TCP - Transmission Control Protocol – RFC793

indigoo.com

- Contents

1. Transport layer and sockets
2. TCP (RFC793) overview
3. Transport Service Access Point (TSAP) addresses
4. TCP Connection Establishment
5. TCP Connection Release
6. TCP Flow Control
7. TCP Error Control
8. TCP Congestion Control RFC2001
9. TCP Persist Timer
10. TCP Keepalive Timer – TCP connection supervision
11. TCP Header Flags
12. TCP Header Options
13. TCP Throughput / performance considerations
14. A last word on „guaranteed delivery“

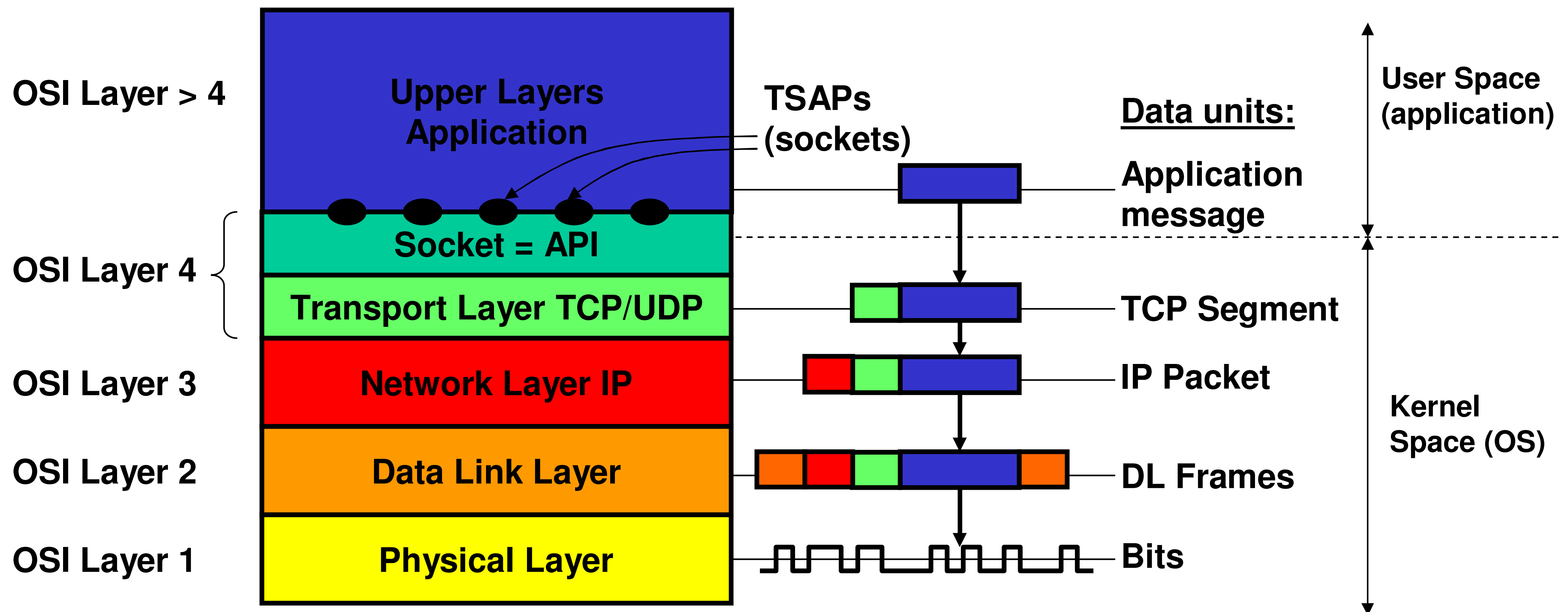
© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

indigoo.com

• **Transport layer and sockets:**

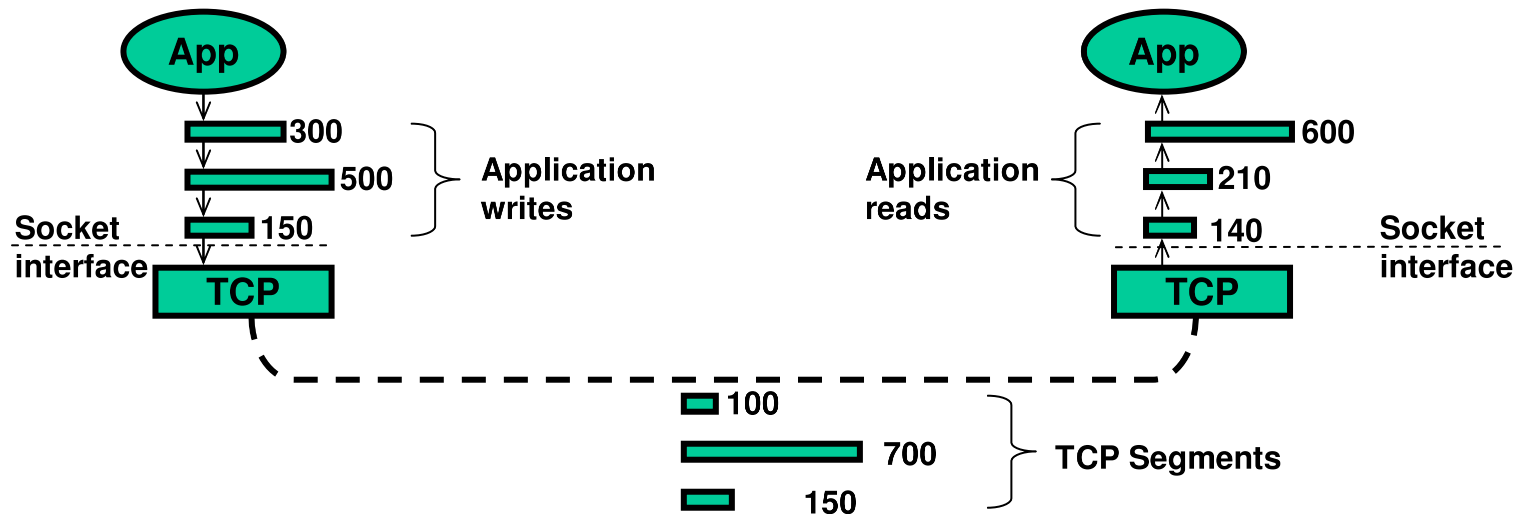
➔ **The transport layer is made accessible to applications through the socket layer (API). The transport layer runs in kernel space (Operating System) while application processes run in user space.**



© Peter R. Egli 2011

• **TCP (RFC793) overview:**

➔ **TCP is a byte stream oriented transmission protocol:**



N.B.: The size of application data chunks (data units passed over socket interface) may be different on the sending and receiving side; the segments sent by TCP may again have different sizes.

➔ **TCP error control provides reliable transmission (packet order preservation, retransmissions in case of transmission errors and packet loss).**

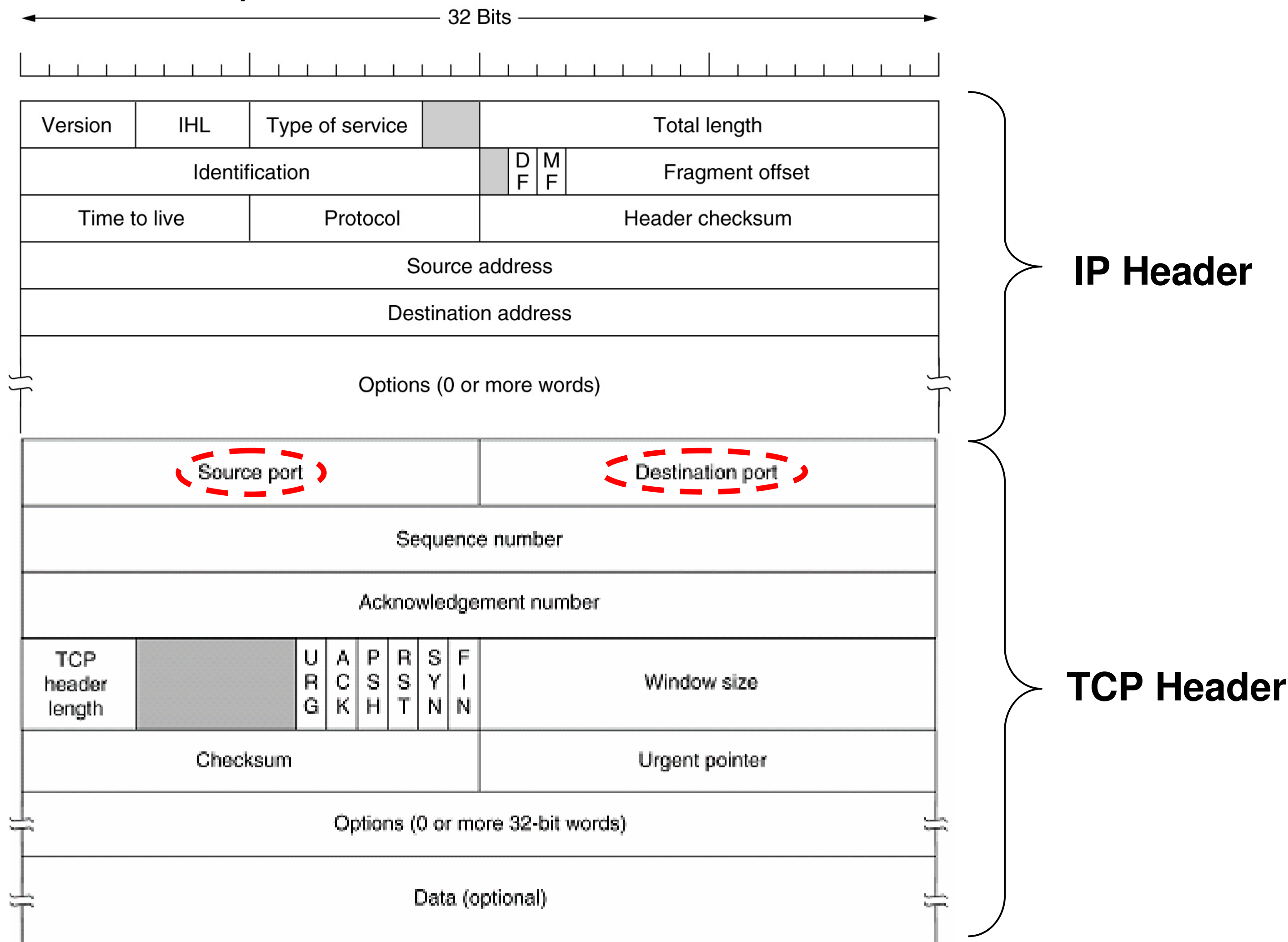
➔ **TCP uses flow control to maximize throughput and avoid packet loss.**

➔ **Congestion control mechanisms allow TCP to react and recover from network congestion.**

TCP - Transmission Control Protocol – RFC793

- **Transport Service Access Point (TSAP) addresses (1/3):**

➔ TSAPs are identified by 16bit port numbers. 65536 port numbers are available (0...65535, but 0 is never used).



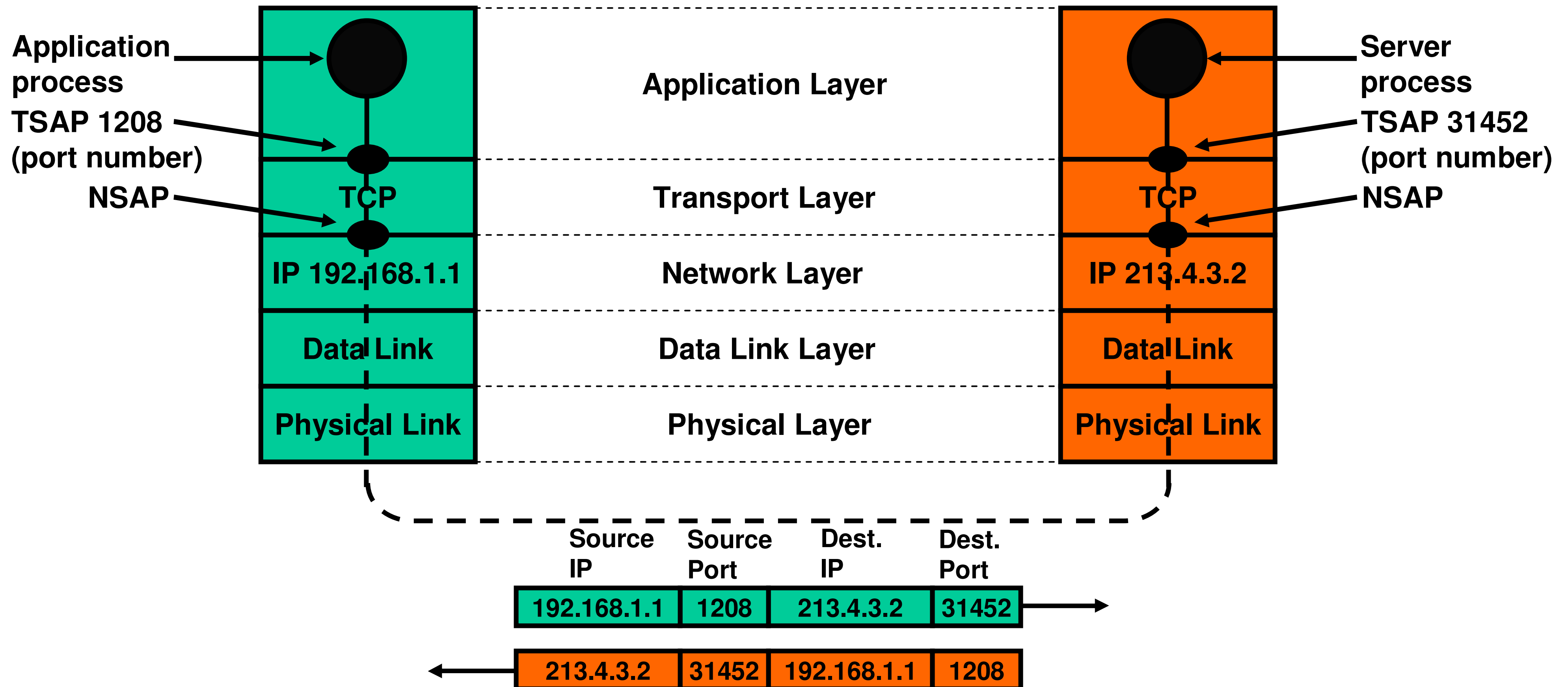
© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

indigoo.com

• **Transport Service Access Point (TSAP) addresses (2/3):**

➔ **Each TSAP (TCP port) is bound to at most 1 socket, i.e. a TCP port can not be opened multiple times.**



© Peter R. Egli 2011

• Transport Service Access Point (TSAP) addresses (3/3):

➔ There are different definitions for the port ranges:

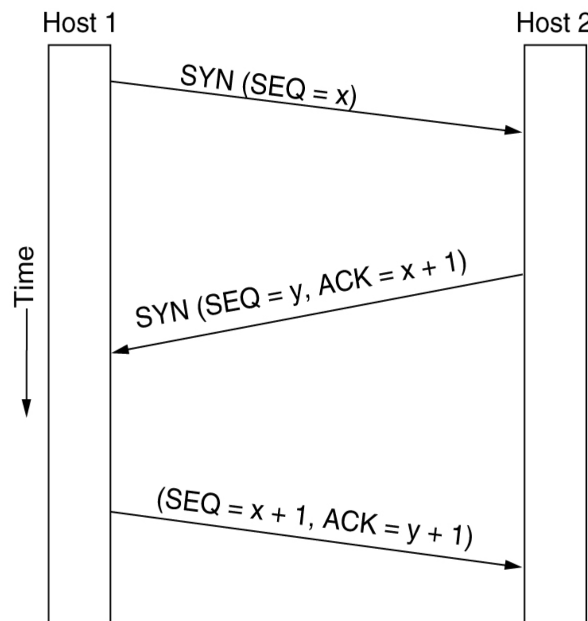
ICANN (former IANA)	Well-known ports	[1, 1023] (Standard ports, e.g. POP3 110)
	Registered ports	[1024, 49151]
	Dynamic/private ports	[49152, 65535]
BSD	Reserved ports	[1, 1023]
	Ephemeral ports	[1024, 5000]
	BSD servers	[5001, 65535]
SUN solaris 2.2	Reserved ports	[1, 1023]
	Non-priviledged ports	[1024, 32767]
	Ephemeral ports	[32768, 65535]

ICANN: Internet Corporation for Assigned Names and Numbers
IANA: Internet Assigned Numbers Authority
BSD: Berkeley Software Distribution
Ephemeral means „short-lived“ (not permanently assigned to an application).

© Peter R. Egli 2011

TCP Connection Establishment (1/4):

➔ A TCP connection is established with 3 TCP packets (segments) going back and forth.



➔ 3-way handshake (SYN, SYN ACK, ACK) is used to synchronize sequence and acknowledge numbers; after 3-way handshake the connection is established.

➔ Host1 performs an „active open“ while host2 does a „passive open“.

➔ A connection consists of 2 independent half-duplex connections. Each TCP peer can close its (outgoing) TCP half-duplex connection any time independently of the other.

➔ A connection can remain open for hours, days, even months without sending data, that is there is no heartbeat poll mechanism!

➔ The Ack-number is the number of the next byte expected by the receiver.

➔The SYN occupies 1 number in the sequence number space (even though a SYN segment usually does not carry user data)!

© Peter R. Egli 2011

 indigoo.com

• **TCP Connection Establishment (2/4):**

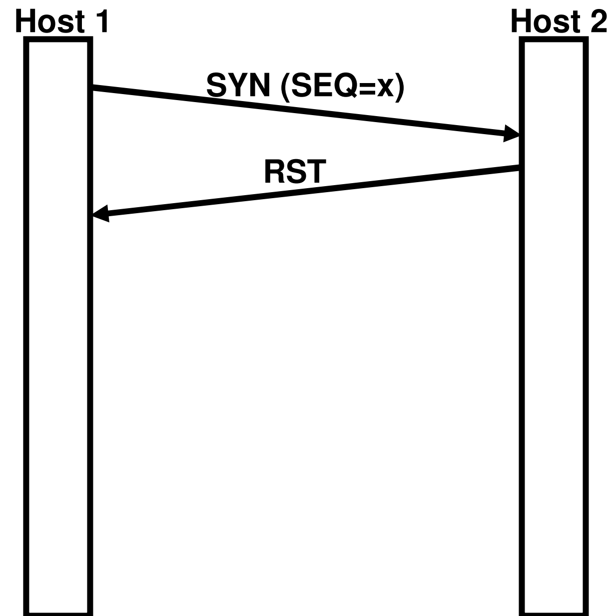
➔ In case of a call collision only 1 connection is created.
Usually Internet applications are client/server where clients do active open and servers passive open (thus no connection establishment collisions possible).
In peer-2-peer applications collisions are possible; e.g. BGP (Internet backbone routing) where BGP speakers establish TCP connections with each other.

Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x , ACK = y + 1)
(b)

© Peter R. Egli 2011

• **TCP Connection Establishment (3/4):**

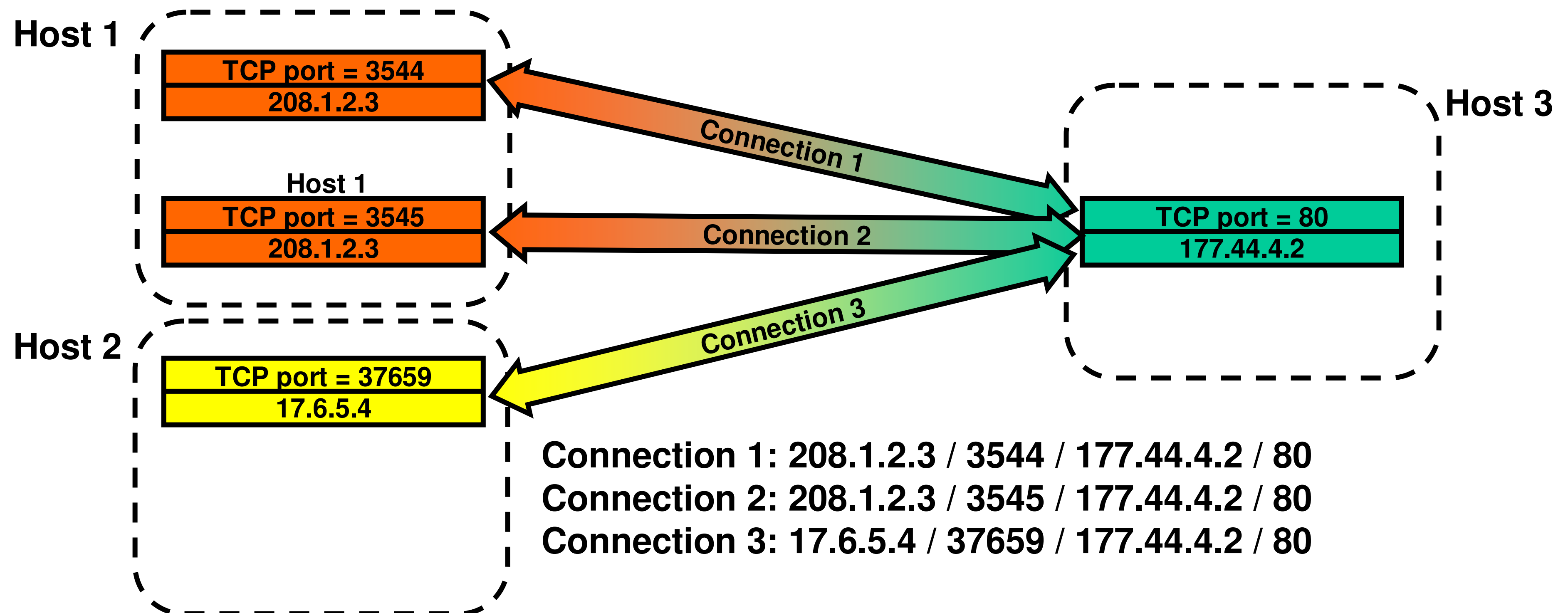
➔ If no server is listening on the addressed port number TCP rejects the connection and sends back a RST (reset) packet (TCP segment where RST bit = 1).



