TCP Segments

TCP Segments –

Before TCP software sends bytes/data out.

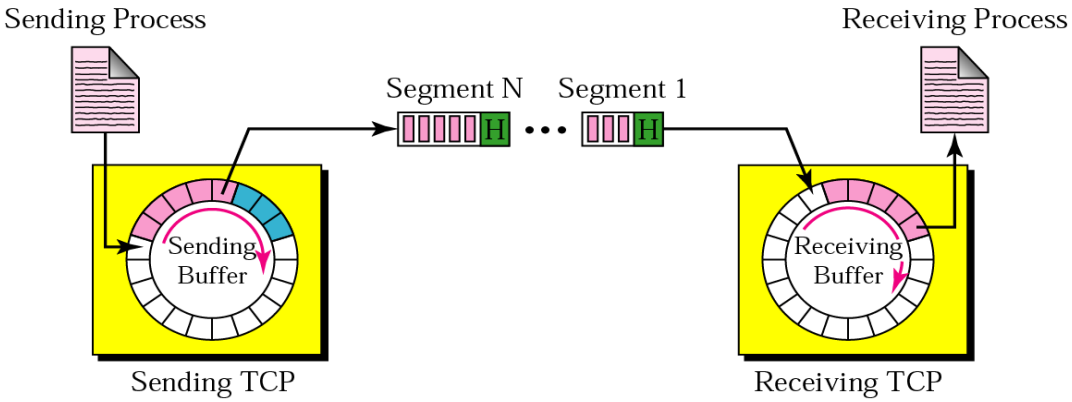
TCP groups a number of bytes together into a segment, and adds a header, before handing them over to IP layer

- **IP can transmit only packets of data, not streams of bytes!**
- **TCP segments are not necessarily of the same size – TCP segment size depends on many factors including:**

  (a) should not be too small – TCP+IP overhead alone = 20+20 = 40 bytes

  (b) should not be too large: 1) max IP = 65,535 bytes,
  2) MTU limit can also additionally be taken into consideration

  (c) should not overwhelm receiving buffer nor network
  [sliding adjustable-size sender window mechanism, slide 7]



http://aboelela.site/profile/projects/net-seal/animations/tcp-flow-control/

TCP Segments (cont.)

TCP Numbering Principles – TCP keeps track of transmitted and received segments using two parameters: sequence number and acknowledgment number

- both refer to byte number not sequence number
- numbering is independent in each direction

Sequence Number – defines the number associated with the first byte contained in that segment

- byte numbering starts with a randomly generated 32-bit number – not (necessarily) 0

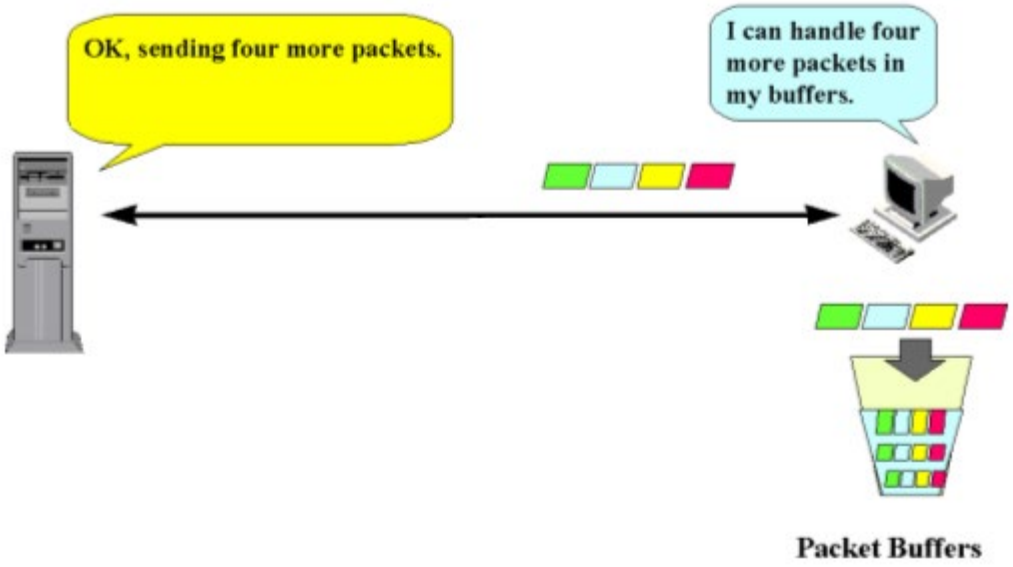
Acknowledgment Number – defines the number of the next byte that the receiving party expects to receive

- acknowledgment numbering is cumulative – the receiver takes the number of the last received byte, adds 1 to it, and announces this sum as the acknowledgment number

TCP Flow Control

Network Flow Control – regulates the amount of data a source can send before receiving an acknowledgment from the destination

- extreme case 1: send byte by byte ⇒ to slow
- extreme case 2: send all data ⇒ overwhelm receiver
- solution: speed matching service – rate at which sender application is sending is matched against rate at which receiving application is receiving

