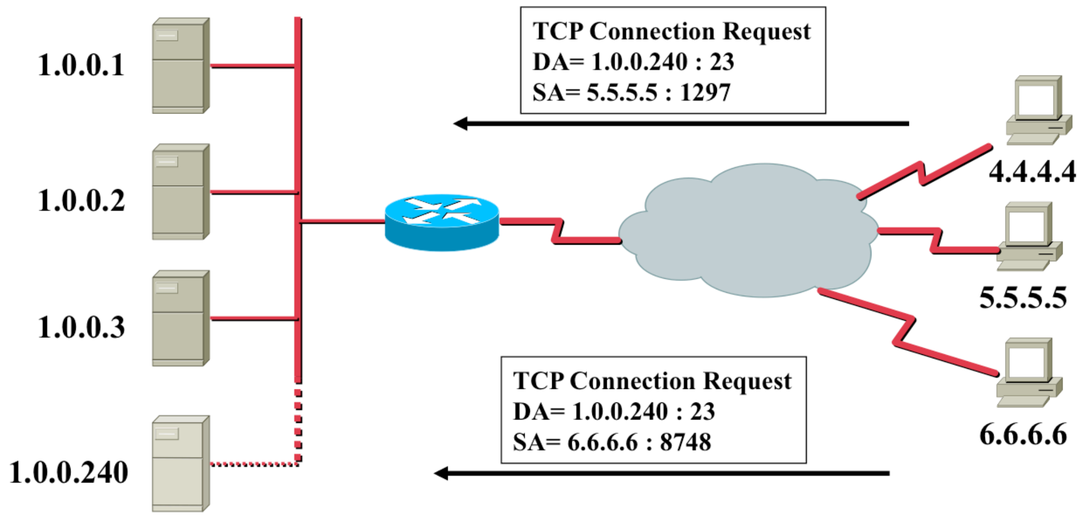


Prot.	Inside Local	Inside Global	Outside Global
TCP	1.0.0.1:23	1.0.0.240:23	4.4.4.4:3931

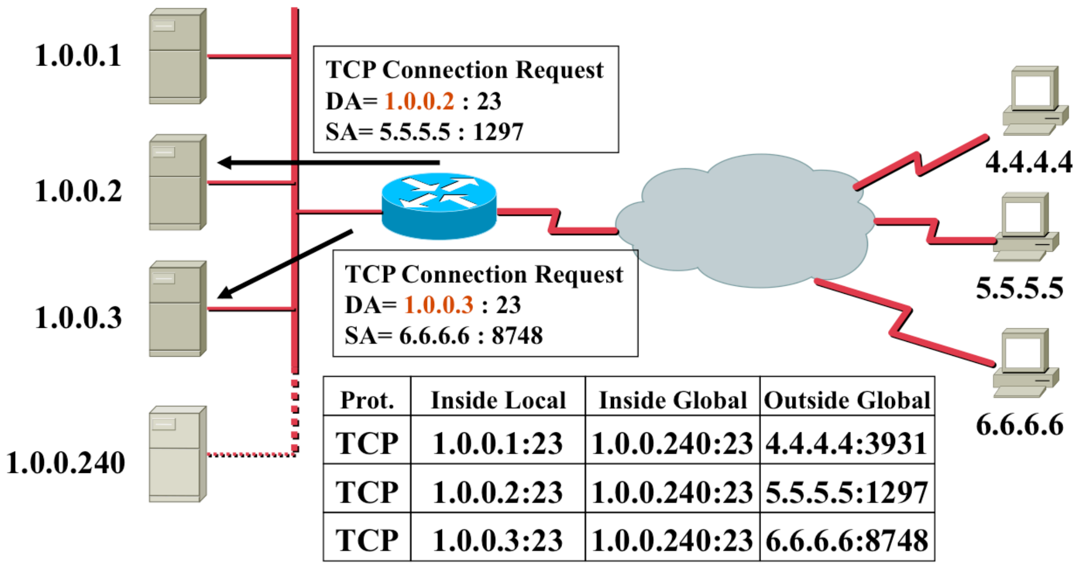
TCP Load Sharing (5)