Demo: Telnet connection to closed port (telnet <server IP> 12345).

• TCP Connection Establishment (4/4):

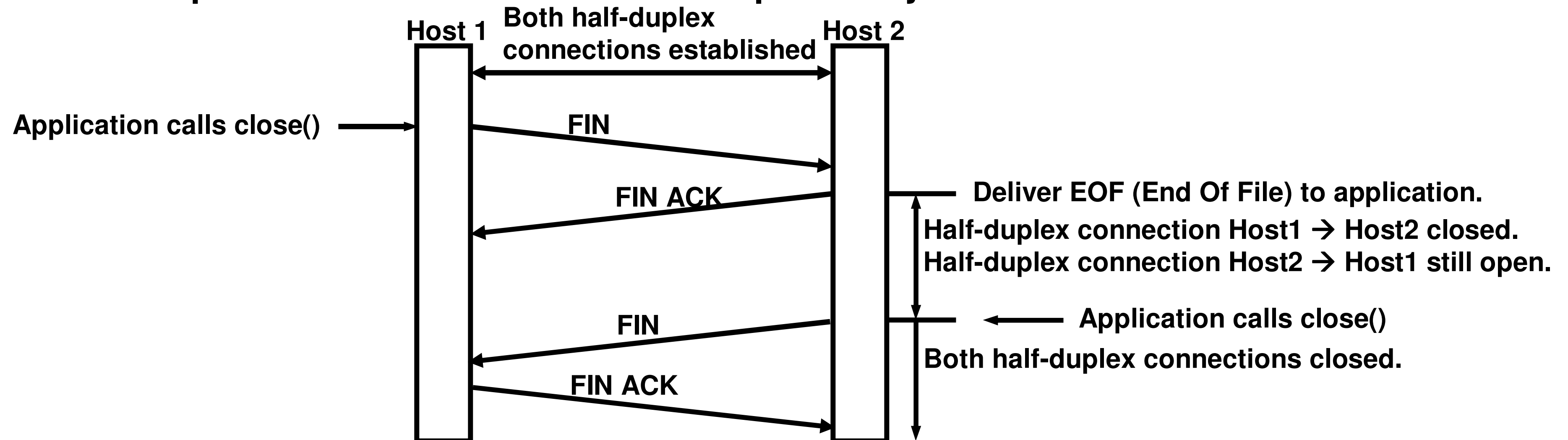
→ A TCP connection is identified by the quadruple source/destination IP address and source/destination port address.
If only one of the addresses of this quadruple is different the quadruple identifies a different TCP connection.



© Peter R. Egli 2011

• **TCP Connection Release (1/2):**

➔ The 2 half-duplex connections are closed independently of each other.



➔ Each host closes "its" half-duplex connection independently of the other (that means the closing of the 2 unidirectional connections is unsynchronized).

➔ Host1 does an active close while host2 a passive close.

➔ Connection closing is a 4-way handshake and not a 3-way handshake since the closing of the half-duplex connections is independent of each other.

➔ Half-close: only one half-duplex connection is closed (still traffic in the other direction).

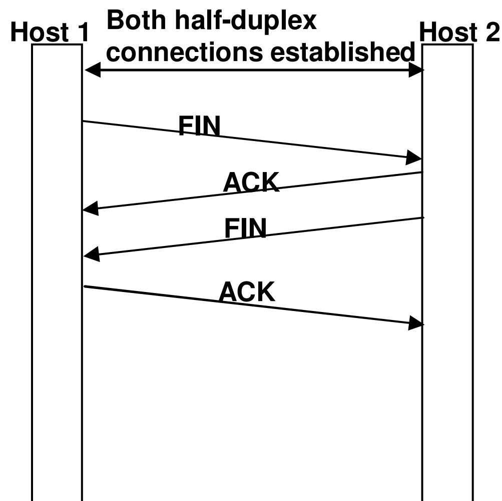
➔ FIN segments occupy 1 number in the sequence number space (as do SYN segments)!

© Peter R. Egli 2011

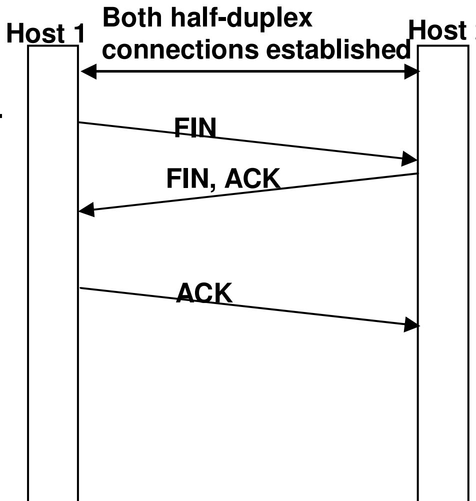
• TCP Connection Release (2/2):

➔ Different scenarios as to how both half-duplex connections are closed:

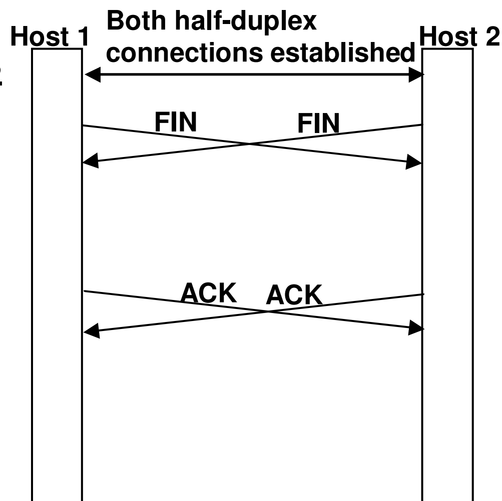
Normal 4-way close:



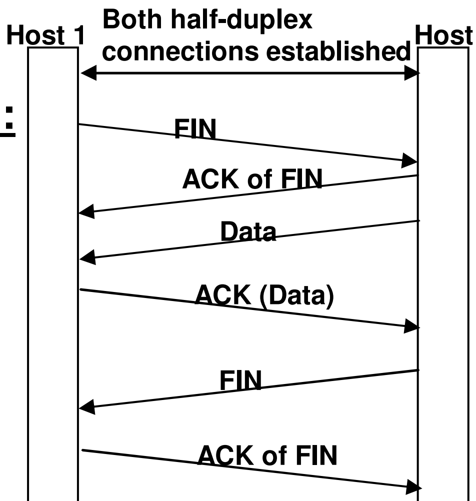
3-way close:



Simultaneous close:



Half-close:



➔Few applications use half-close, e.g. UNIX rsh command:

```
#rsh <host> sort < datafile
```

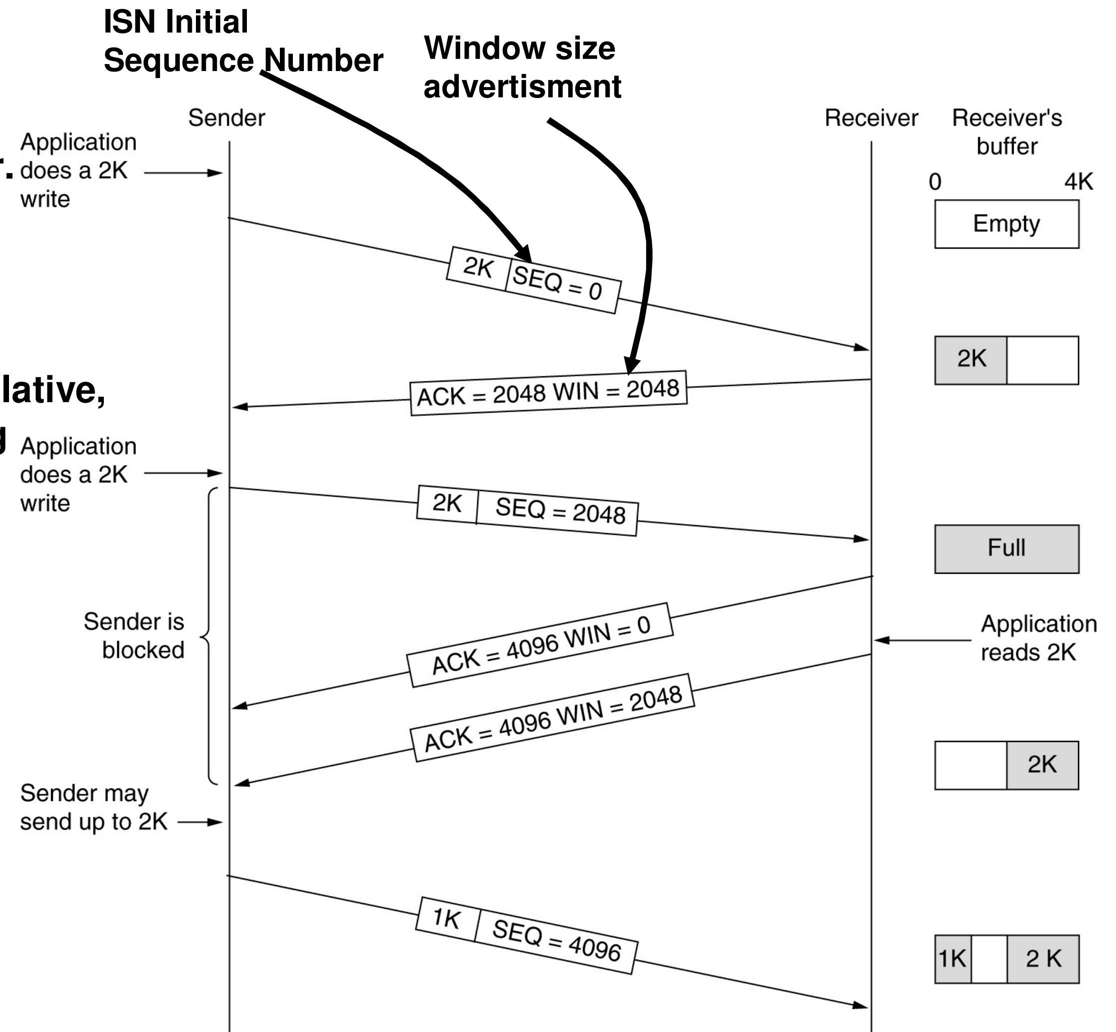
This command is executed remotely. The command needs all input from Host1. The closing of the connection Host1→ Host2 is the only way to tell Host2 that it can start executing the command.
The output of the command is sent back to Host1 through the still existing half-duplex connecion Host2 → Host1.

© Peter R. Egli 2011

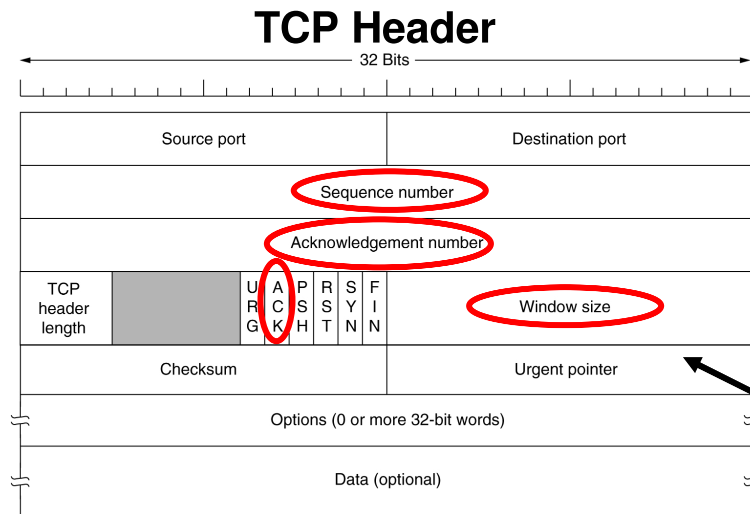
• TCP Flow Control (1/10):

➔ Sliding window mechanism:

- Unlike lock-step protocols TCP allows data burst for maximizing throughput.
- The receiver advertises the size of receive buffer.
- The sequence and acknowledge numbers are per byte (not per segment/packet).
- The receiver's ack number is the number of the next byte it expects; this implicitly acknowledges all previously received bytes. Thus acks are cumulative, Ack=X acknowledges all bytes up to and including X-1.



TCP Header



TCP fields involved in flow control

© Peter R. Egli 2011

• TCP Flow Control (2/10):

1. 3-way handshake for connection establishment. Through corresponding socket calls (indicated by ‚socket()..‘) host ‚A‘ and ‚B‘ open a TCP connection (host ‚A‘ performs an active open while host ‚B‘ listens for an incoming connection request). Host ‚A‘ and ‚B‘ exchange and acknowledge each other the sequence numbers (ISN Initial Sequence Number). Host ‚A‘ has a receive buffer for 6000 bytes and announces this with Win=6000. Host ‚B‘ has a receive buffer for 10000 bytes and announces this with Win=10000. Note that the SYNs occupy 1 number in the sequence number space thus the first data byte has Seq=ISN+1.

2. Application Process (AP) ‚A‘ writes 2KB into TCP. These 2KB are stored in the transmit buffer (Tx buffer). The data remains in the Tx buffer until its reception is acknowledged by TCP ‚B‘. In case of packet loss TCP ‚A‘ has the data still in the Tx buffer for retransmissions.

3. TCP ‚A‘ sends a first chunk of 1500 bytes as one TCP segment. Note that Seq=1001 = ISN+1. These 1500 bytes are stored in TCP ‚B‘s receive buffer.

4. TCP ‚A‘ sends a chunk of 500 bytes. These 500 bytes again are stored in TCP ‚B‘s receive buffer (along with the previous 1500 bytes). The sequence number is Seq=2501 = previous sequence number + size of previous data chunk. Note that the initial 2KB data are still in TCP ‚A‘s Tx buffer awaiting to be acknowledged by TCP ‚B‘.

5. TCP ‚B‘ sends an acknowledge segment acknowledging successful reception of 2KB. This is indicated through Ack=3001 which means that TCP ‚B‘ expects that the sequence number of the next data byte is 3001 or, in other words, the sequence number of the last data byte successfully received is 3000. Upon reception of the acknowledge TCP ‚A‘ flushes the Tx buffer.

6. AP ‚B‘ reads out 2KB with 1 socket call. Application ‚B‘ may have called ‚receive()‘ much earlier and only now TCP ‚B‘ (the socket interface) unblocked the call and returned the chunk of 2KB data. The 2KB data are deleted from host ‚B‘s Rx buffer.

7. TCP ‘B’ sends a pure window update segment to signal to TCP ‘A’ that the receive window size is now 10000 again (Rx buffer). Note that a real TCP implementation would not send a window update if the Rx buffer still has reasonable free capacity. A real TCP implementation would wait for more data to arrive, acknowledge this next data and together with the acknowledge segment also signal the new receive window size. Only when the Rx buffer’s capacity falls below a threshold it is advisable to send a TCP segment merely updating the window size.

© Peter R. Egli 2011

• TCP Flow Control (3/10):

(8) AP ‚A' writes 4KB into TCP. Shortly after application ‚B' writes 2KB into TCP.

(9) TCP ‚A' sends a chunk of 1500 bytes as one TCP segment. Seq = 3001 = last sequence number + size of last data segment.

(10) TCP ‚B' sends a chunk of 1500 bytes as one TCP segment. Seq = 4001 = ISN + 1 (since it is the first TCP segment with data). Win = 8500 = Rx buffer size – size of data in buffer. Ack = 4501 = sequence number of last received segment + size of data. TCP ‚A' deletes the acknowledged 1500 bytes from the Tx buffer (they are successfully received by TCP ‚B', even though not necessarily received by application ‚B'; thus these 1500 do not need to be kept in the Tx buffer for retransmissions and can be deleted).

(11) AP ‚A' writes another 2KB into TCP ‚A'. These 2 KB are stored in TCP ‚A's Tx buffer along with previous 2.5KB of data. Around the same time AP ‚B' reads 1KB of data from its socket. These 1KB are immediately freed from the Rx buffer to make room for more data from TCP ‚A'.

(12) TCP ‚B' sends a chunk of 500 bytes (Seq = 5501 = last sequence number + data size of last segment). The window update in this segment indicates that the Rx buffer has room for 9500 bytes.

(13) TCP ‚A' sends a segment with 1000 bytes. TCP ‚B' writes this data into its Rx buffer there joining the previous 500 byte. Shortly after AP ‚B' reads 1KB from its socket interface leaving 500 byte of data in the Rx buffer.

(14) TCP ‚A' sends a segment with 1000 bytes. TCP ‚B' writes this data into its Rx buffer there joining the previous 500 byte. Shortly after AP ‚B' reads 1KB from its socket interface leaving 500 byte of data in the Rx buffer. Thereupon TCP ‚A' sends an acknowledgment segment with Ack= 4001 + sizes of last 2 received data segments. This Ack segments makes TCP ‚B' delete the 2KB in its Tx buffer since these are no longer needed for possible retransmissions.

(15) TCP ‚A' sends a segment with 1500 bytes. TCP ‚B' writes this data into its Rx buffer there joining the previous 500 byte. Shortly after AP ‚B' reads 2KB from its socket interface thus emptying the Rx buffer.

© Peter R. Egli 2011

• TCP Flow Control (4/10):

16 TCP ‚B' sends an acknowledge segment acknowledging successful reception of all data so far received.
This leaves 2KB of unsent data in TCP ‚A's Tx buffer.
Since no data is in the Rx buffer the receive window size is at its maximum again (Win=10000).
Around the same time AP ‚A' reads out 2KB from the Rx buffer.

17 TCP ‚A' sends a segment with 1500 bytes. TCP ‚B' writes this data into its Rx buffer.

18 TCP ‚A' sends a last data segment with 500 bytes. TCP ‚B' writes this data into its Rx buffer there joining the previous 1500 bytes. Shortly after that AP ‚B' reads out 2KB and thus empties the Rx buffer.