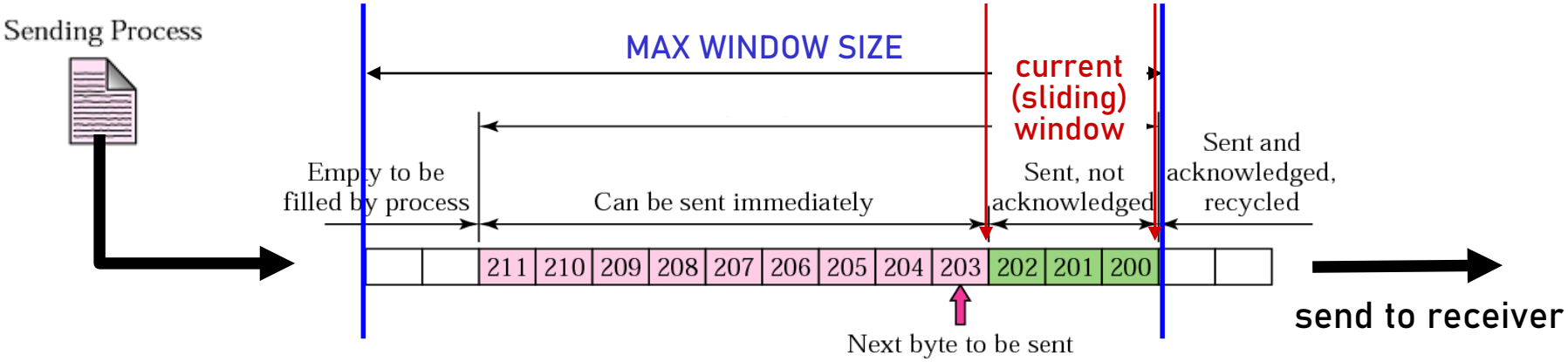


TCP Flow Control (cont.)

TCP Flow Control – controls amount of buffer data that sender can send to receiver at any particular time, using 'sliding window' principle

- **SLIDING WINDOW** – spans a portion of sending buffer containing bytes that host can send, or have sent, to other side without worrying that it might overwhelm the receiver (i.e., unacknowledged bytes)
- window's left pointer moves as host sends more bytes
- window's right pointer moves as host receives new acknowledgment

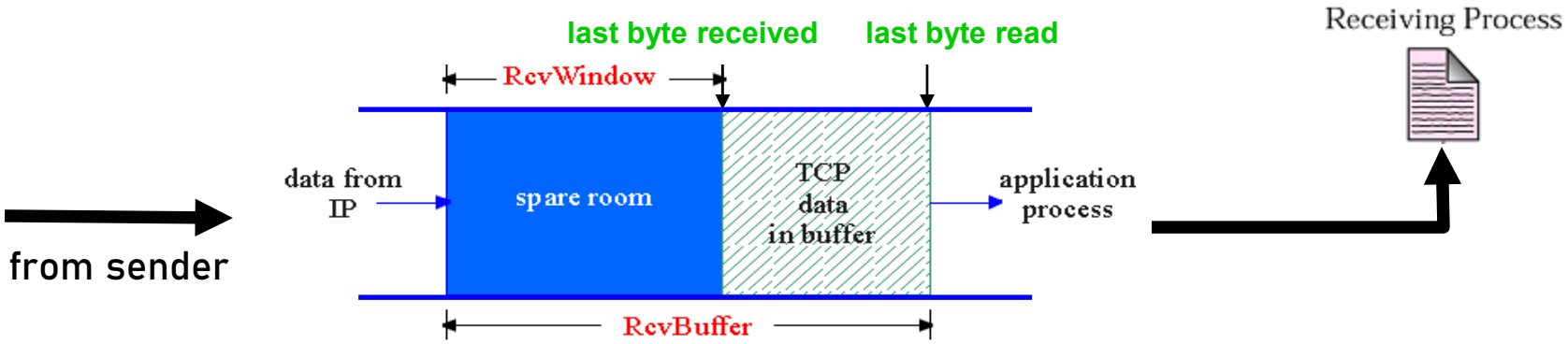
What is max size of sender's sliding window?! How is it determined?!



TCP Flow Control (cont.)

Receive Window – variable used to give sender an idea of how much free buffer space is available at the receiver

- during connection establishment receiver allocates a receive buffer – buffer size = RcvBuffer
- as receiver receives bytes that are correct and in order, it places them in receive buffer ⇒ spare room in buffer = RcvWindow = RcvBuffer – [Last Byte Received – Last Byte Read]
- receiver advertises RcvWindow in 'window size' field of TCP headers
- RcvWindow can be changed at any point during communication - 'window size' field is adjusted accordingly