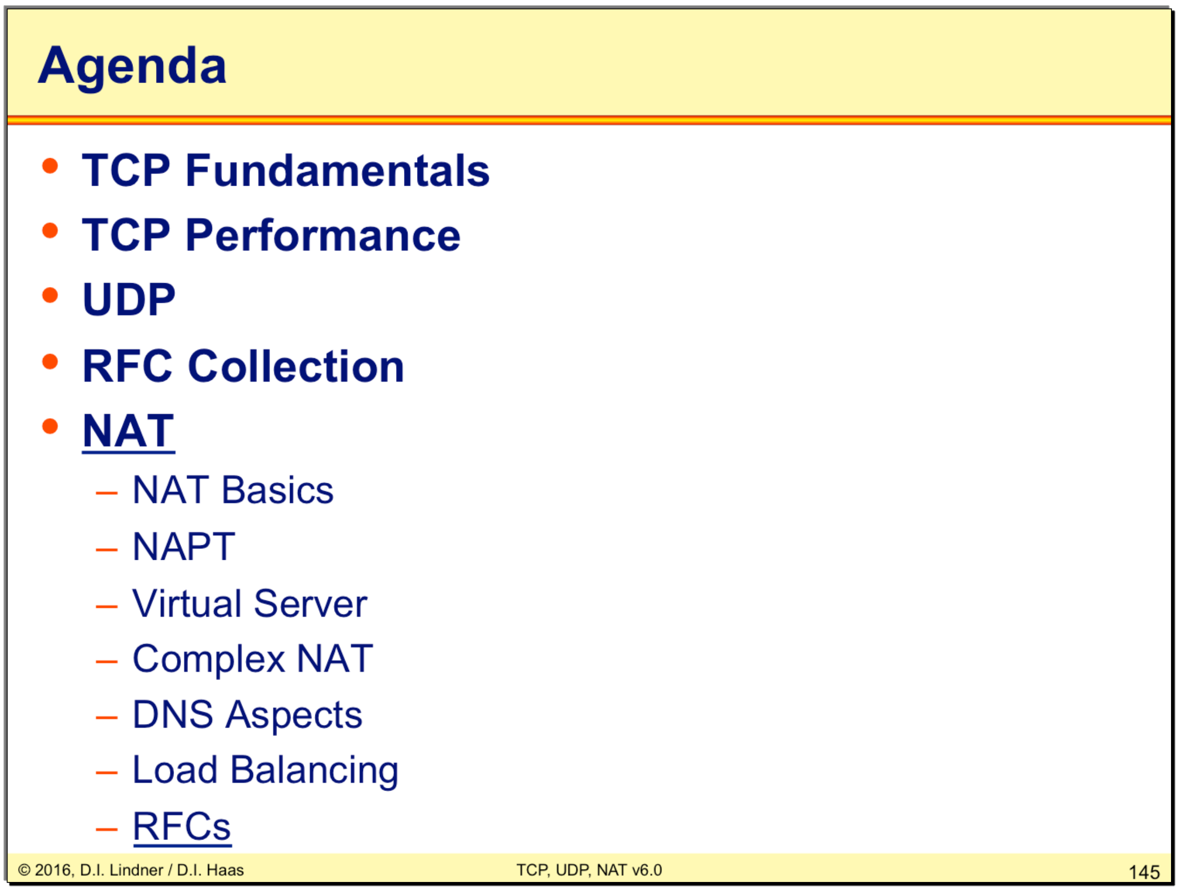
TCP Load Sharing (6)



TCP Load Sharing (7)



Prot.	Inside Local	Inside Global	Outside Global
TCP	1.0.0.1:23	1.0.0.240:23	4.4.4.4:3931
TCP	1.0.0.2:23	1.0.0.240:23	5.5.5.5:1297
TCP	1.0.0.3:23	1.0.0.240:23	6.6.6.6:8748



Agenda

- TCP Fundamentals
- TCP Performance
- UDP
- RFC Collection
- NAT
  - NAT Basics
  - NAPT
  - Virtual Server
  - Complex NAT
  - DNS Aspects
  - Load Balancing
  - RFCs