19 TCP ‚B' sends an acknowledgment segment that acknowledges all data received from TCP ‚A'.

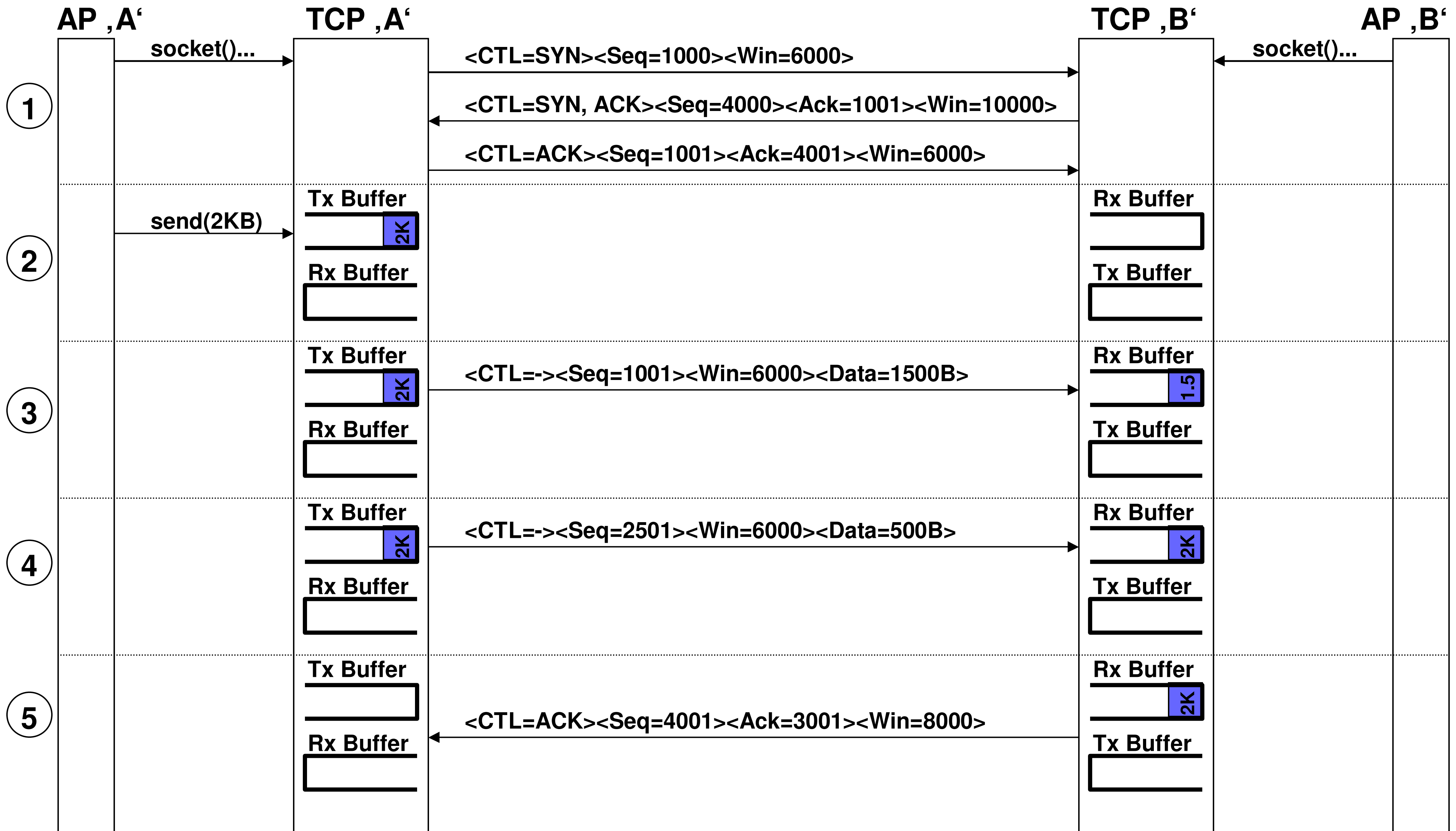
20 AP ‚A' is finished with sending data and closes its socket (close()). This causes TCP ‚A' to send a FIN segment in order to close the half-duplex connection ‚A'→'B'.
TCP ‚B' acknowledges this FIN and closes the connection ‚B'→'A' with a FIN.
Note well that FINs also occupy 1 number in the sequence number space. Thus the acknowledgment sent back by TCP ‚B' has Ack=previous sequence number in ‚A's FIN segment + 1.

Legend:
CTL = Control bits in TCP header (URG, ACK, PSH, RST, SYN, FIN)
Seq = Sequence number
Win = Window size
Ack = Acknowledgement number
Data = Size of application data (TCP payload)

© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

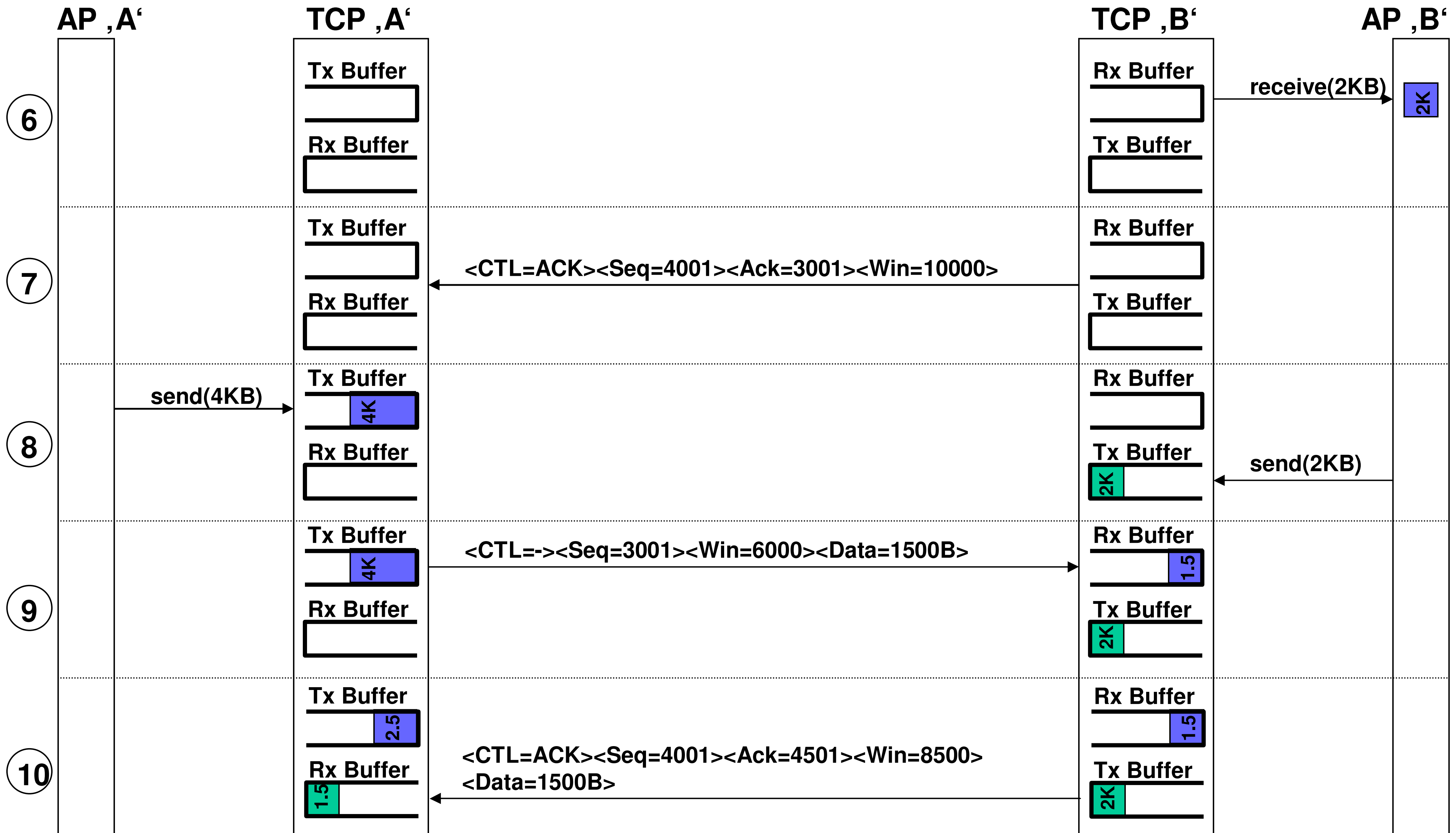
indigoo.com



© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

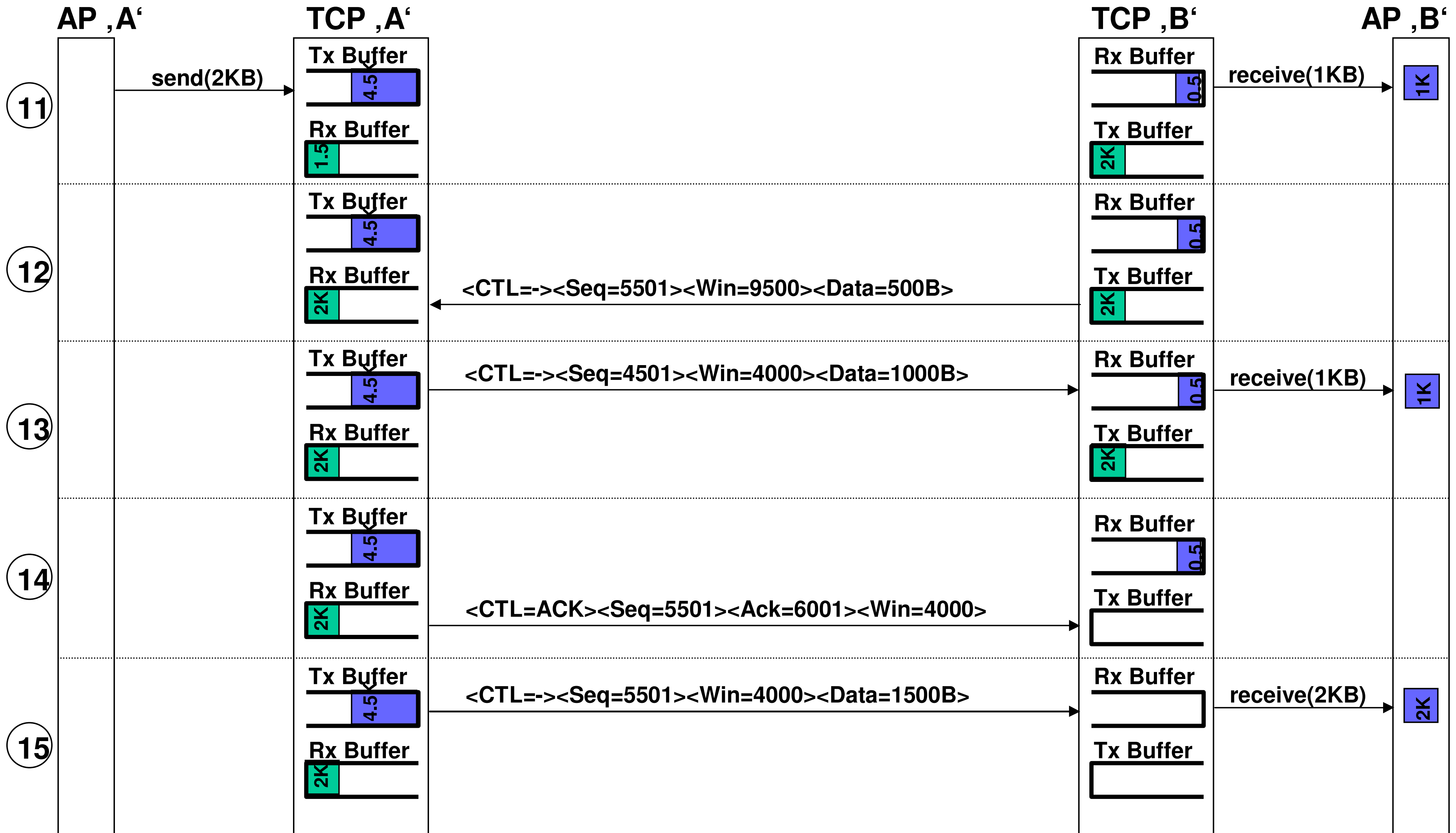
indigoo.com



© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

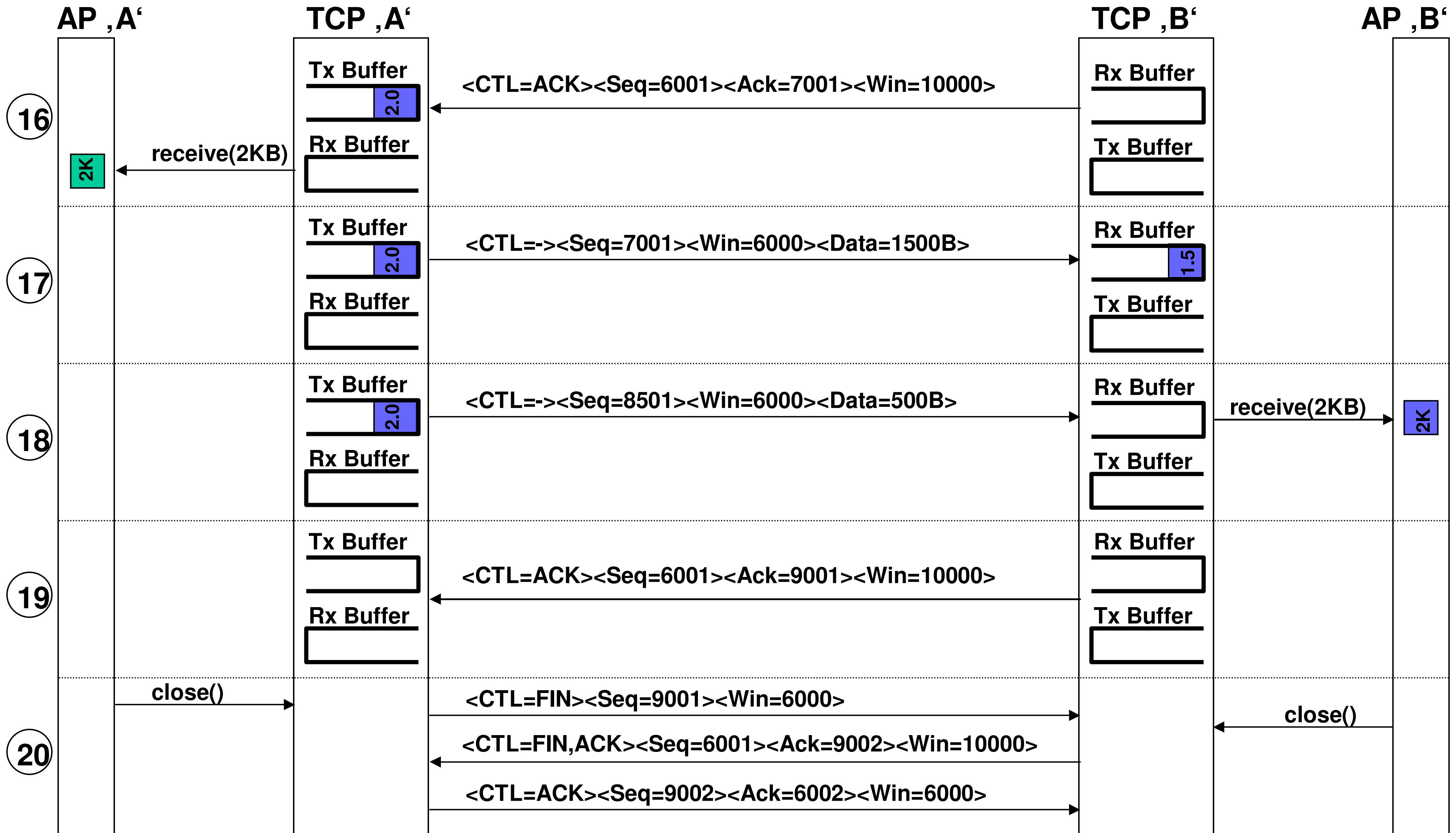
indigoo.com



© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

indigoo.com

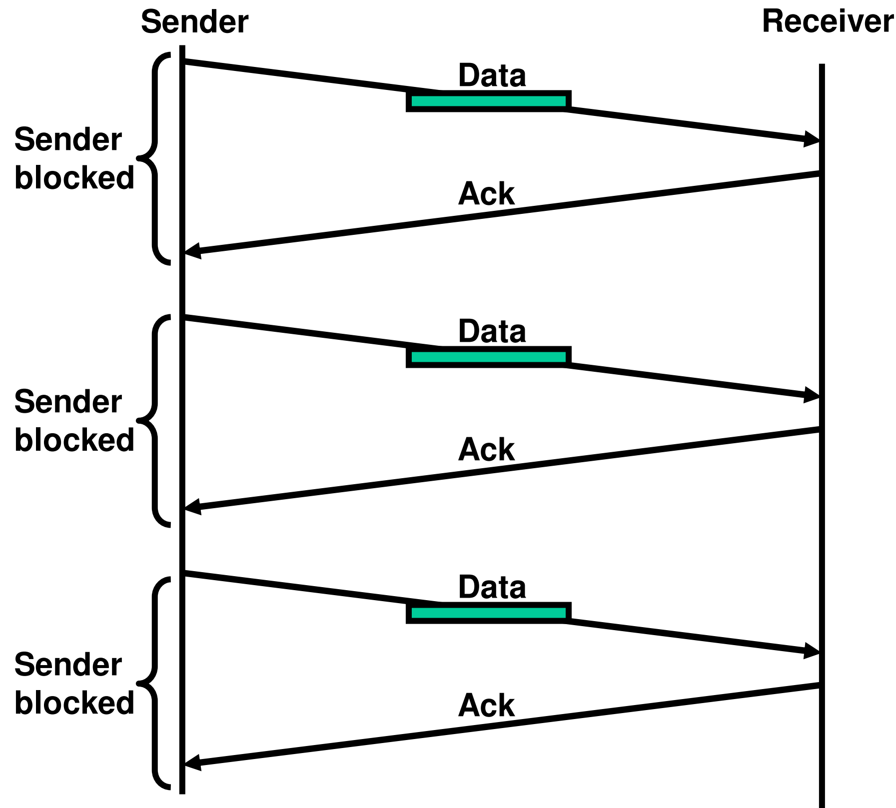


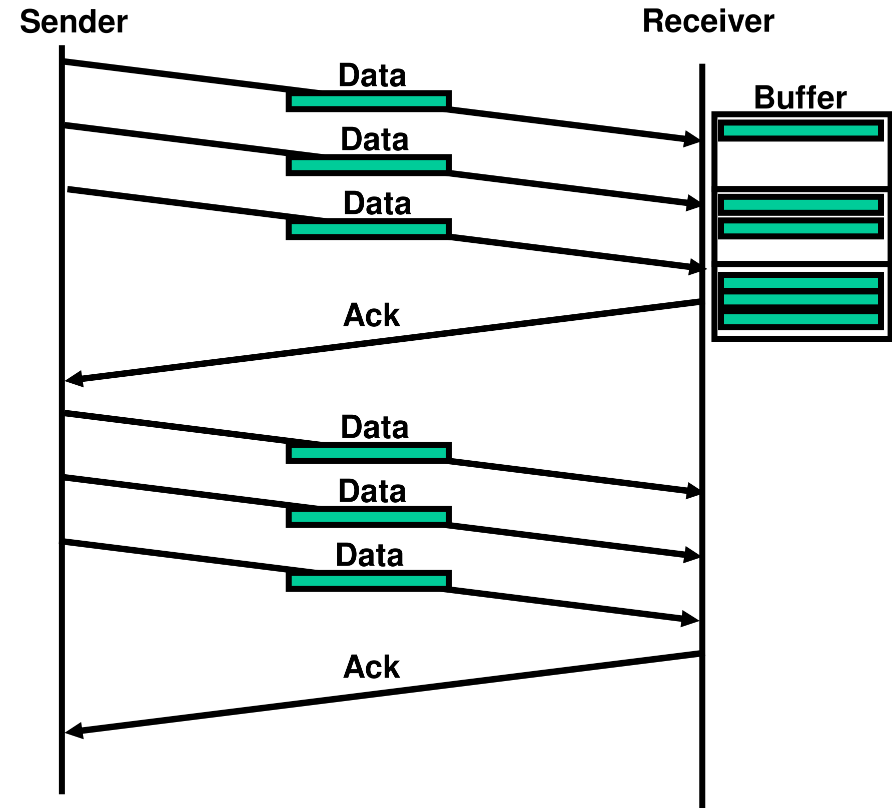
© Peter R. Egli 2011

- **TCP Flow Control (5/10):**

➔ **Sliding window (TCP) versus lock-step protocol:**

1. **Lock-step protocol: sender must wait for Ack before sending next data packet.**
2. **Sliding window: sender can send (small) burst before waiting for Ack.**