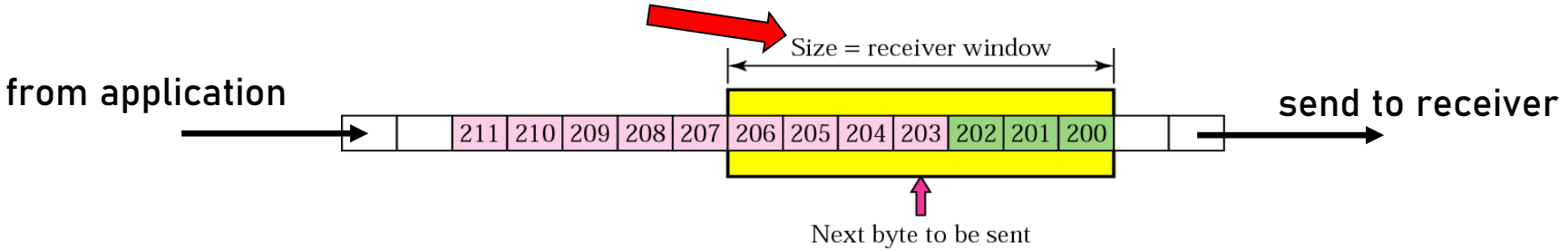
receiver buffer

TCP Flow Control (cont.)

Sender Window – **sliding window over sender buffer – must be of receive window (RcvWindow) size!**

… upon learning senders RcvWindow value

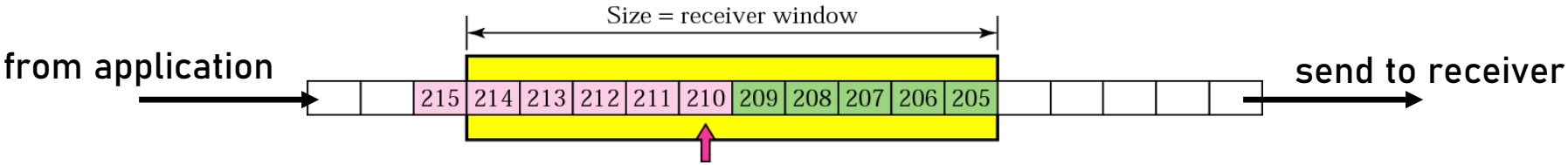
- **by keeping amount of unacknowledged data ≤ RcvWindow sender is assured that it is not overflowing the receiver**



sender buffer

Example **[expending sender window]**

If receiver's application reads data faster than it comes in ⇒ RcvWindow increases ⇒ new RcvWindow value can be relayed to sender resulting in expansion of sender's max sliding window size. And, vice versa …



sender buffer

TCP Flow Control (cont.)

Example [TCP window size adjustment]

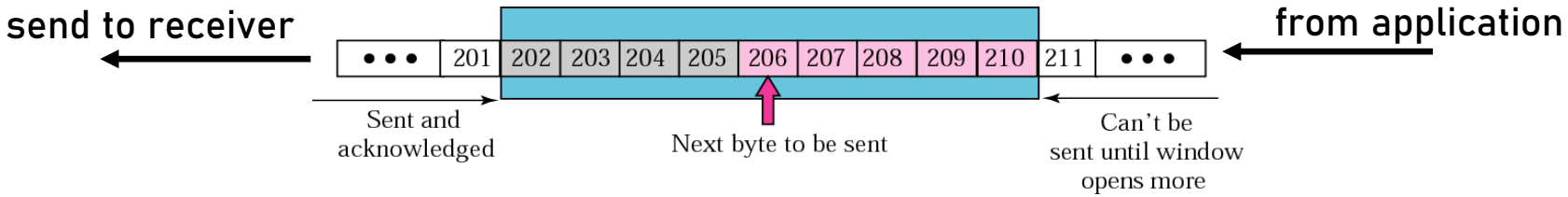
What is the value of the receiver window (RcvWindow) for sender host A if the receiver, host B, has a buffer of size 5,000 and 1,000 bytes of received and unprocessed data?

Solution:

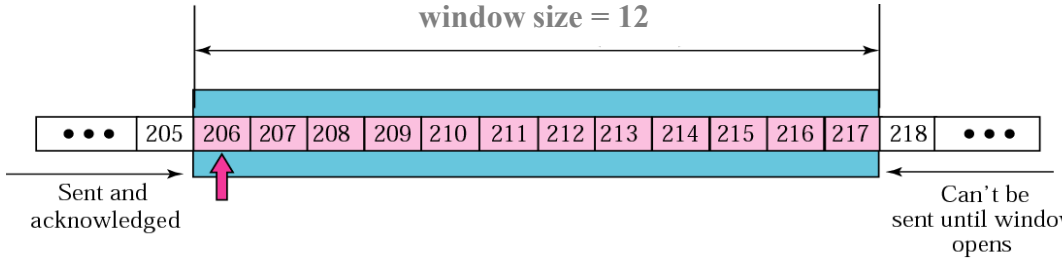
RcvWindow = 5,000 – 1,000 = 4,000. Host B can receive only 4,000 bytes of data before overflowing its buffer. Host B advertises this value in its next segment to A.

Example [TCP window size adjustment]

In the figure below, the sender receives a packet with an acknowledgment value of 206 and RcvWindow of 12. The host has not sent any new bytes. Show the new window.



Solution:



TCP Flow Control (cont.)