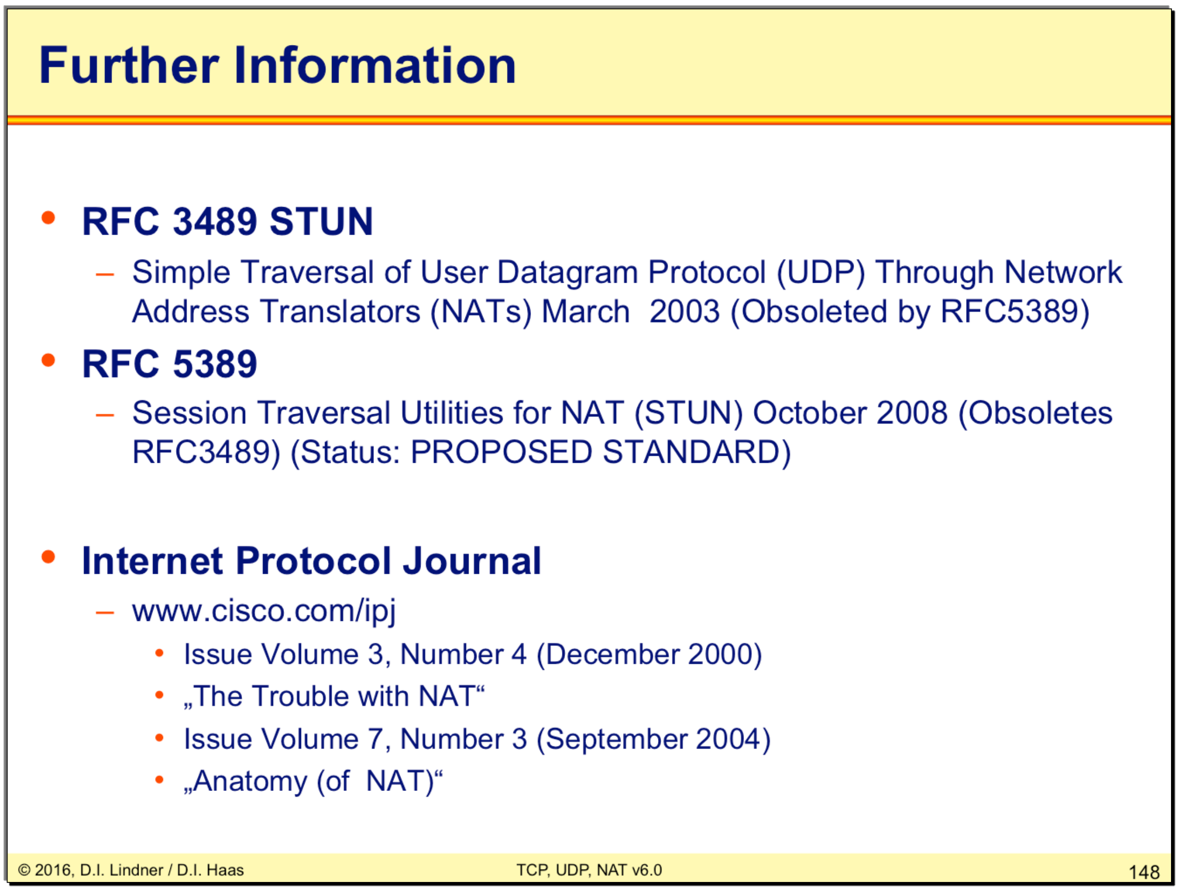
© 2016, D.I. Lindner / D.I. Haas TCP, UDP, NAT v6.0 145

Further Information

- **RFC 1631**
  - NAT
- **RFC 2391**
  - Load Sharing Using IP Network Address Translation (LSNAT)
- **RFC 2666**
  - IP Network Address Translator (NAT) Terminology and Considerations
- **RFC 2694**
  - DNS ALG
- **RFC 2776**
  - Network Address Translation Protocol Translation (NAT-PT)
- **RFC 2993**
  - Architectural Implications of NAT
- **RFC 3022**
  - Traditional IP Network Address Translator (Traditional NAT)

Further Information

- **RFC 3027**
  - Protocol Complications with the IP Network Address Translator,
- **RFC 3235**
  - Network Address Translator (NAT)-Friendly Application Design Guidelines
- **RFC3303**
  - Middlebox Communication Architecture and Framework
- **RFC 3424**
  - IAB Considerations for Unilateral Self Address Fixing (UNSAF) Across Network Address Translation
- **RFC 3715**
  - IPsec—Network Address Translation (NAT) Compatibility Requirements



Further Information

- **RFC 3489 STUN**
  - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs) March 2003 (Obsoleted by RFC5389)
- **RFC 5389**
  - Session Traversal Utilities for NAT (STUN) October 2008 (Obsoletes RFC3489) (Status: PROPOSED STANDARD)
- **Internet Protocol Journal**
  - www.cisco.com/ipj
    - Issue Volume 3, Number 4 (December 2000)
    - „The Trouble with NAT“
    - Issue Volume 7, Number 3 (September 2004)
    - „Anatomy (of NAT)“