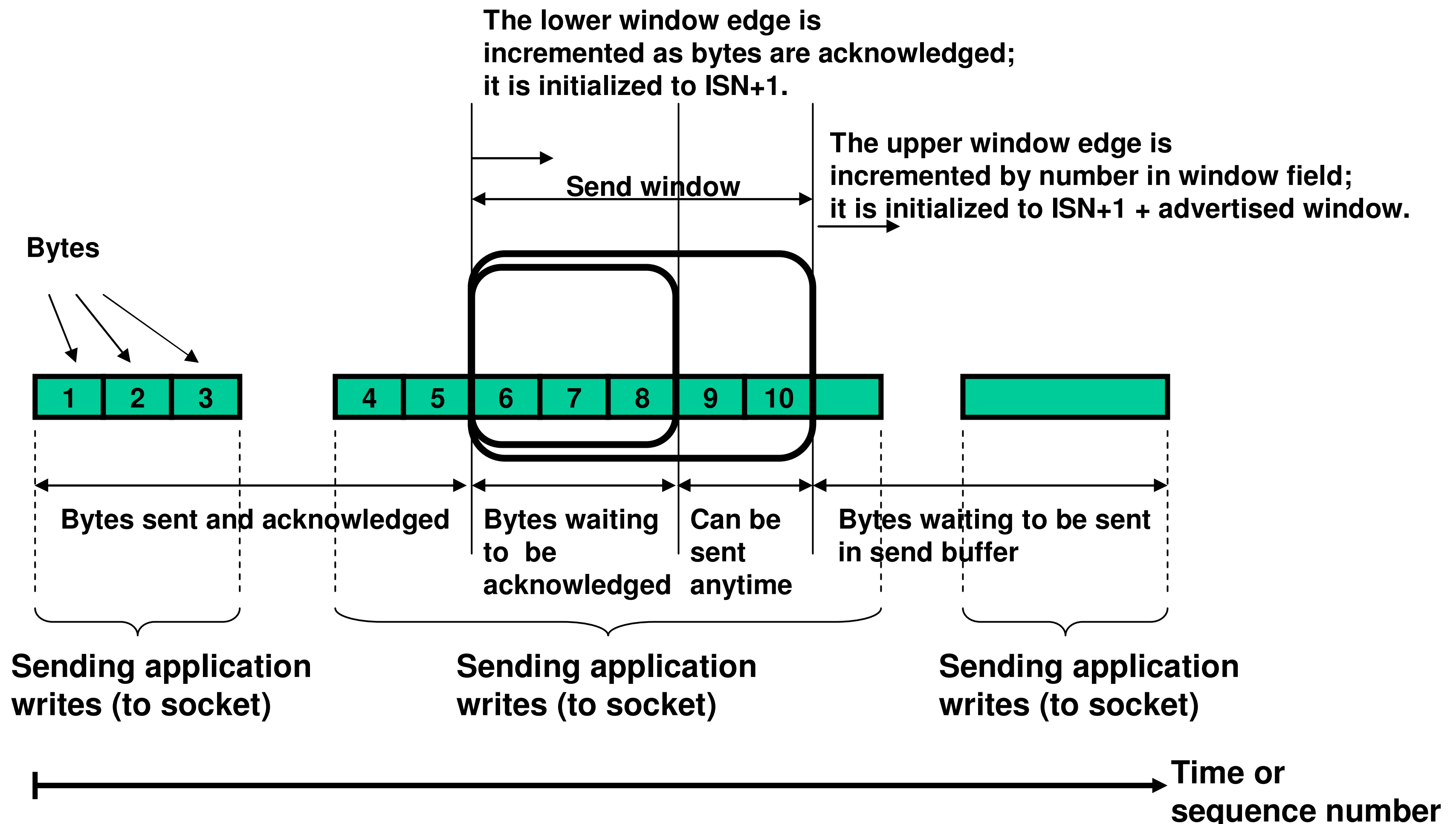


© Peter R. Egli 2011

• TCP Flow Control (6/10):

➔ Sliding window mechanism:

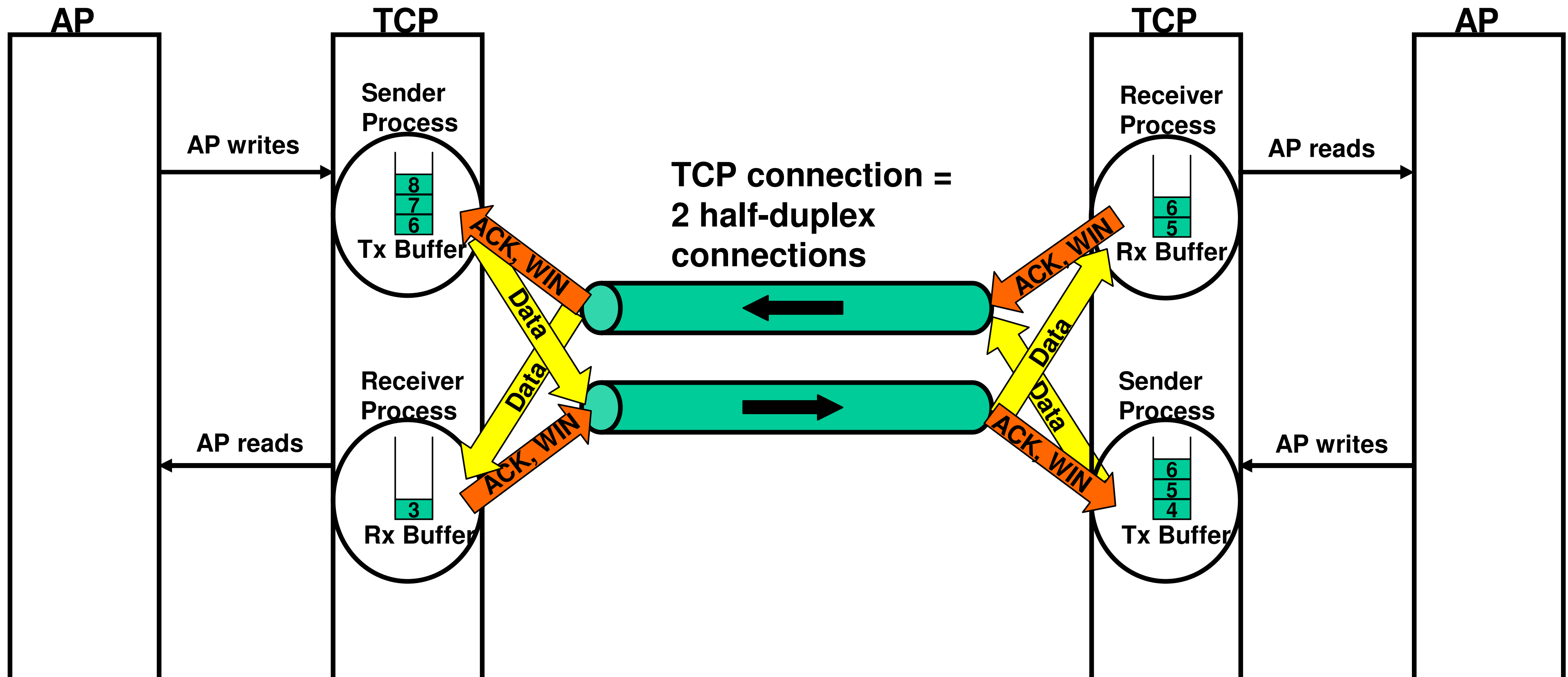
Window size, acknowledgments and sequence numbers are byte based, not segment based.



• TCP Flow Control (7/10):

➔ TCP layer and application interworking:

ACK and window size advertisments are „piggy-backed“ onto TCP segments.
TCP sender and receiver „process“ on a host are totally independent of each other in terms of sequence and acknowledge numbers.



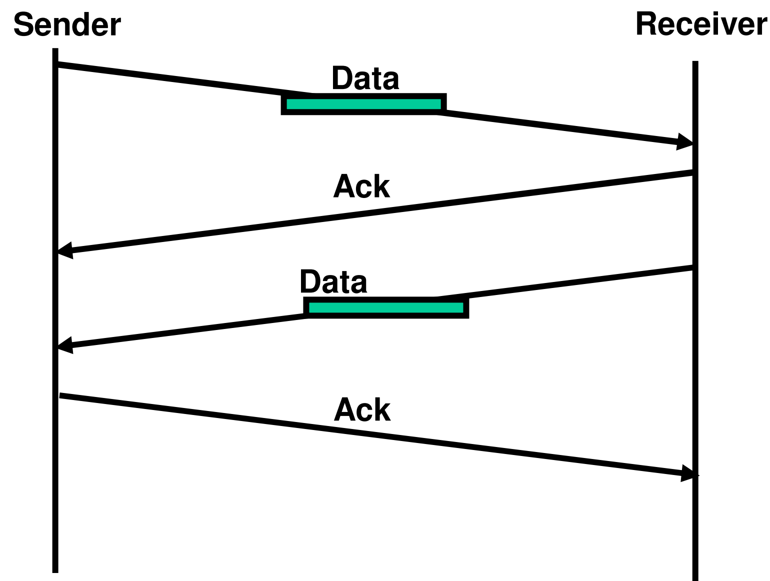
© Peter R. Egli 2011

• TCP Flow Control (8/10):

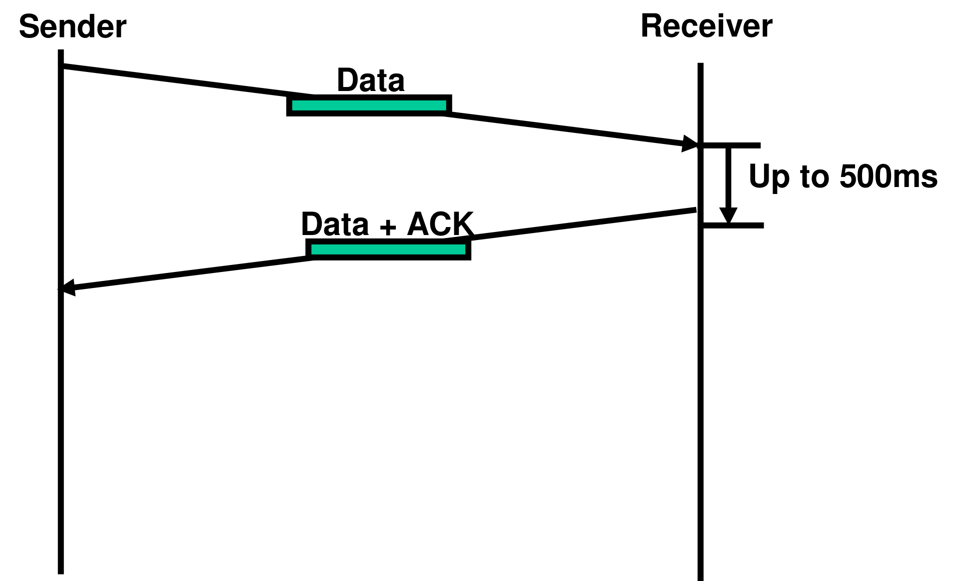
➔ Delayed acknowledgments for reducing number of segments:

The receiver does not send Acks immediately after the receipt of an (error-free) packet but waits up to ~200ms/500ms if a packet is in the send buffer (depends on host OS).
If so it „piggybacks“ the Ack onto the transmit packet; if no transmit packet is available the receiver sends an Ack latest after ~200ms/500ms.

Without delayed acks:



With delayed acks:



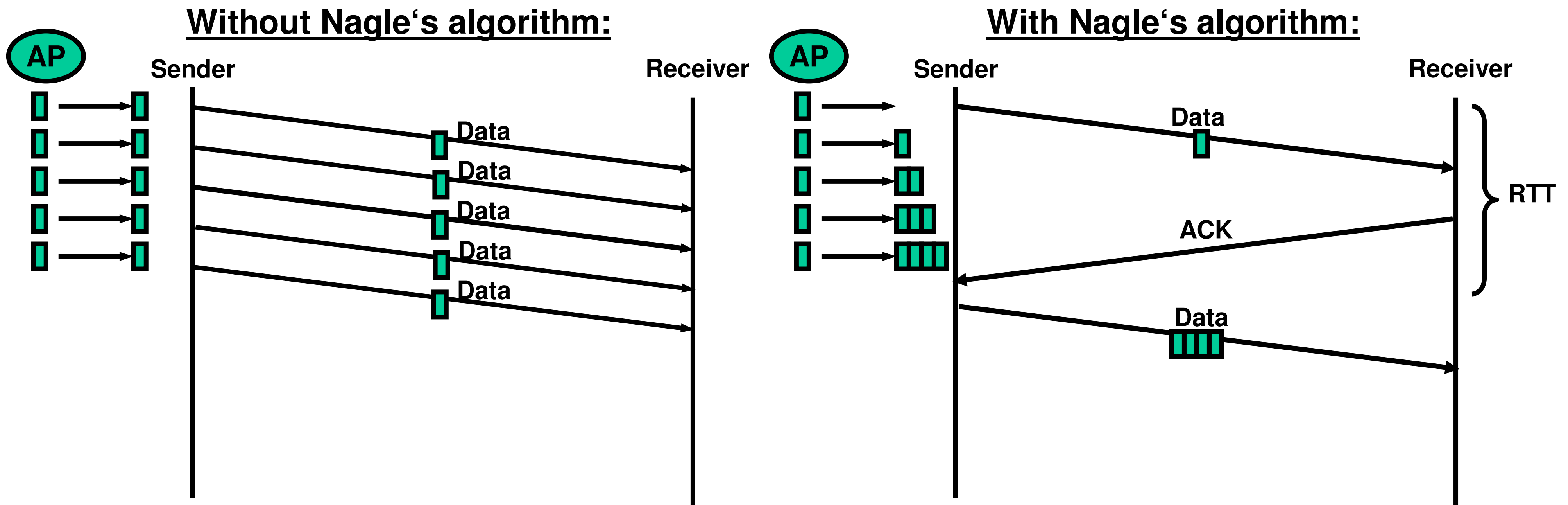
Demo: Telnet connection with Echo:
➔ Without 'delayed ack' 4 segments per character (character, echoed character, 2 acks).
➔ With 'delayed ack' 2 segments (character, ack).

© Peter R. Egli 2011

• TCP Flow Control (9/10):

➔ Nagle's Algorithm for further optimization:

In interactive applications (Telnet) where single bytes come in to the sender from the application sending 1 byte per TCP segment is inefficient (98% overhead).
When activated Nagle's algorithm sends only the first byte and buffers all subsequent bytes until the first byte has been acknowledged. Then the sender sends all so far buffered bytes in one segment. This can save a substantial number of TCP segments (IP packets) when the user types fast and the network is slow.



RTT Round Trip Time (time between sending a packet and receiving the response).

• TCP Flow Control (10/10):

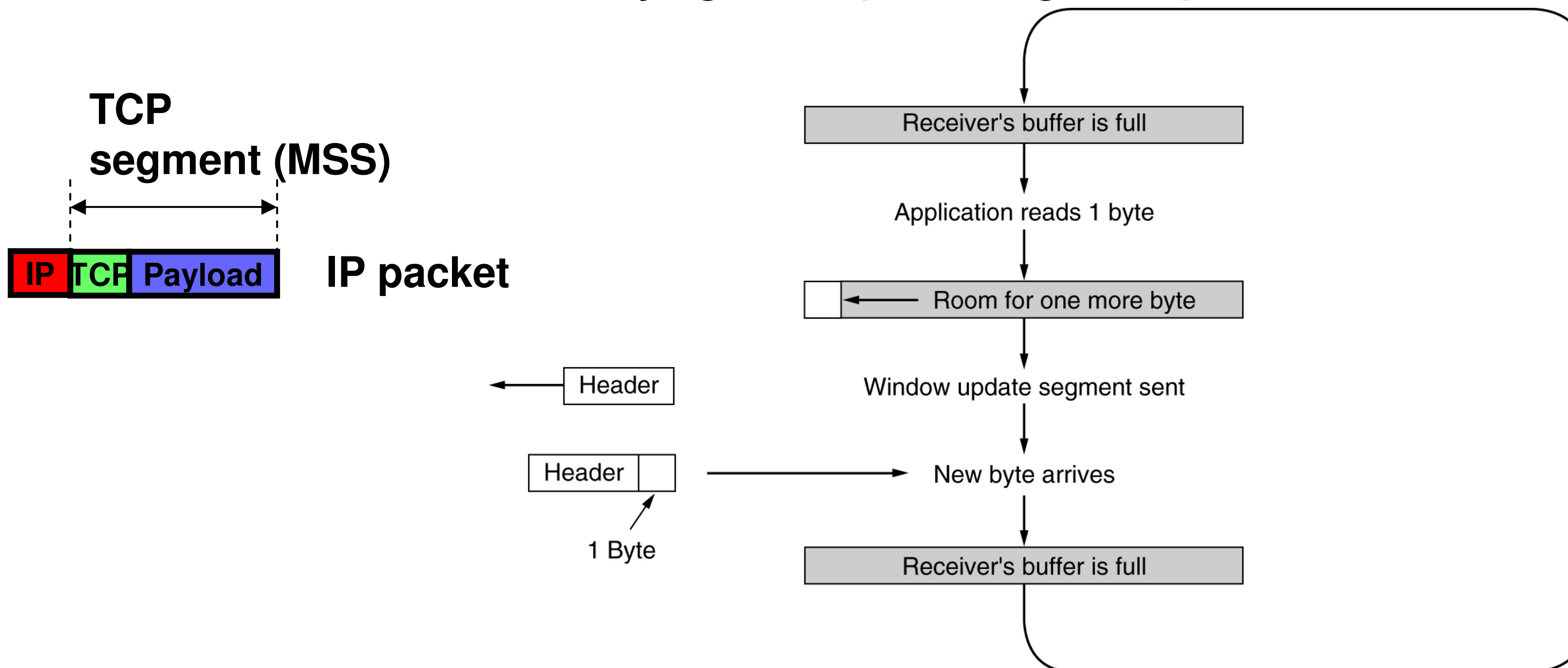
➔ **Silly window syndrome:**

This Problem occurs when the receiver reads out only small amounts of bytes from the receive buffer and the sender sends relatively fast (and in large chunks). Then the flow-control mechanism becomes inefficient and ruins TCP performance.

Solution: Clark's algorithm.

➔ **The receiver only sends window updates when there is sufficient space in receive buffer (sufficient = min(MSS, half buffer size)).**

➔ **The sender must not send „tynigrams" (small segments).**

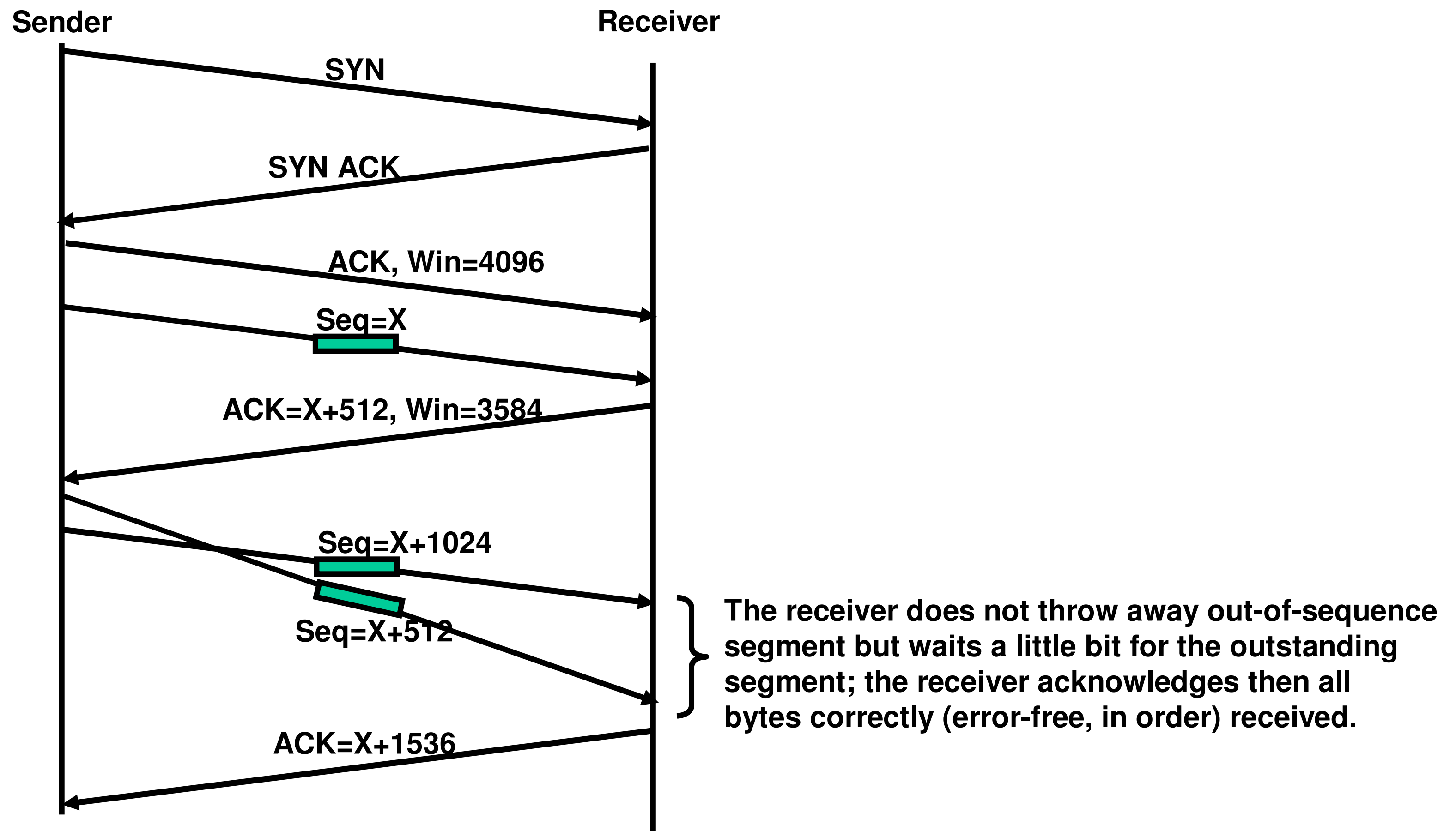


© Peter R. Egli 2011

• TCP Error Control (1/6):