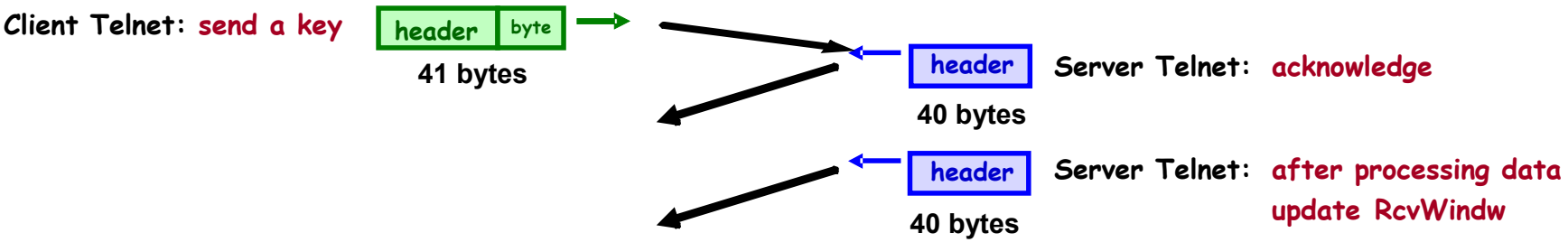
Silly Window Syndrome – (SWS) phenomenon occurs as a result of

1) a transmitter immediately sends out very small amounts of data – **SWS by Sender** (bad for network)

2) recipient advertises window sizes that are too small – **SWS by Receiver** (bad for receiver)

- **basic TCP sliding window technique does NOT set minimum size on transmitted segments** ⇒ this can result in a situation where many small inefficient segments are sent, rather than a smaller number of larger ones

SWS by Sender – occurs if sending application creates data slowly (e.g. 1 byte at a time), and TCP immediately sends data ⇒ **considerable network overhead**! (20 byte IP + 20 byte TCP)



TCP Flow Control (cont.)

Example [SWS by Sender]

Suppose a TCP client is receiving data from the sending application in blocks of 10 bytes. How long should the client wait before sending data?

(a) send right away ⇒ 10 bytes data + 40 bytes TCP/IP header ⇒ 400% overhead

(b) wait for more data ⇒ may have to wait too long ⇒ considerably applicat. delay

SWS by Sender Avoidance – Nagle's Solution:

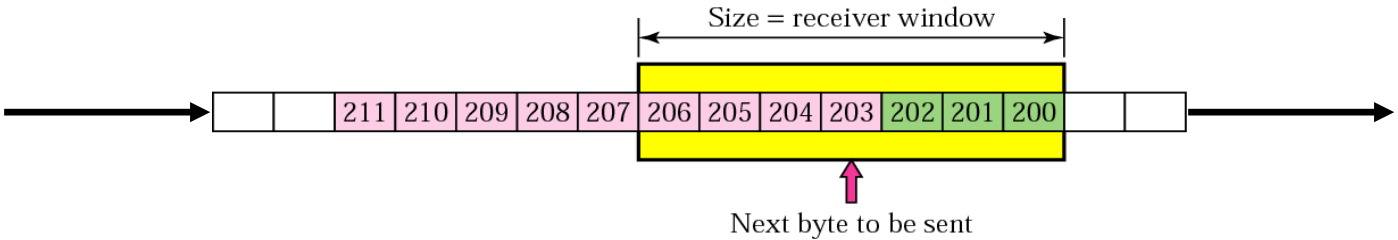
(a) as long as there is no un-ACK data outstanding in the buffer, data can be immediately send – even if only 1 byte

(b) while there is un-ACK data, all subsequently segments are held in the buffer until

(b.1) previous segment is acknowledged (fast network)

(b.2) enough data have accumulated to fill a max-size segment (slow network)

- **advantage: algorithm self-adjusts to traffic load**
- **disadvantage: should be disabled in interactive applications**