➔ Out of sequence segments:

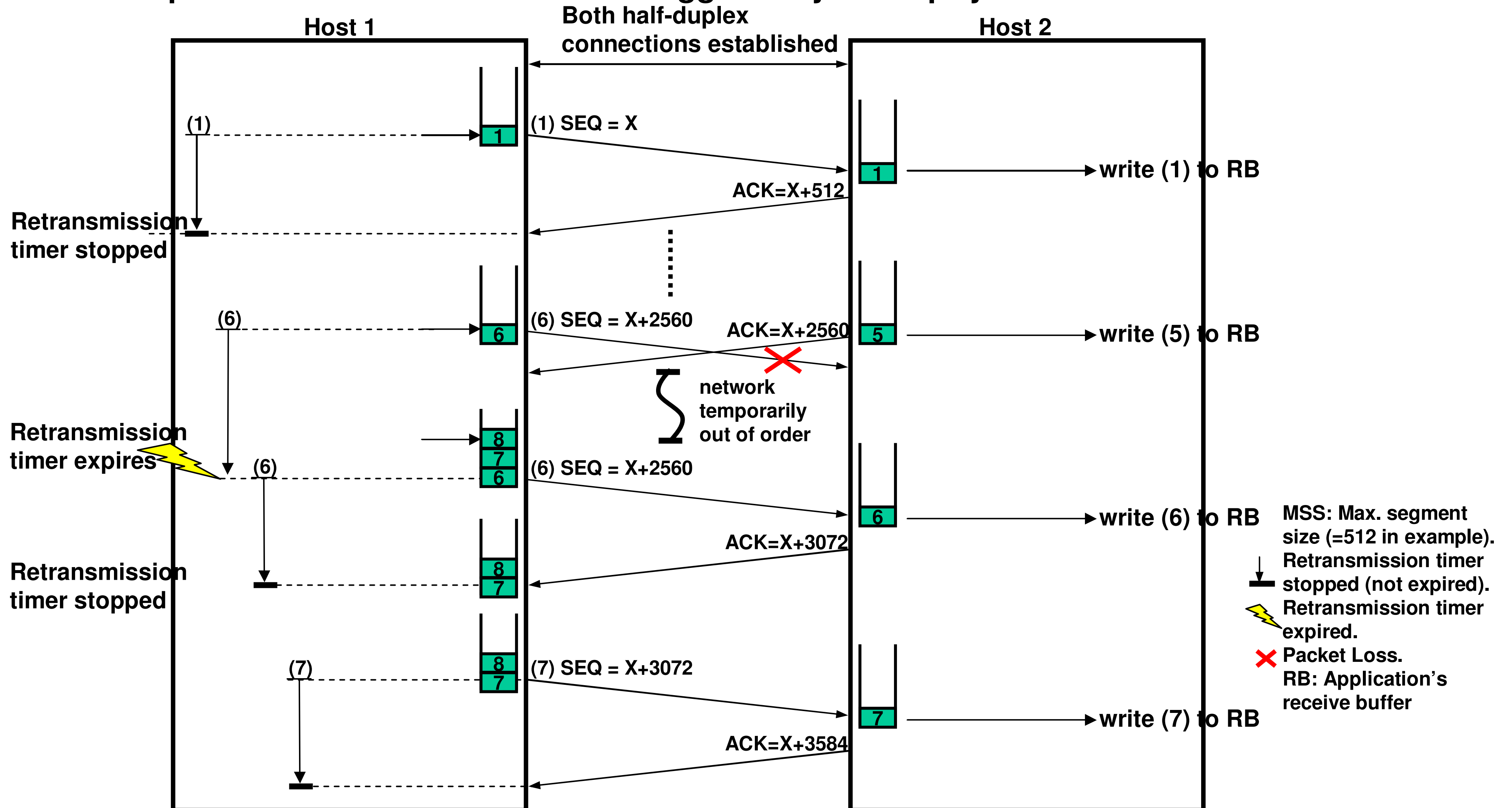
Out of sequence segments are not dropped but buffered for later use.



© Peter R. Egli 2011

• TCP Error Control (2/6):

➔ Lost packets cause retransmissions triggered by the expiry of the retransmission timer:

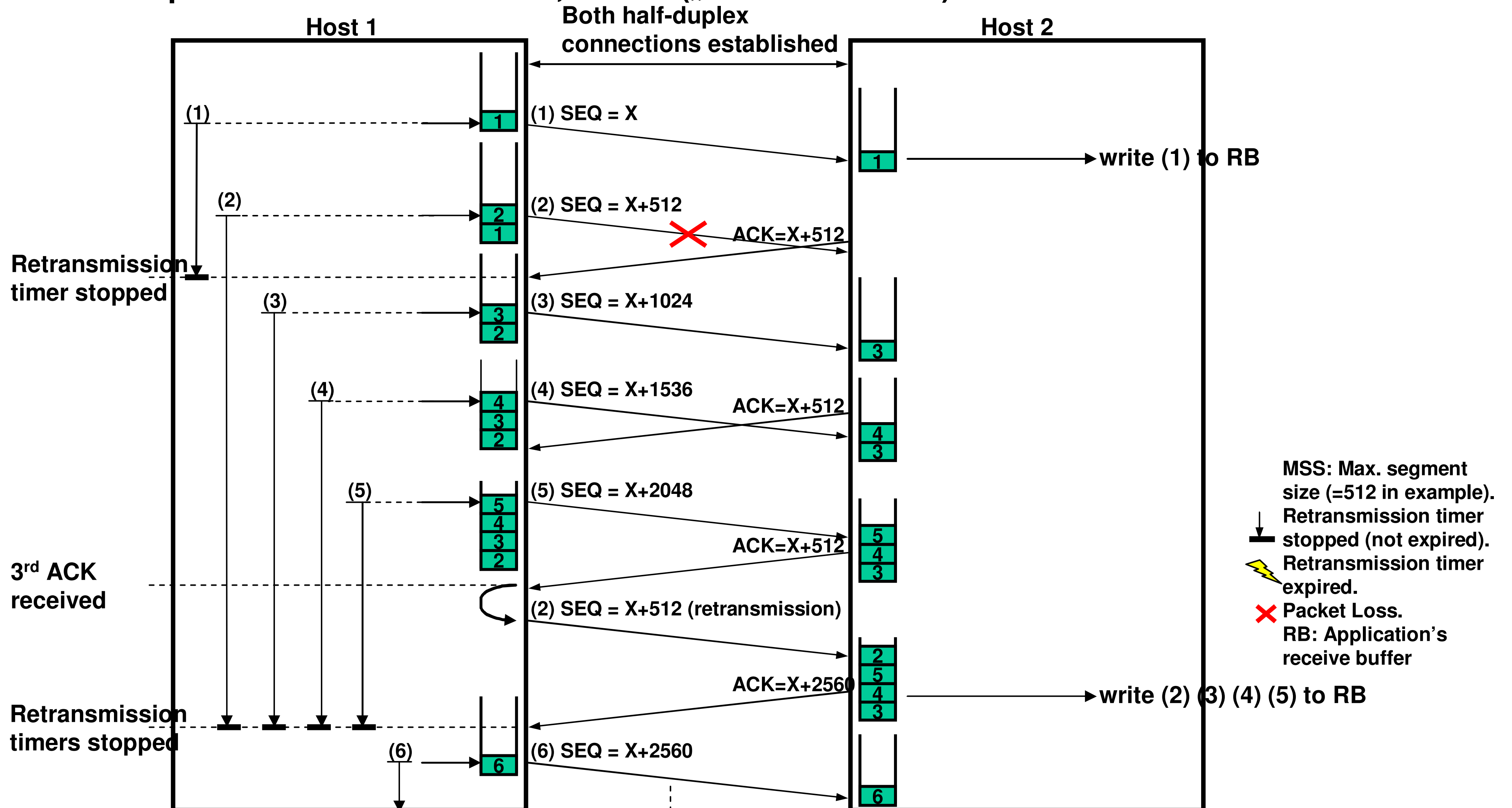


TCP - Transmission Control Protocol – RFC793

indigoo.com

• TCP Error Control (3/6):

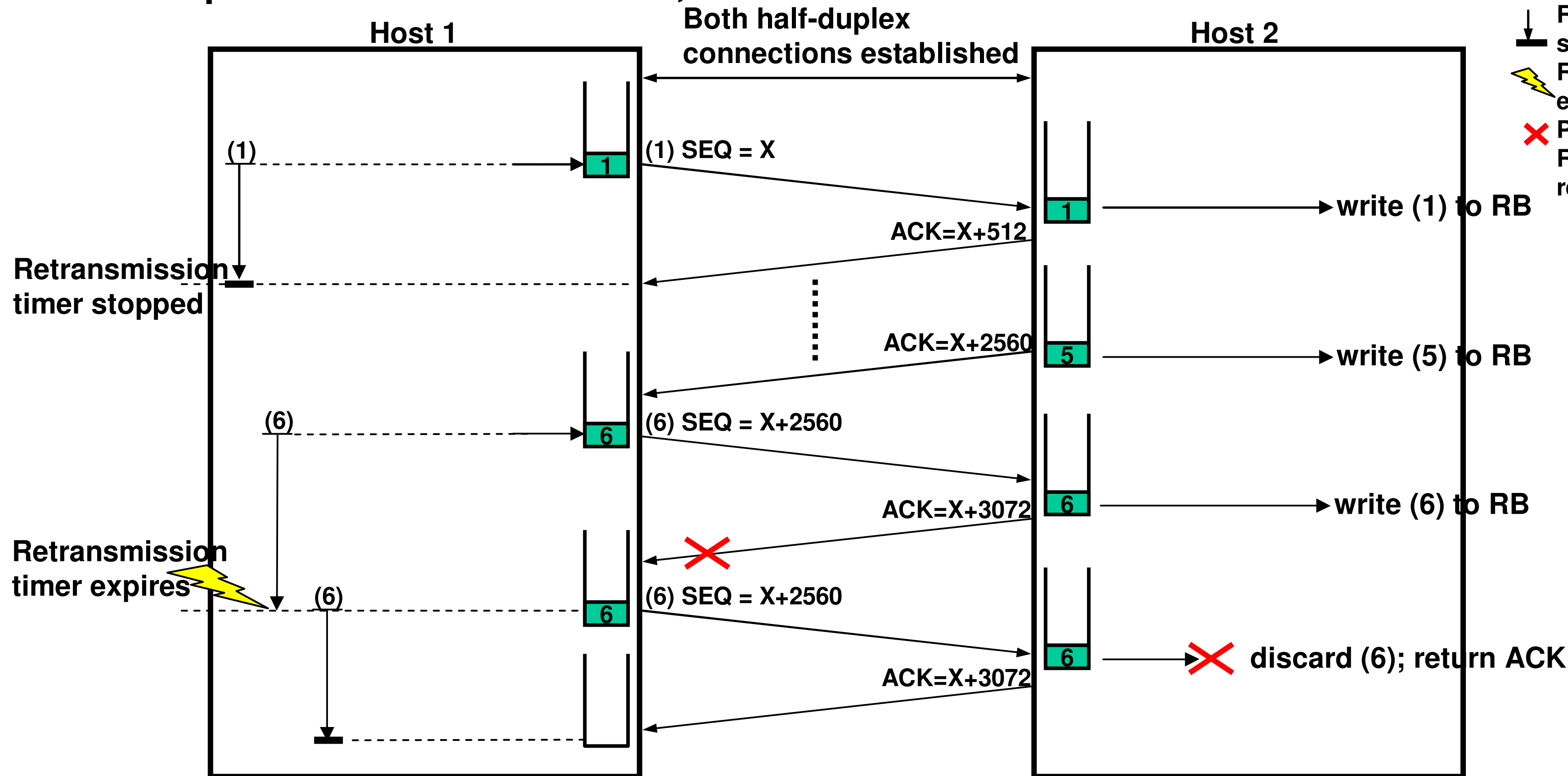
➔ **Lost packets / retransmissions, 3 acks („fast retransmit“):**



TCP - Transmission Control Protocol – RFC793

• **TCP Error Control (4/6):**

➔ **Lost packets / retransmissions, lost ack:**



➔ **TCP does not specify how often a specific segment has to be retransmitted (in case of repeated packet loss of the same segment). Typical values are max. 5 or 8 retransmissions of the same segment (if more retransmissions necessary connection is closed).**

© Peter R. Egli 2011

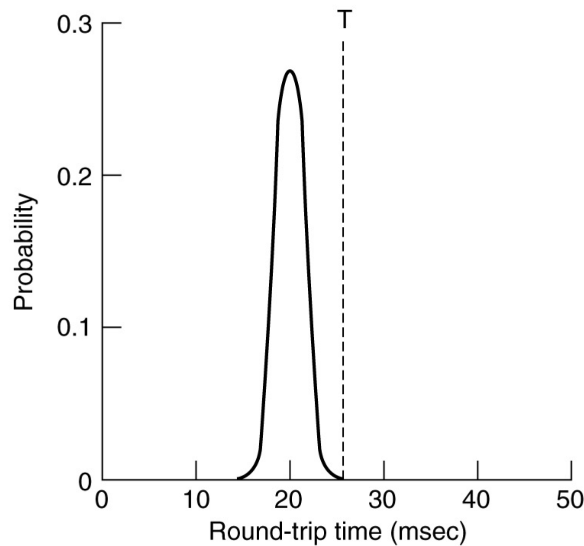
• TCP Error Control (5/6):

➔ Retransmission timer (RTO Retransmission Timeout) assessment:

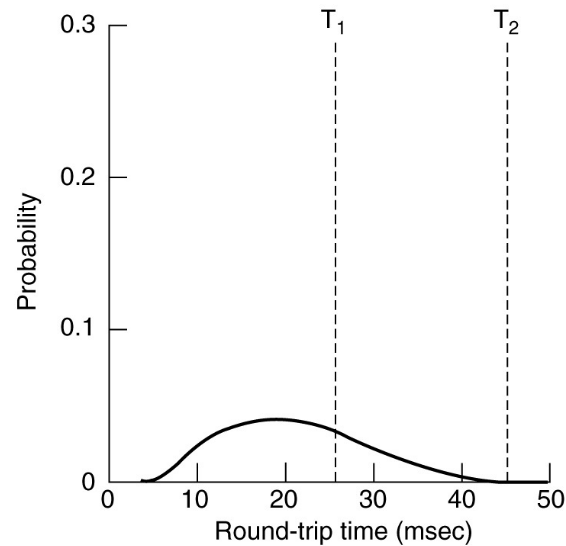
Problem:

a. Single data link ➔ RTO = (2 + e)*RTT where e<<2 (RTT rather deterministic).

b. Internet (multiple data links) ➔ RTT varies considerably, also during lifetime of a TCP conn.



(a)



(b)

Solution:

Constantly adjust RTO based on measurement of RTT. For each segment that was sent start a (retransmission) timer.

RTO = RTT * 4 * D

where D is mean deviation as per $D_{new}=\alpha D_{old} + (1- \alpha) * |RTT – M|$

M = observed RTT value for a specific ACK

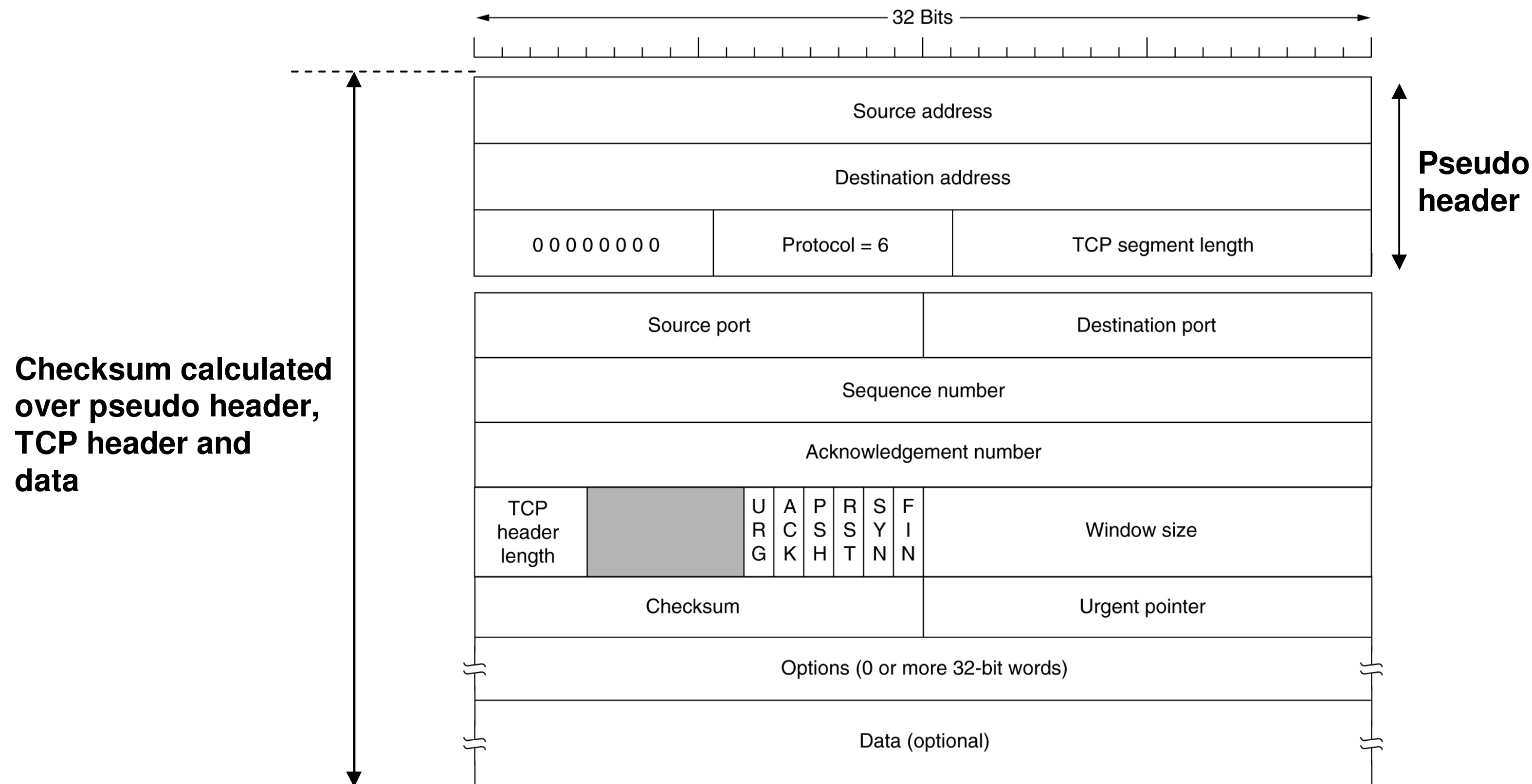
α = smoothing factor that determines how much weight is given to the old value (typ. α=7/8)

© Peter R. Egli 2011

• TCP Error Control (6/6):

➔ TCP Header Checksum Calculation:

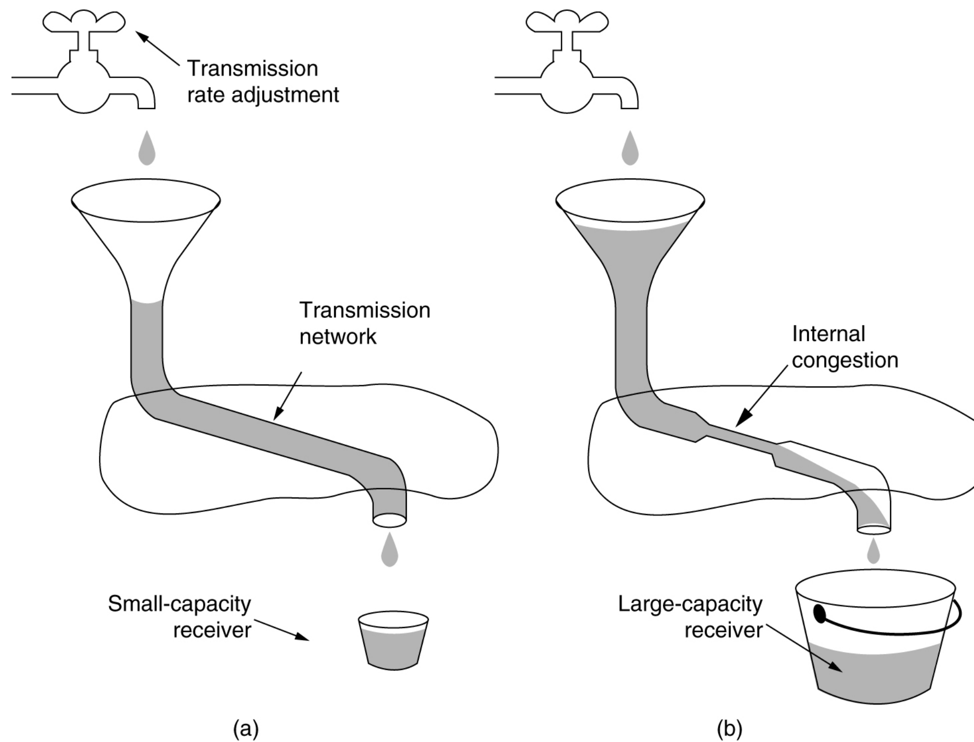
TCP uses a 16-bit checksum for the detection of transmission errors (bit errors). The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the pseudo IP header, TCP header and data (the checksum field is initialized with zeroes before the checksum calculation). By including the IP header in the checksum calculation TCP depends on IP; it may not run on anything other than IP (this is a violation of the layering principle!).



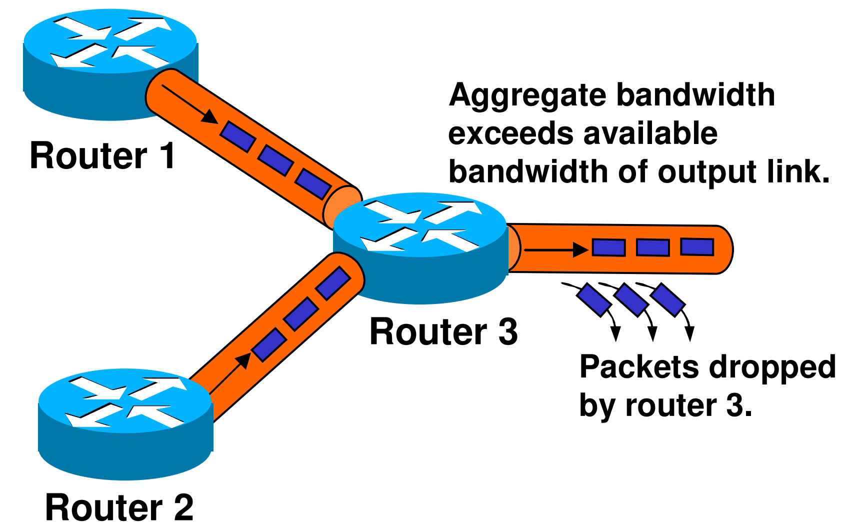
• TCP Congestion Control RFC2001 (1/6):

➔ Congestion control is a control (feedback control) mechanism to prevent congestion (congestion avoidance).

Problem (a): A fast network feeding a low capacity receiver (congestion in receiver).
Problem (b): A slow network feeding a high-capacity receiver (congestion in network).



Congestion can always occur (aggregate ingress Bandwidth exceeds egress bandwidth).



© Peter R. Egli 2011

• TCP Congestion Control RFC2001 (2/6):

➔Prior to 1988 TCP did not define a congestion control mechanism. TCP stacks just sent as many TCP segments as the receiver's buffer could hold (based on advertise window). With the growth of the Internet router links became congested and thus buffers got full which resulted in packet loss. Packet loss lead to retransmissions which in turn aggravated the congestion problem.

➔ It became necessary to avoid congestion in the first place (it's always better to avoid a problem in the first place rather than cure it!).

➔ How to avoid congestion efficiently (ideas of Van Jacobson):

a. Sender can detect packet loss and interpret it as network congestion (this is in most cases true; packet loss can however also be due to bit errors).
b. When a sender detects packet loss it should reduce the transmission rate quickly.
c. When there is no (more) congestion (no more packet loss) the sender can slowly increase the transmission rate.
e. At the very beginning of a TCP session a sender does not know the maximum possible transmission rate yet. Thus it should start slowly with sending segments („slow start" procedure).
f. If a sender receives an ACK packet it knows that a segment has been received and is no longer on the network (in transmission). This fact is used for the slow start algorithm (see below). ➔ This procedure is „self-clocking".

➔ Network stability is achieved through quickly reacting to bad news (packet loss), moderately reacting to good news (no packet loss, all segments sent are acknowledged).

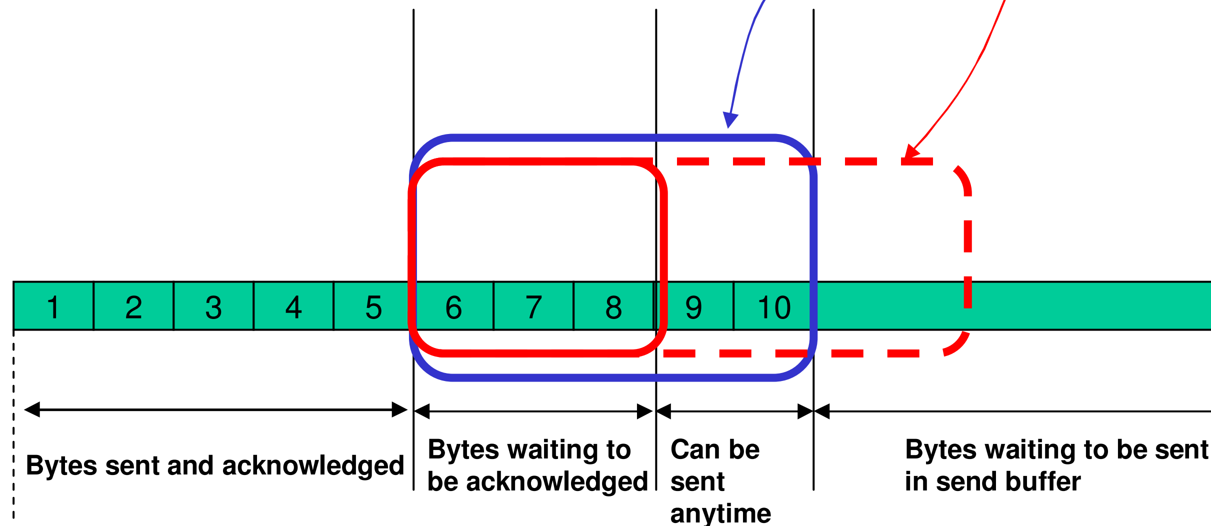
© Peter R. Egli 2011

• TCP Congestion Control RFC2001 (3/6):

➔ **Congestion control window:**

A congestion control window is used in addition to the flow control window (both windows work in parallel).
The max. number of segments that can be sent = min(Ws, Wc).
Ws = Flow control window
Wc = Congestion control window

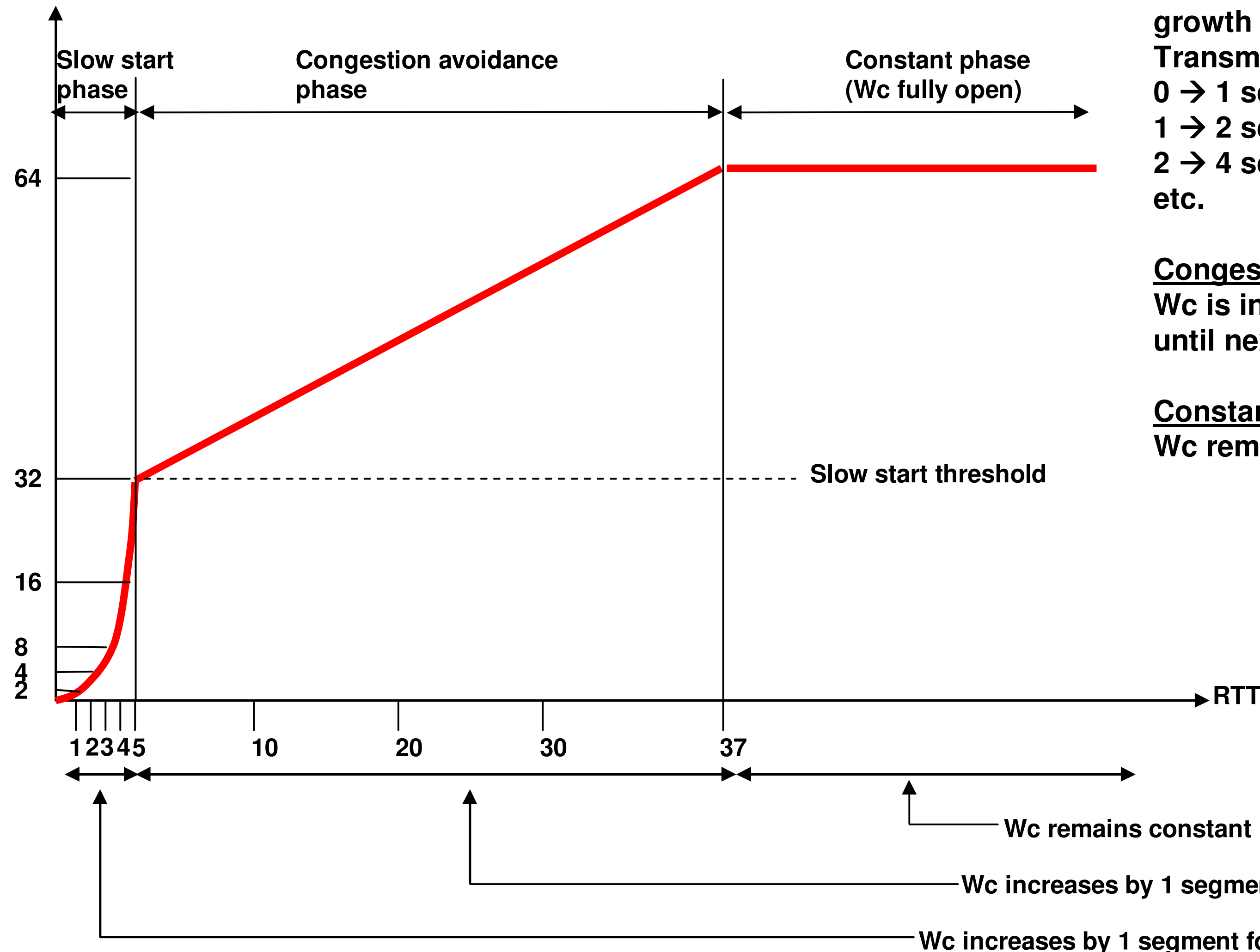
Heavily loaded networks ➔ flow of segments controlled by Wc.
Lightly loaded networks ➔ flow of segments controlled by Ws.



© Peter R. Egli 2011

• TCP Congestion Control RFC2001 (4/6):

➔ Normal congestion control window procedure:



Slow start phase:
Wc starts opening „slowly“ from 1 segment until threshold (SST) reached (exponential growth of Wc).
Transmission number:
0 → 1 segment sent
1 → 2 segments sent (burst)
2 → 4 segments sent (burst)
etc.

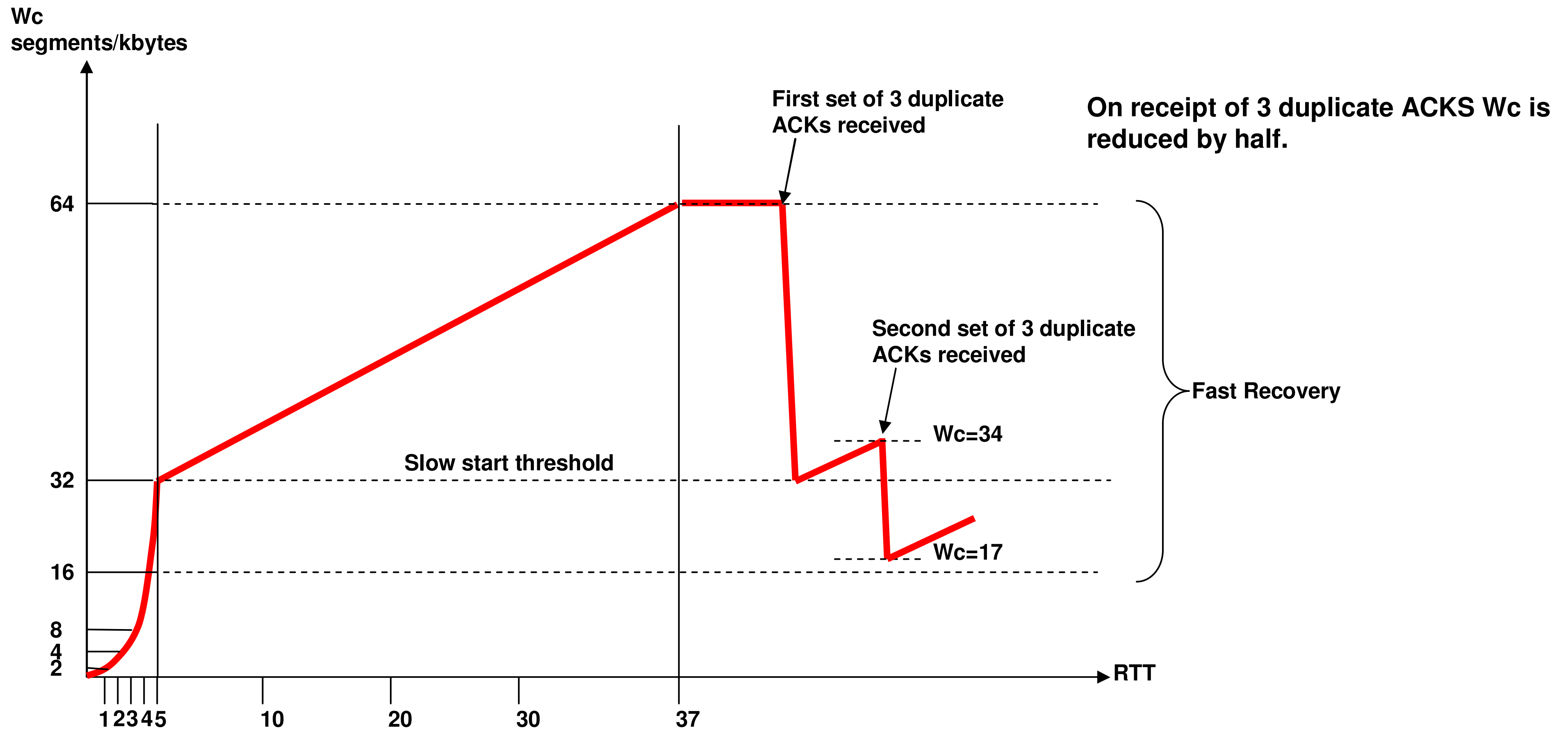
Congestion avoidance phase:
Wc is increased by 1/Wc (linear growth) until next threshold reached.

Constant phase:
Wc remains constant.

© Peter R. Egli 2011

- **TCP Congestion Control RFC2001 (5/6):**

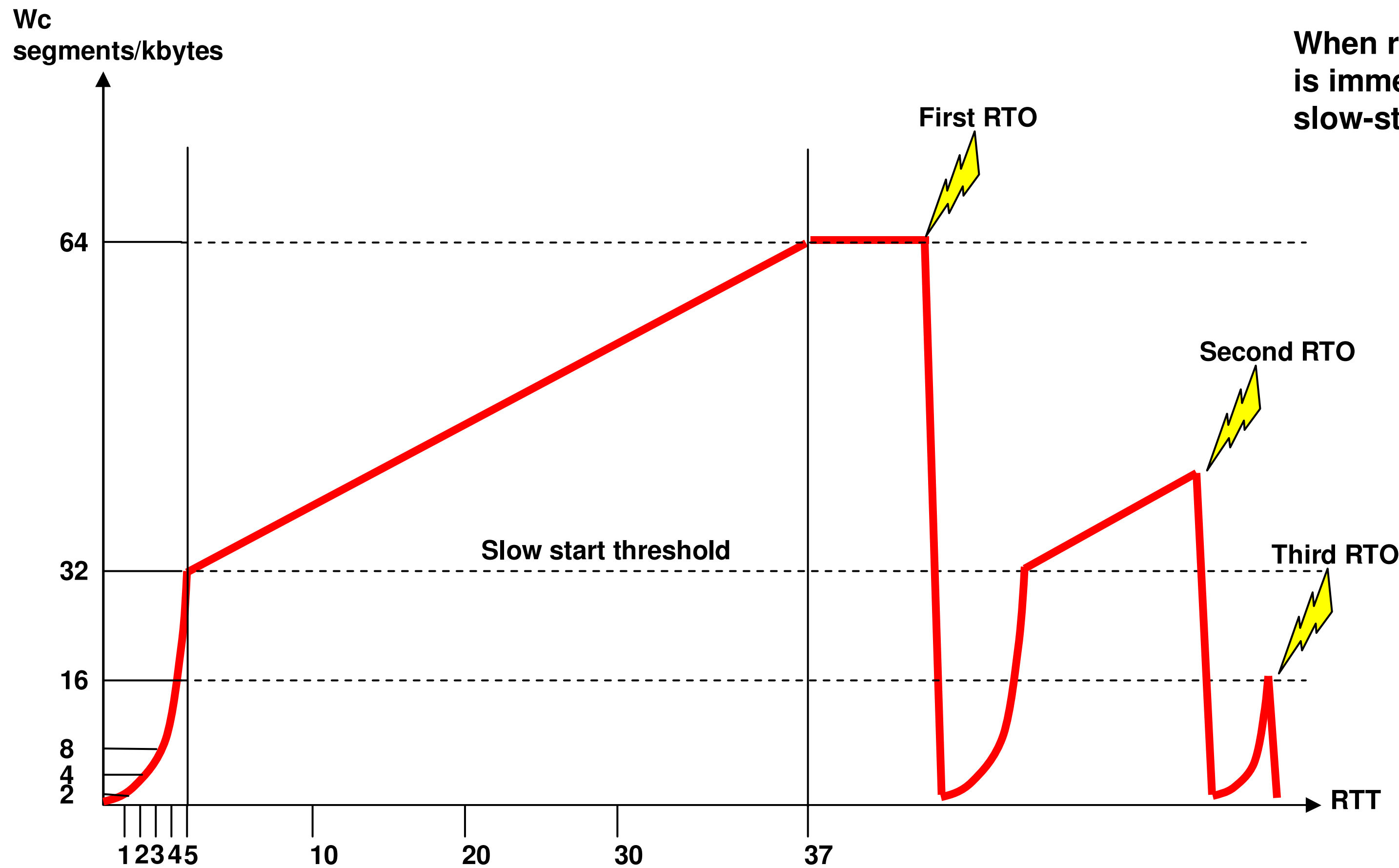
➔ **Light congestion (reception of 3 duplicate Acks):**



© Peter R. Egli 2011

• TCP Congestion Control RFC2001 (6/6):

➔ **Heavy congestion (no reception of Acks):**

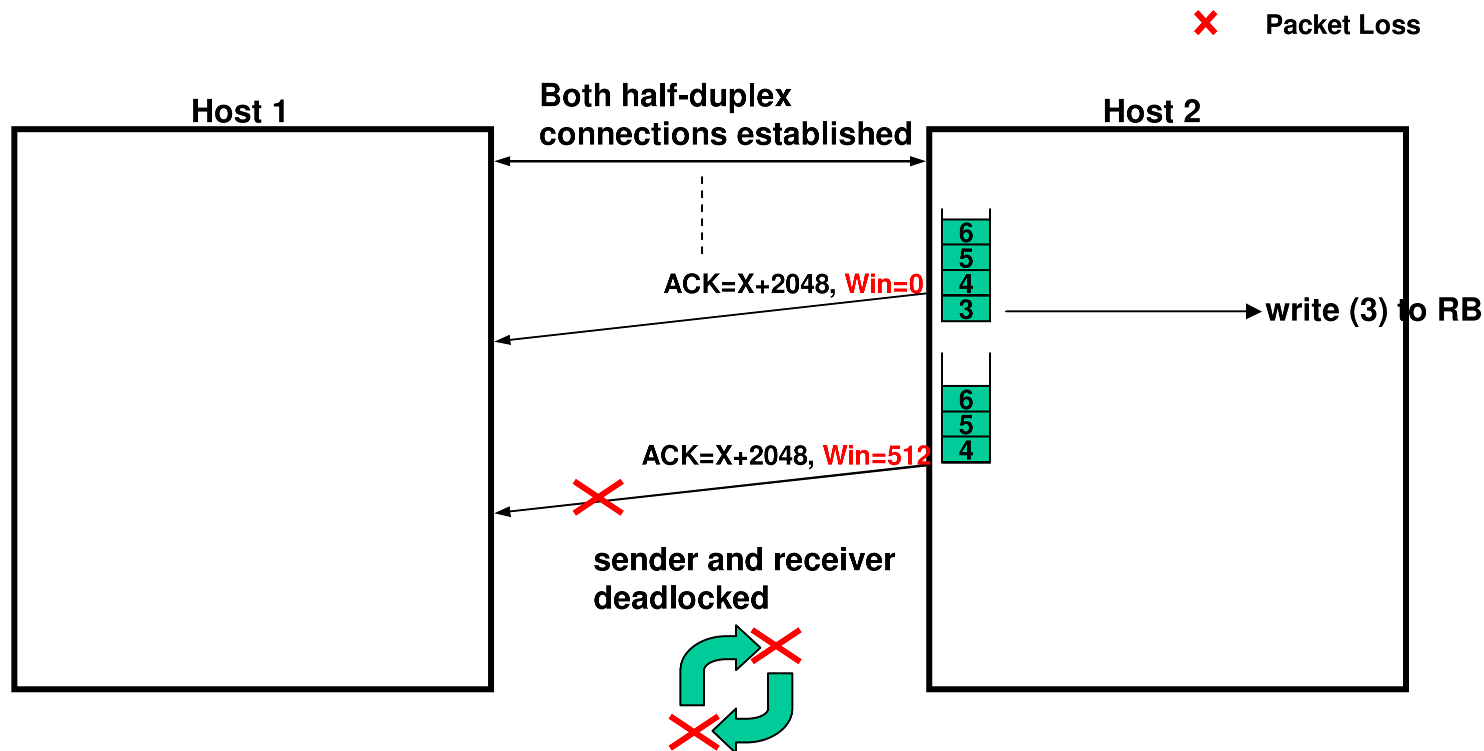


When retransmission timer expires Wc is immediately reset to 1. From there slow-start restarts normally.