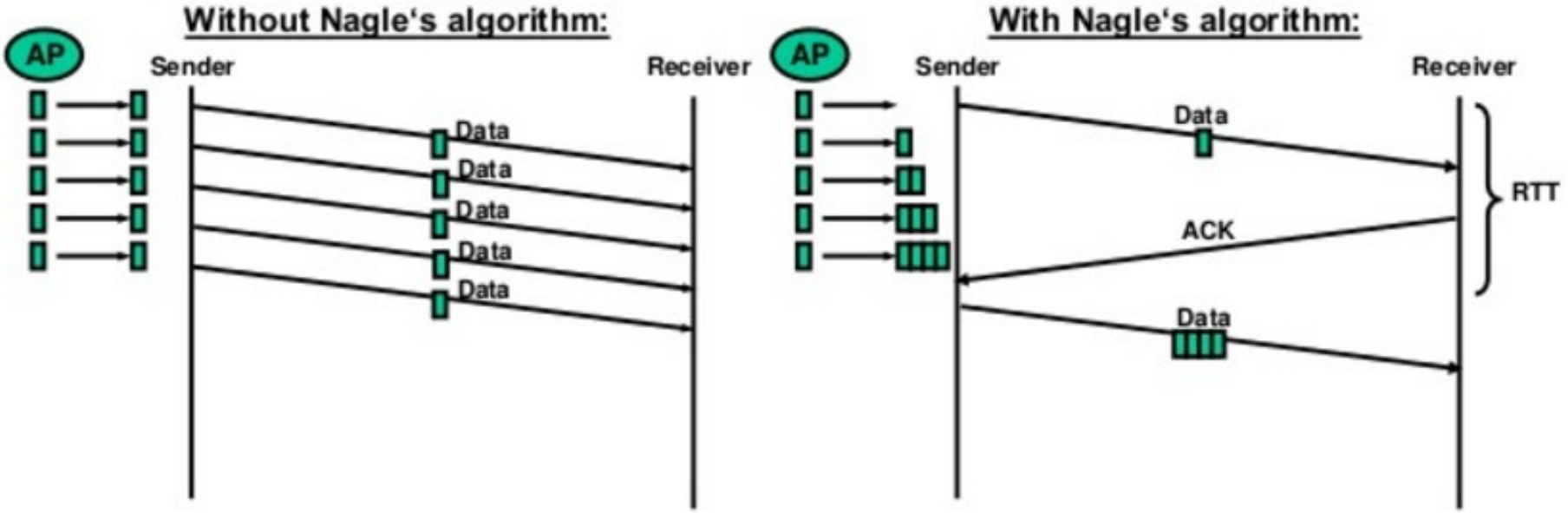


6. TCP Flow Control (9/10)

➔ Nagle's Algorithm for further optimization:

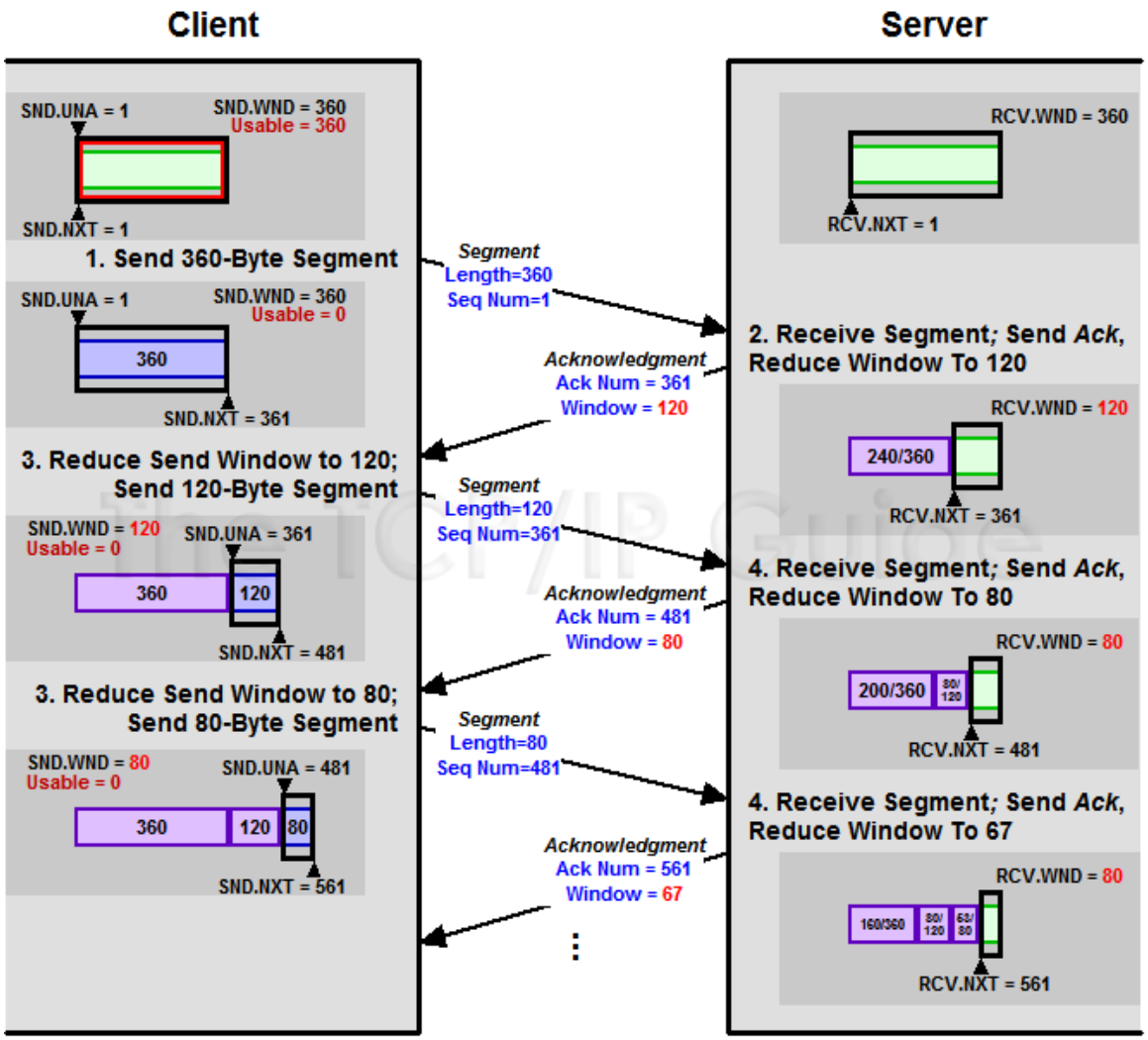
In interactive applications (Telnet) where single bytes come in to the sender from the application sending 1 byte per TCP segment is inefficient (98% overhead).