© Peter R. Egli 2011

• **TCP Persist Timer (1/2):**

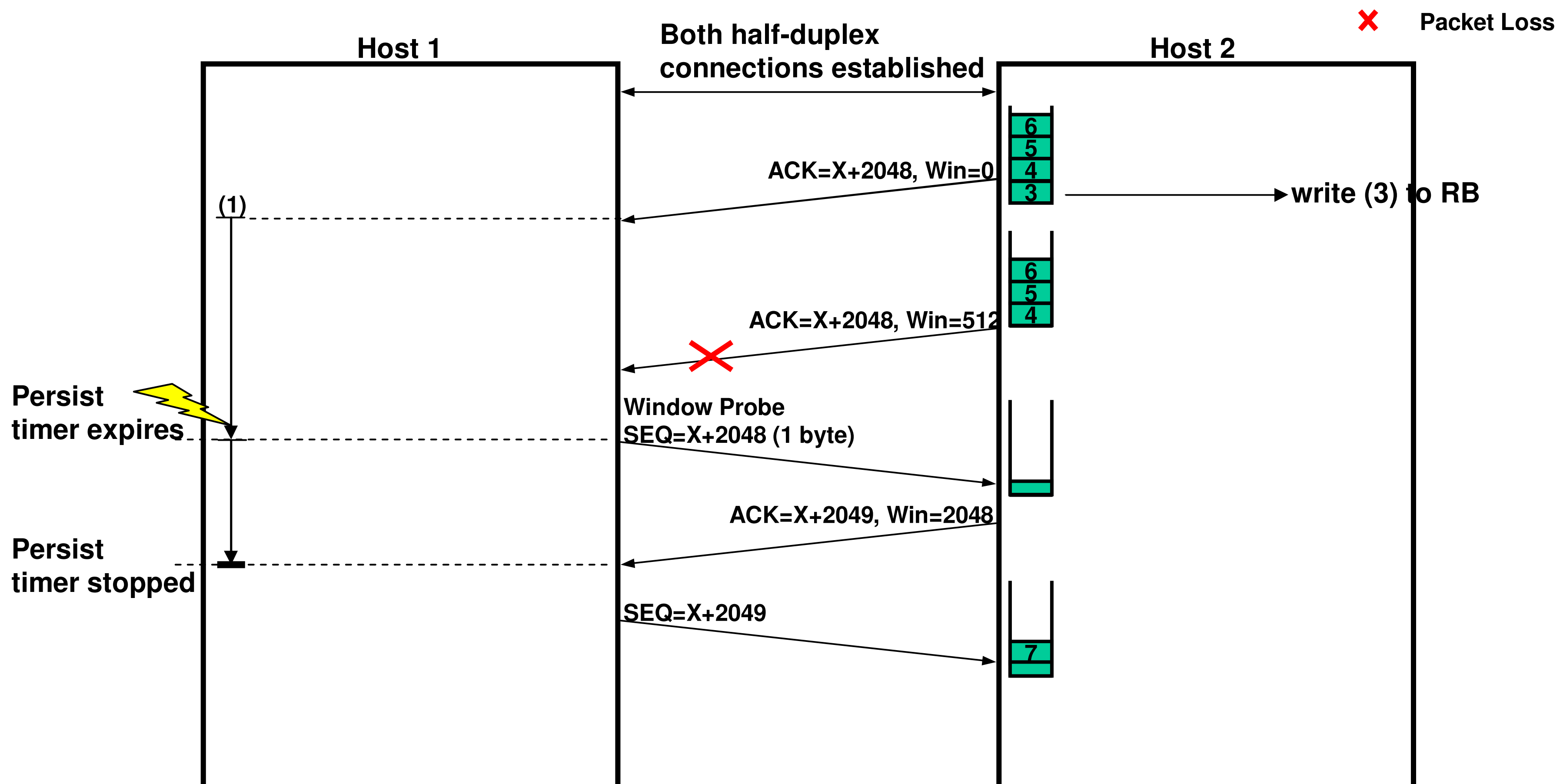
➔ **Problem: Possible deadlock situation in TCP flow control:**
The receiver buffer is full thus the receiver sends an ACK with Win=0 (window is closed). Thereafter data is read from the receiver buffer to the application. The receiver is ready again to receive data and sends an ACK with Win>0 (window is re-opened). This segment however is lost. Sender and receiver are now deadlocked: sender waits for the window to be re-opened, the receiver waits for data.



© Peter R. Egli 2011

• TCP Persist Timer (2/2):

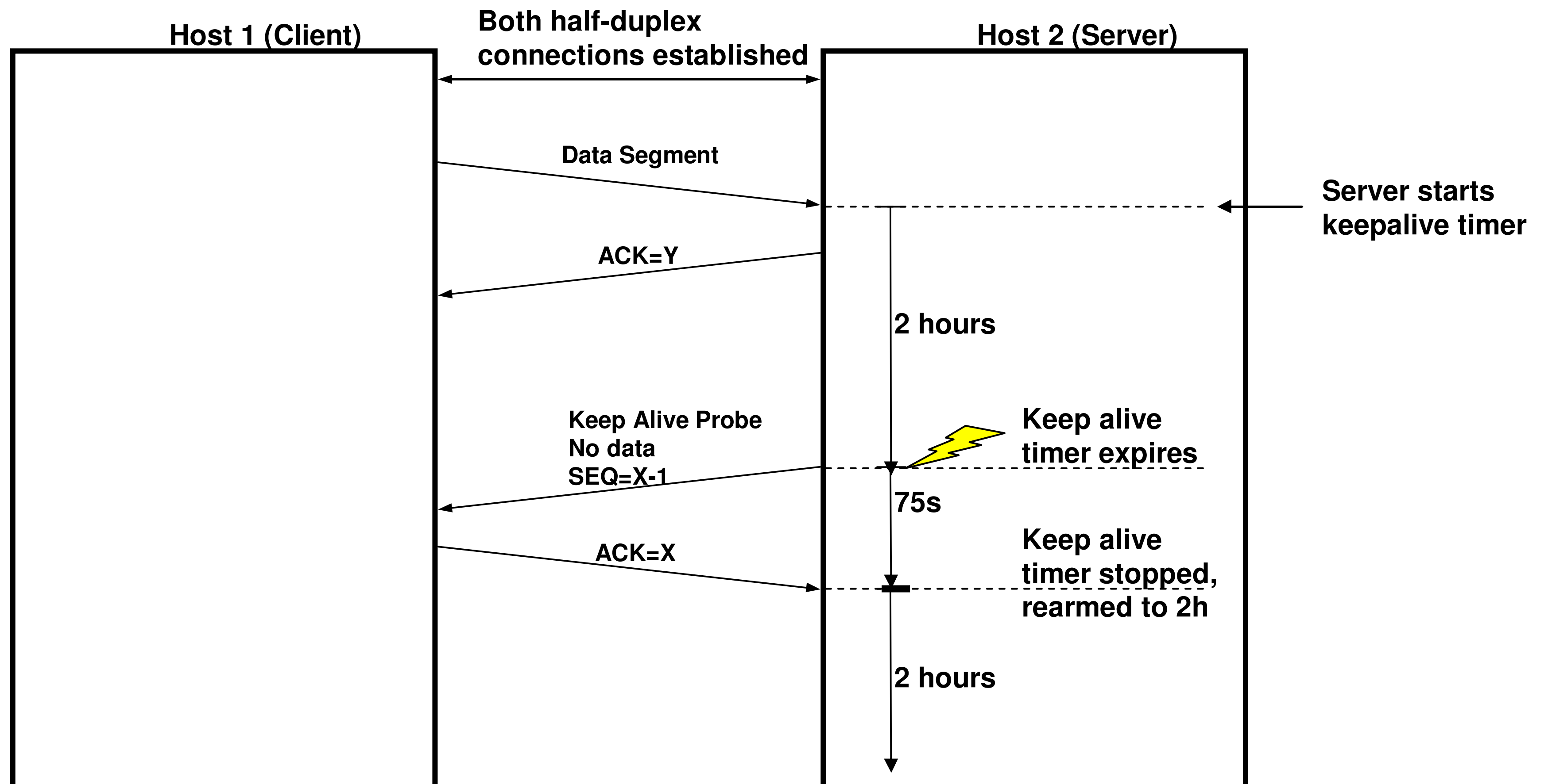
➔ Solution: The sender sends a probe segments containing 1 single byte of data to invoke the receiver to acknowledge previous bytes again.
Persist timer values are ascertained by a „exponential backoff algorithm“ that produces output values from 5s (min.) to 60 (max.) seconds.



• TCP Keepalive Timer – TCP connection supervision (1/2):

➔ The keepalive timer is used to periodically check if TCP connections are still active.

Case 1: TCP connection is still alive (i.e. client is still alive and the connection open).

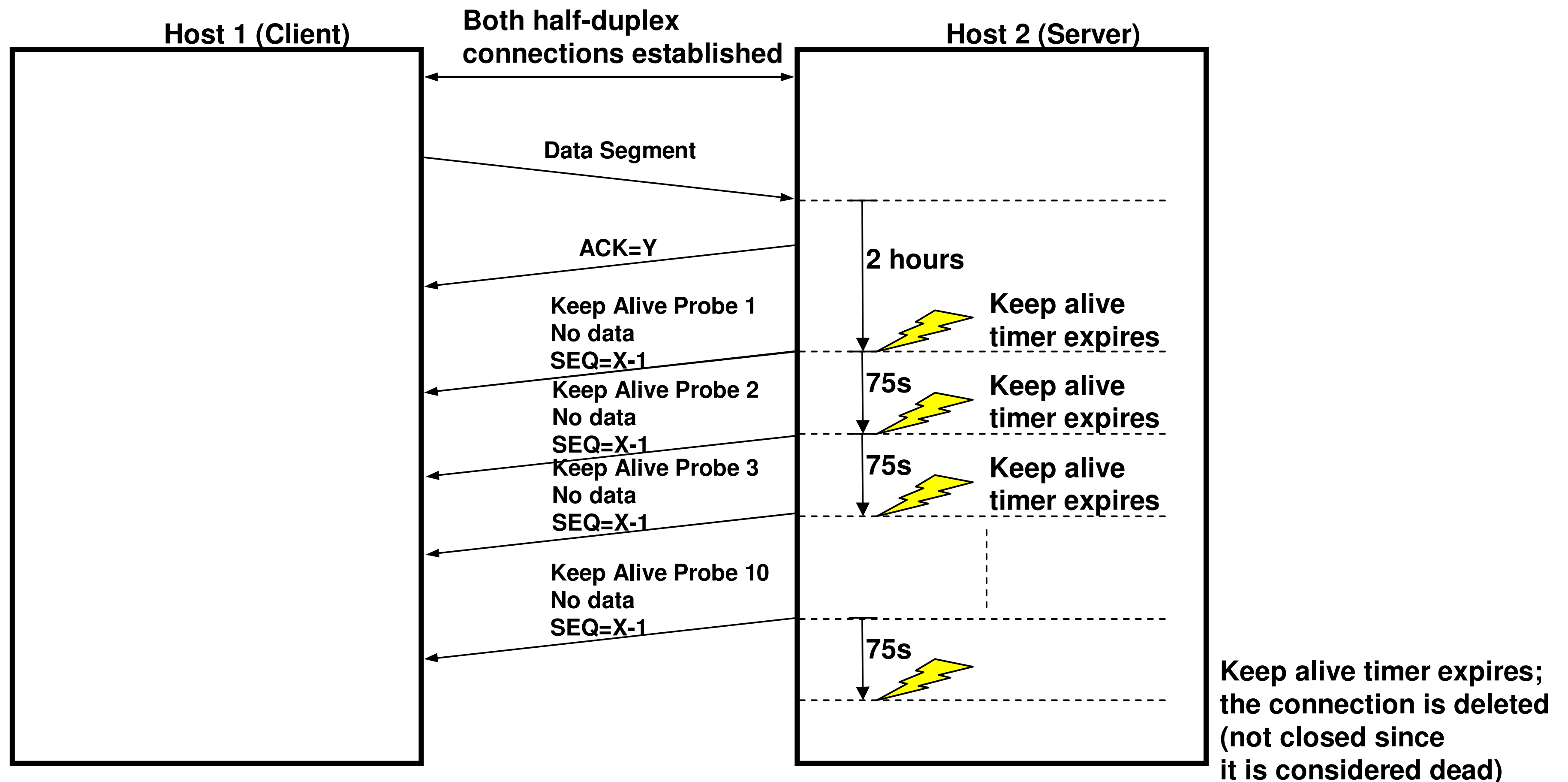


© Peter R. Egli 2011

• **TCP Keepalive Timer – TCP connection supervision (2/2):**

➔ **The keepalive timer is used to periodically check if TCP connections are still active.**

Case 2: The TCP connection dead.



• TCP Header Flags (1/2):

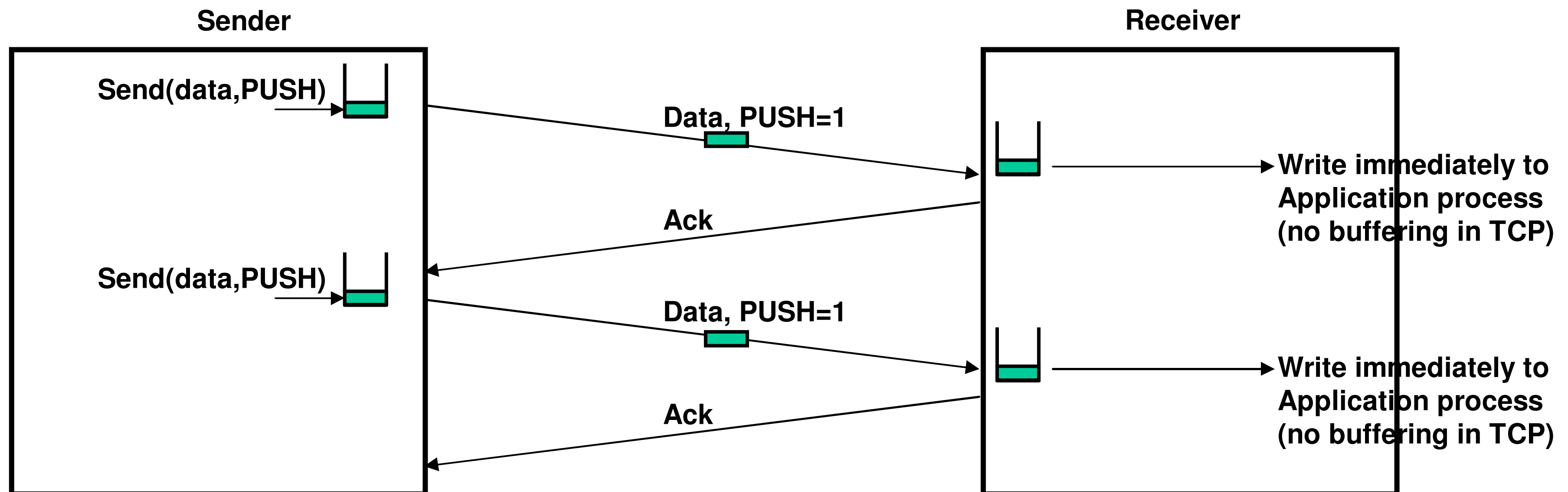
➔ **PUSH flag:**

Problem: Segment size and time of transmit are determined by TCP (flow control, congestion control, TCP does internal buffering). This is unsuitable for interactive applications like X Windows, TELNET where data should be sent as soon as possible.

➔ **Solution: With PUSH flag data can be expedited:**

Sender: Send data immediately without further buffering.

Receiver: When receiving PUSH flag „pushes“ all buffered data to the application (no further Buffering).

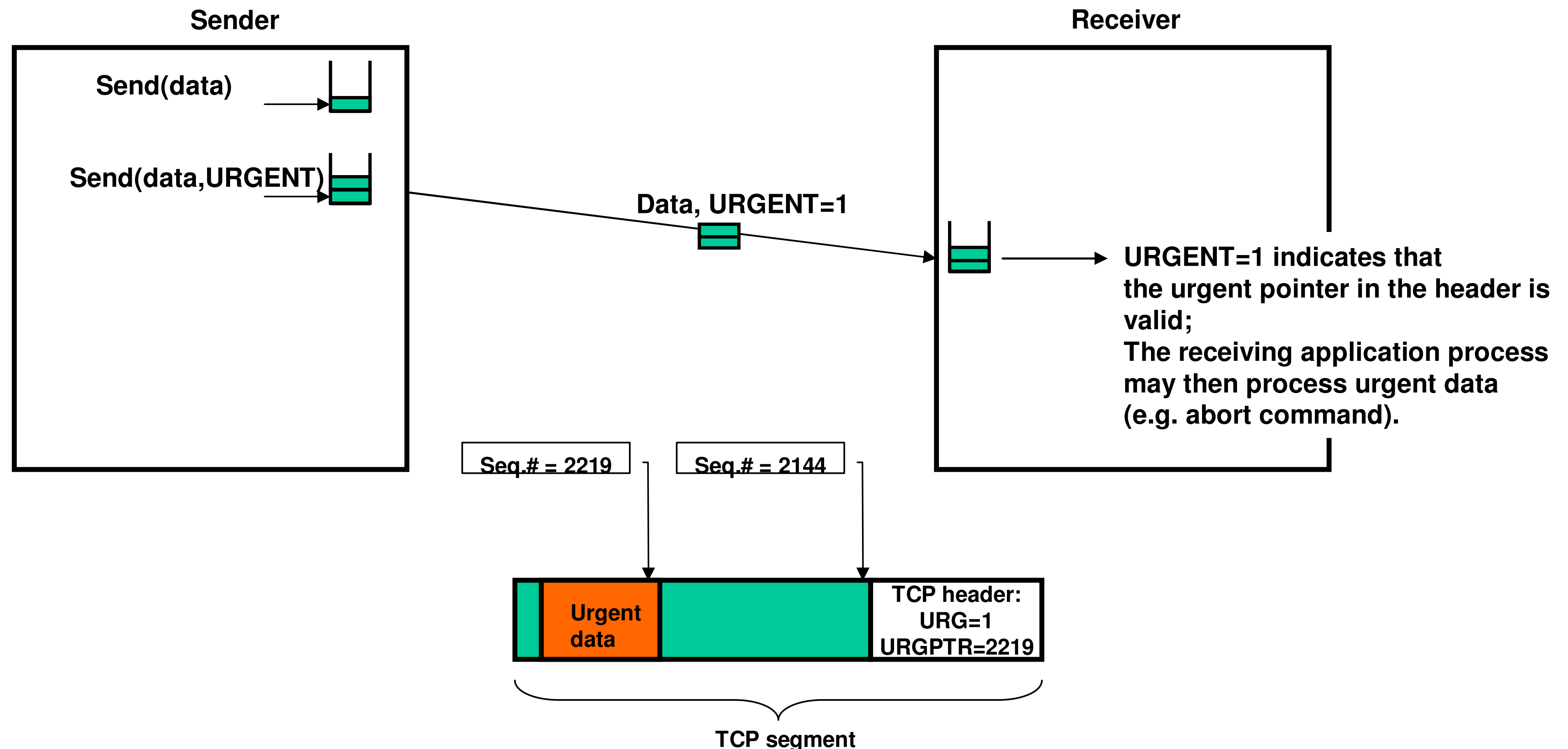


© Peter R. Egli 2011

• TCP Header Flags (2/2):

➔ URGENT flag:

This flag allows a sender to place urgent (=important) data into the send stream; the urgent pointer points to the first byte of urgent data; the receiver can then directly jump to the processing of that data; example: Ctrl+C abort sequence after other data.