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.



TCP Flow Control (cont.)

Example [SWS by receiver]



This diagram shows one example of how the phenomenon known as TCP *silly window syndrome* can arise.

The client is trying to send data as fast as possible to the server, which is very busy and cannot clear its buffers promptly.

Each time the client sends data the server reduces its receive window.

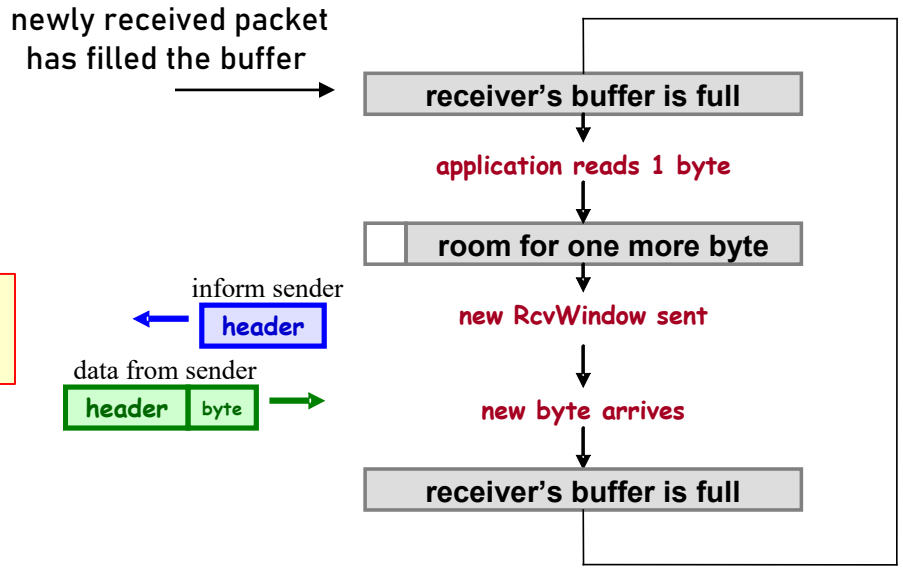
The size of the messages the client sends shrinks until it is only sending very small, inefficient segments.

source: http://www.tcpipguide.com/free/t_TCPSillyWindowSyndromeandChangesTotheSlidingWindow.htm

TCP Flow Control (cont.)

SWS by Receiver:

Receive Window Size = 1 byte



2 headers exchanged for 1 byte of data !!!

SWS by Receiver Avoidance

1) **Clark's Solution**: send an ACK as soon as data arrives, but! with RcvWindow = 0 until
   (a) at least half of the receive buffer is empty, or
   (b) enough space to accommodate a maximum size segment
   (default value - 536 bytes, can be changed in TCP options)

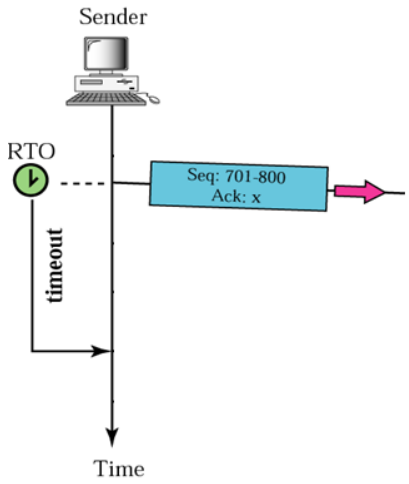
2) **Delayed Acknowledgments**: do NOT ACK segments immediately – wait until decent amount of space available
   - **advantage**: less traffic than with Clark's Solution
   - **disadvantage**: delayed ACKs may force sender to retransmit

TCP Error Control

TCP Error Control – TCP delivers entire stream to application program on the other end reliably

- no corrupted segments
- no lost segments
- all segments in order
- no duplicated segments

Special Tools for TCP Error Control



(a) **checksum** – enables detection of corrupted segments on receiver's side

(b) **sequence numbers** – enable detection of lost, out-of-order or duplicate segments on receiver side

(c) **acknowledgments**

(d) **timers**

} enable error control on sender side

- TCP sender starts one timer for each segment sent; if corresponding ACK does not arrive before timer expires segment is considered lost or corrupted and must be retransmitted

TCP Error Control (cont.)

Detection of Corrupted, Lost & Out-Of-Order Segments

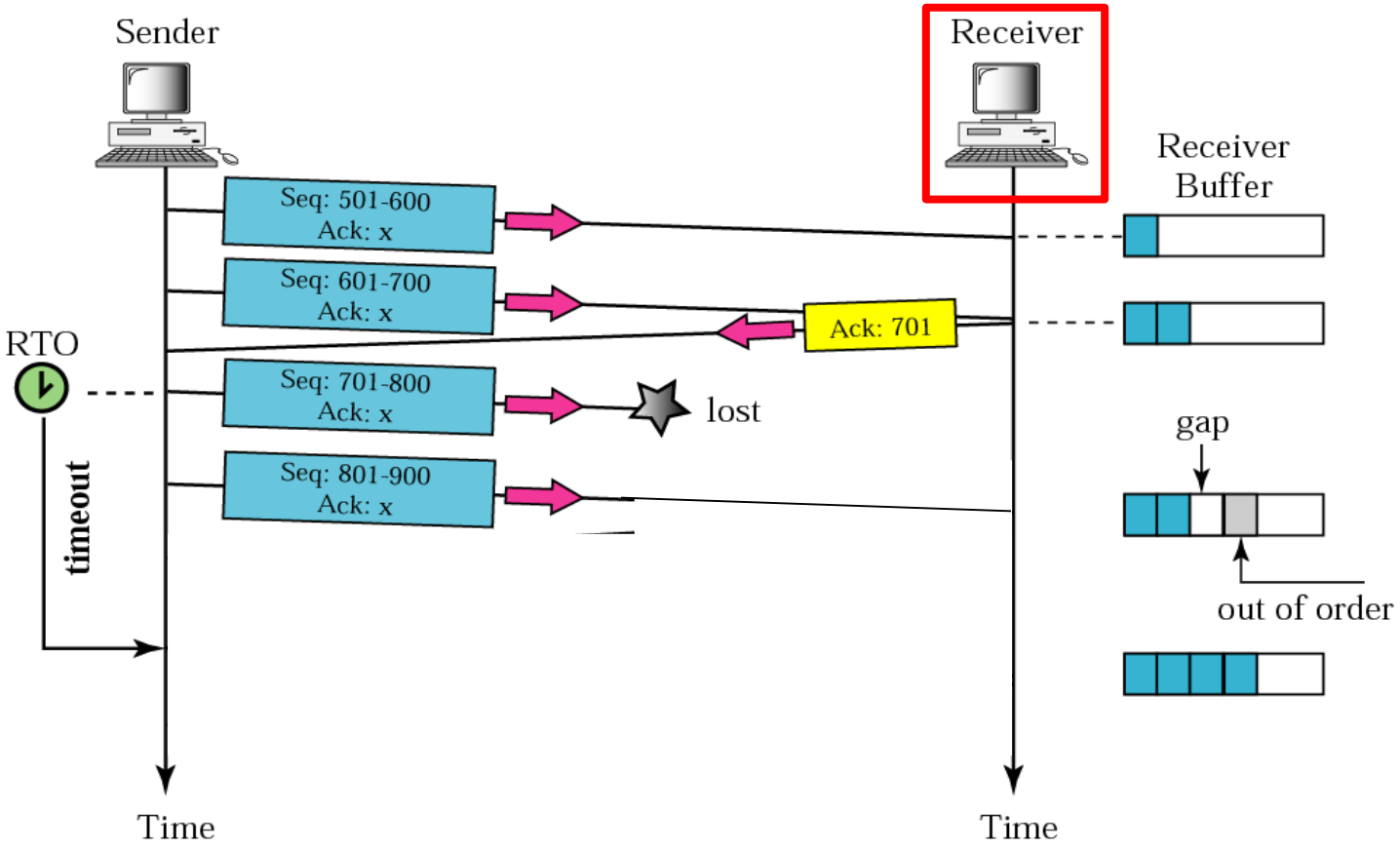
1) **Receiver** Detects

(a) corrupted segments - immediately using checksum

- action: discard corrupted segment

(b) lost and out-of-order segments by receiving seg. with higher sequence numbers than expected

- action: discard or keep?