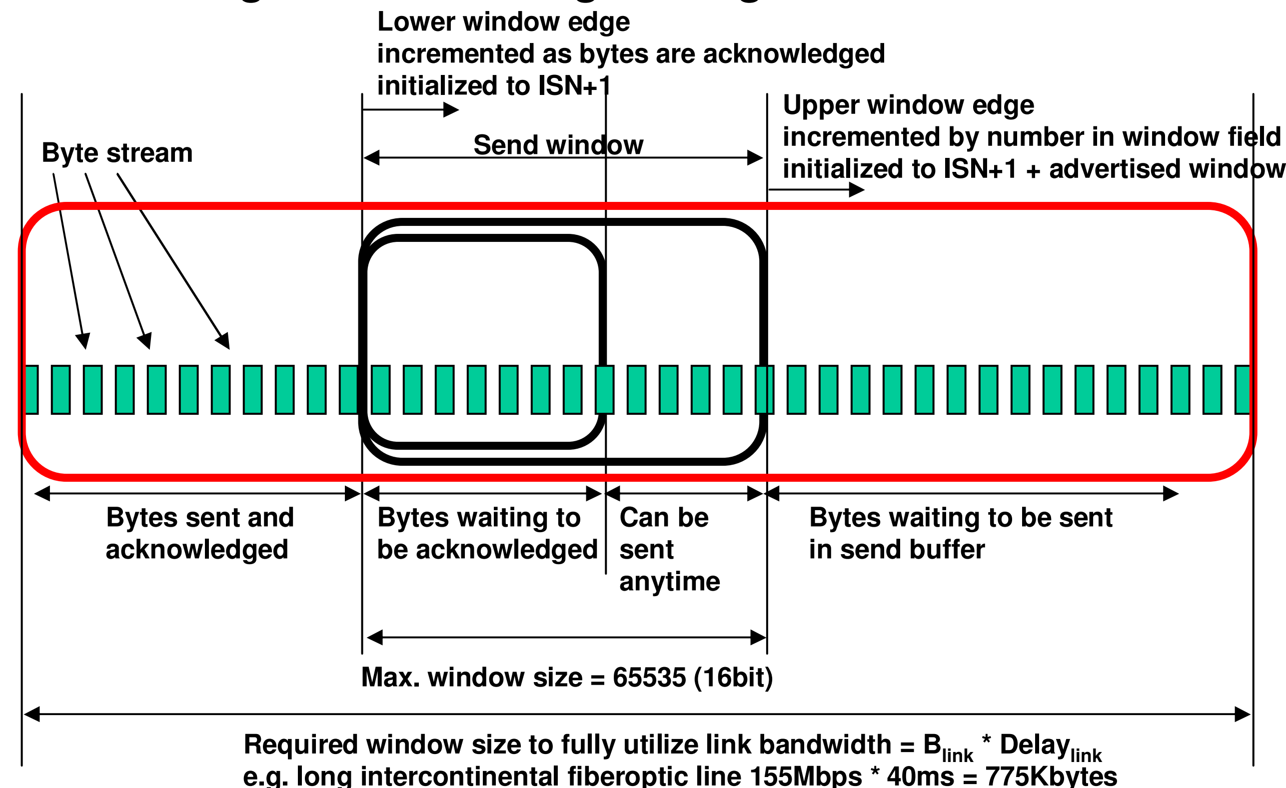
• TCP Header Options (1/3):

➔ Window scale option:

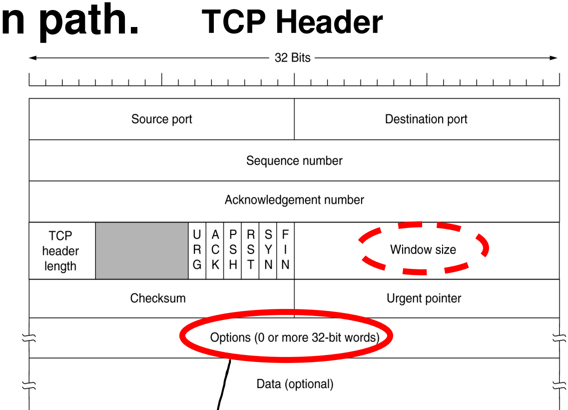
Problem: High-speed networks with long delay; network (transmission path) then stores large amounts of data („long fat pipe“).

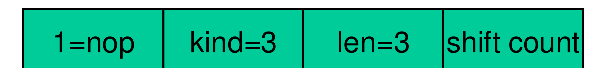
E.g. 155Mbps line with 40ms delay ➔ 775Kbytes are underway in the transmission path but the maximum window size is restricted to 65535 bytes (16 bit field).

Solution: With the window scaling option the window size can be increased (exponentially) thus making the window large enough to „saturate“ the transmission path.



TCP Header





Window size field remains unchanged, but is scaled by window scale option

E.g. shift count=3: ➔ Ws * 2^3 = Ws * 8

E.g. shift count=0: ➔ Ws * 2^0 = Ws * 1

© Peter R. Egli 2011

• TCP Header Options (2/3):

➔ Timestamp option:

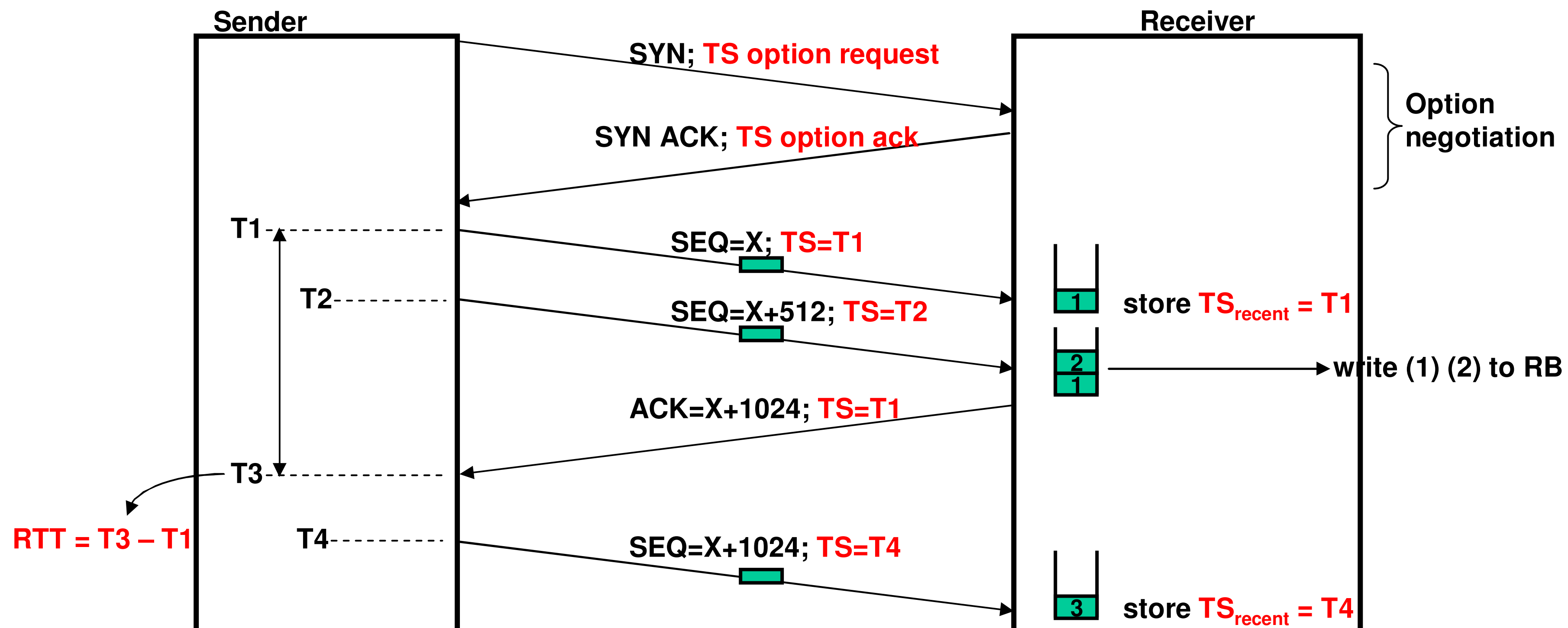
Problem: RTO calculation would require a recomputation for each TCP segment sent. Some TCP implementations recompute the RTO only once per window (less processing power required). This is ok for slower links where the sliding window contains only few TCP segments. On faster links this behavior leads to inaccurate RTO computations (and thus less-than-optimal flow control). The TCP timestamp option allows to accurately ascertain the RTO.

Enable timestamp and window scale option on Windows:

Registry: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Tcp1323Opts = 3

Timestamp option in TCP header

1=nop	1=nop	kind=8	len=10
timestamp value			
timestamp echo reply			



© Peter R. Egli 2011

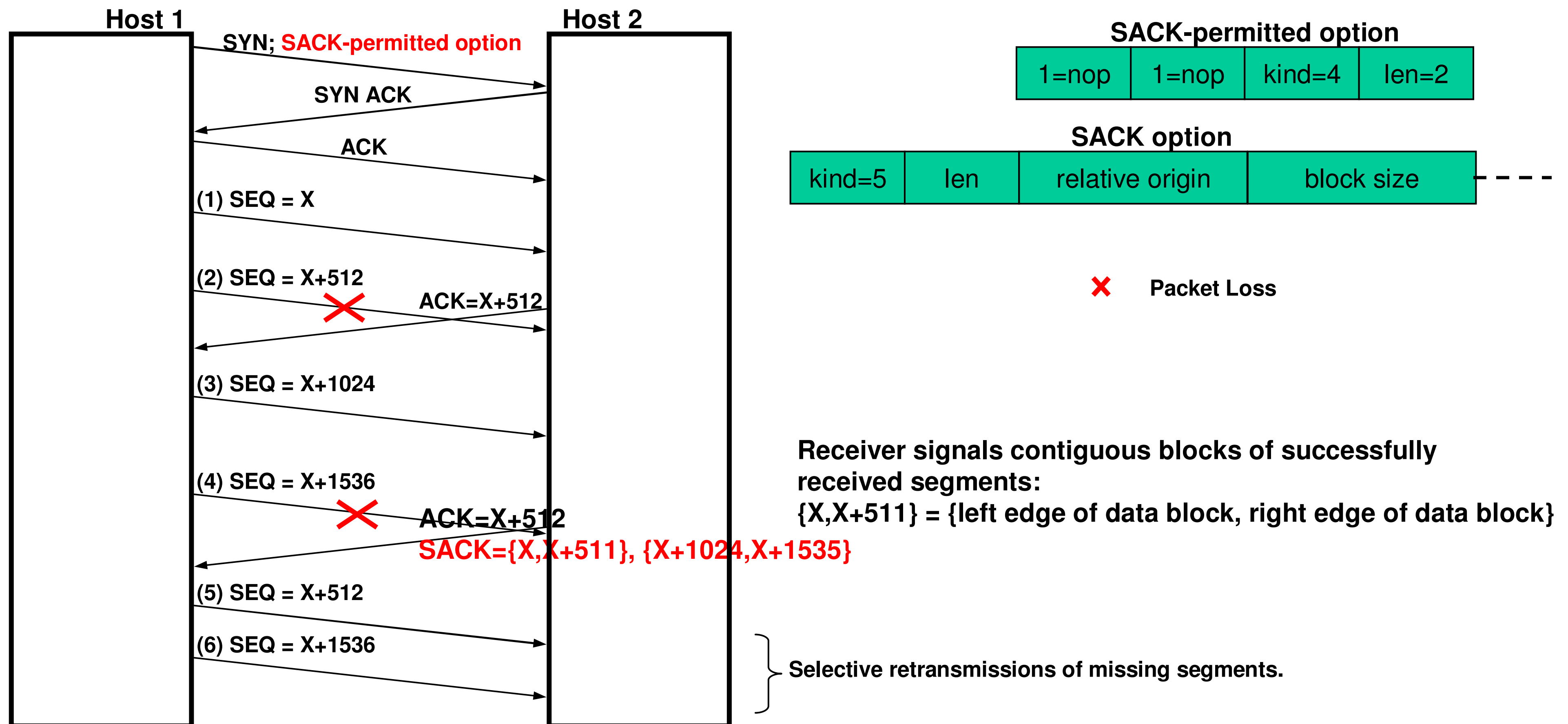
TCP - Transmission Control Protocol – RFC793

indigoo.com

• TCP Header Options (3/3):

➔ **SACK permitted option (selective retransmissions):**

Receiver can request sender to retransmit specific segments (due to loss).
The SACK does not change the meaning of the ACK field; if the SACK option is not supported by the sender TCP will still function (less optimally though in case of retransmissions).



© Peter R. Egli 2011

• **TCP Throughput / performance considerations („goodput“) (1/2):**

➔ **TCP is delay sensitive!**

max. throughput = Ws / RTT **[Padhye 98]**

This is not an exact formula and it is valid only for average RTT values.

The maximum throughput is bound by the window size Ws and decreases with increased RTT (=delay). TCP as such is NOT suited for networks with long delays (e.g. satellite and interplanetary links).

➔ **TCP is not independent of the underlying network (as should be the case in theory)!**

TCP was designed to run over wired networks (low Bit Error Rate BER, packet loss due to congestion). TCP performs badly on radio links (high BER, packet loss due to errors).

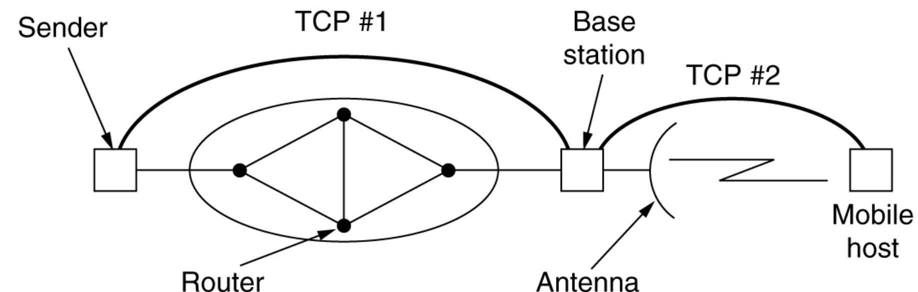
In case of packet loss on a radio link the sender should try harder instead slowing down. Slowing down just further decreases the throughput.

On wired networks the sender should slow down in case of packet loss (caused by congestion) in order to alleviate the problem.

How to handle TCP connection that spans a wired and a radio link?

➔ **„Split TCP“: Effectively 2 separate TCP connections interconnected by the base station passing TCP payload data between the connections. Each of the TCP connections is optimized for their respective use. Example:**

TCP accelerator devices for satellite links.



• TCP Throughput / performance considerations („goodput“) (2/2):

TCP adaptations for radio (satellite) links as per RFC2488, RFC3481:

a. Data link protocol:

Use data link protocols that do not do retransmissions and flow control or make sure that retransmission mechanism and flow control of data link do not negatively affect TCP performance.

b. Path MTU discovery:

Path MTU discovery enables TCP to use maximum sized segments without the cost of fragmentation/reassembly. Larger segments allow the congestion window to be more rapidly increased (because slow start mechanism is segment and not byte based).

c. Forward Error Correction:

TCP assumes that packet loss is always due to network congestion and not due to bit errors. On radio links (satellite, microwave) this is not the case – packet loss is primarily due to bit errors. The congestion recovery algorithm of TCP is time consuming since it only gradually recovers from packet loss. The solution is to use FEC to avoid false signals as much as possible. FEC means that the receiver can detect and correct bit errors in the data based on a FEC code.

d. Selective ACKs (SACK):

The TCP algorithms „Fast Retransmit“ and „Fast Recovery“ may considerably reduce TCP throughput (because after a packet loss TCP gingerly probes the network for available bandwidth; in a radio environment this is the wrong strategy). To overcome this use selective ACKs (SACK) option to selectively acknowledge segments.

e. Window scale option:

Long delay in fast transmission links limits TCP throughput because large amounts of data are underway but TCPs sliding window only allows up to 65536 bytes to be „in the air“. To overcome this use the window scaling option thus increasing the size of the sliding window.

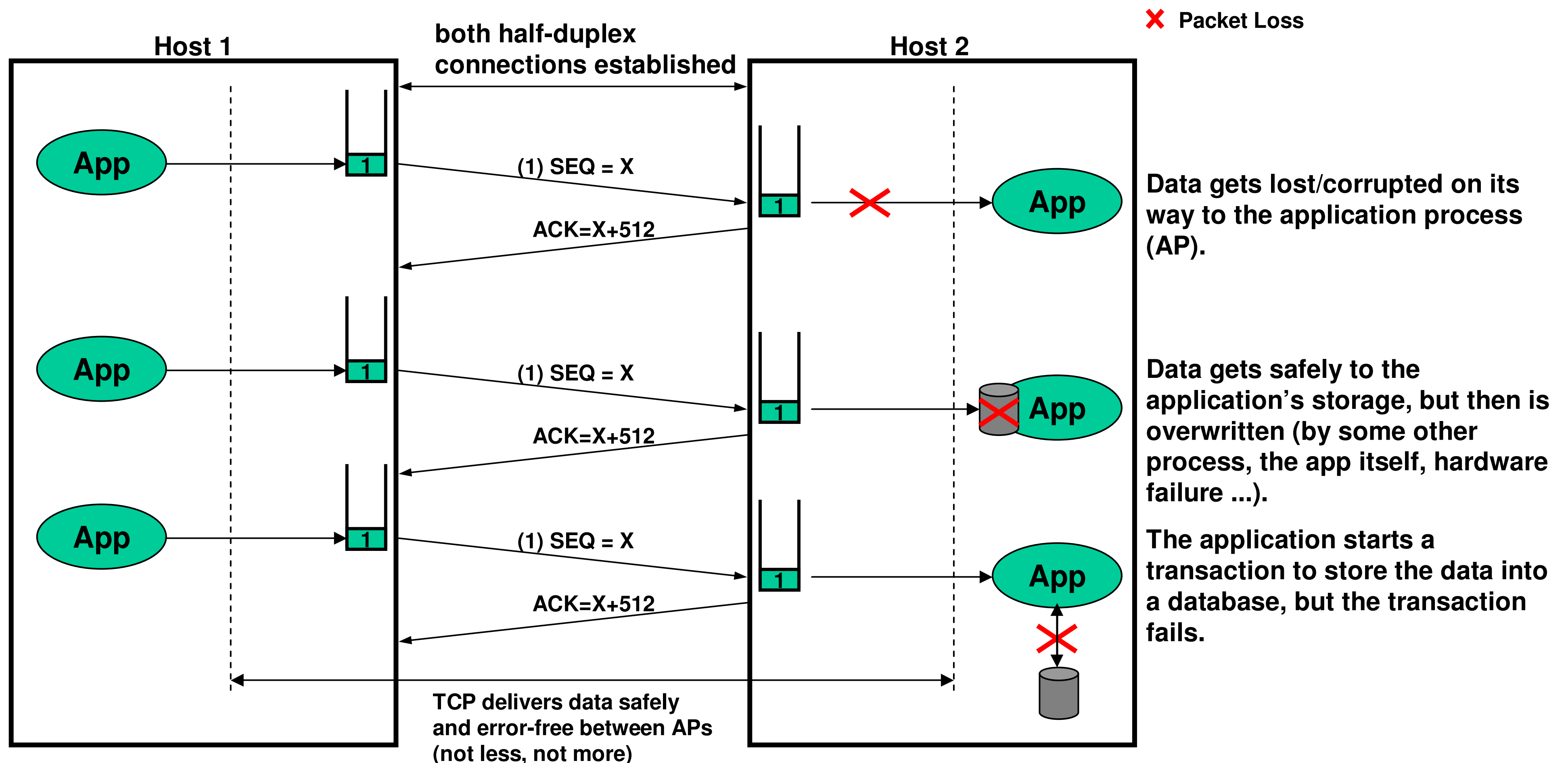
© Peter R. Egli 2011

TCP - Transmission Control Protocol – RFC793

indigoo.com

- **A last word on „guaranteed delivery“:**

➔ **TCP provides guaranteed transmission between 2 APs. That‘s it. There are still zillions of reasons why things can fail! Thus it is still up to the application to cater for application error detection and error handling (see RFC793 chapter 2.6 p.8).**



© Peter R. Egli 2011