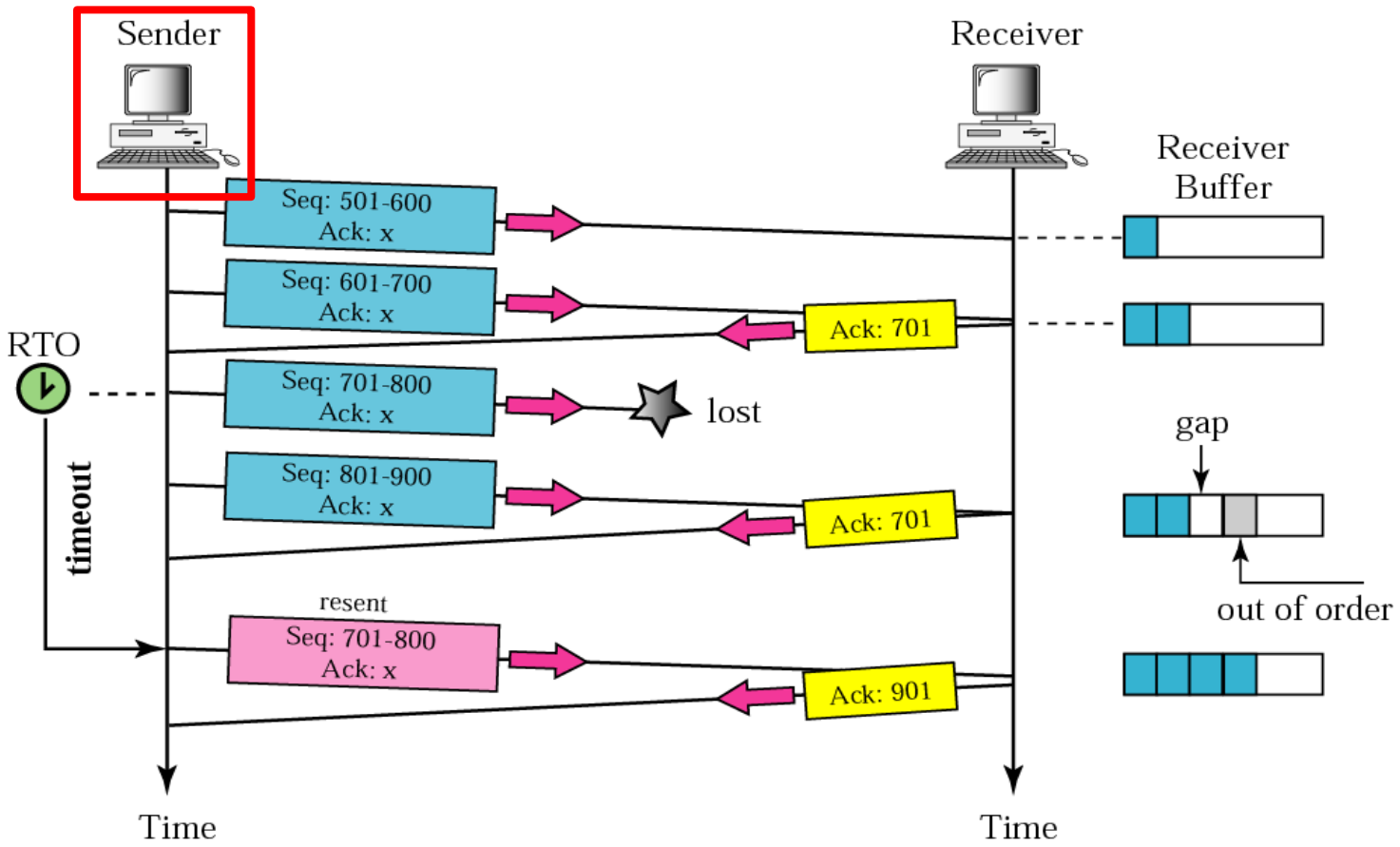


TCP Error Control (cont.)

Detection of Corrupted, Lost & Out-Of-Order Segments

2) **Sender** Detects corrupted, lost and out-of-order segments, by NOT receiving their ACKs within **Time-out** interval

- **action**: retransmit

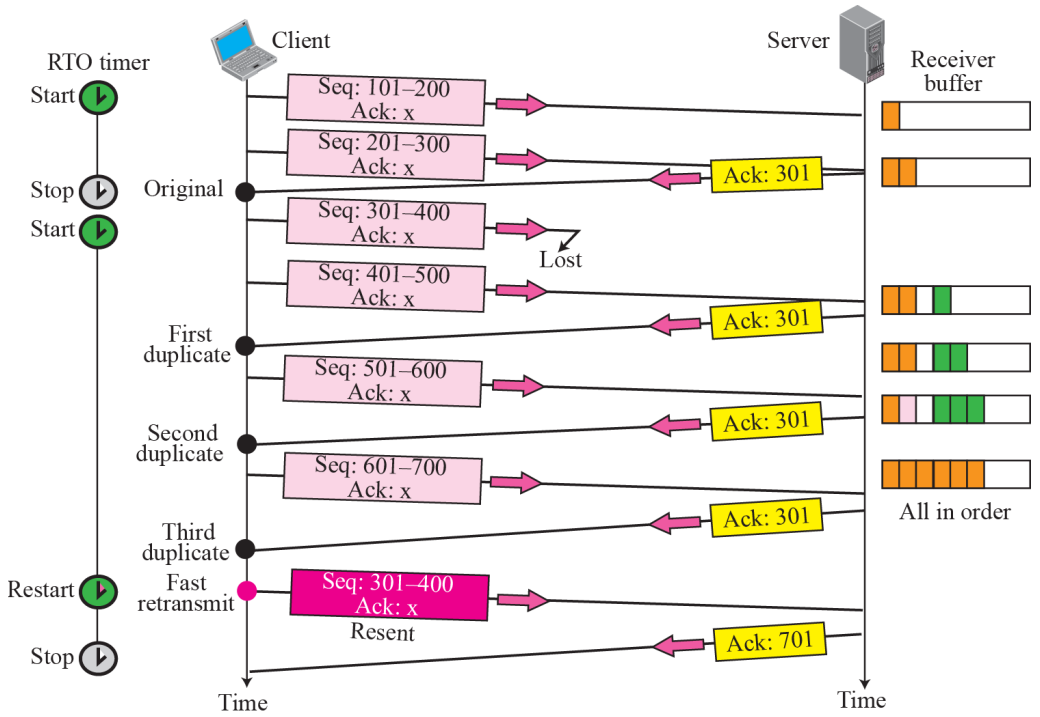


TCP Error Control (cont.)

Receiver Action on Detecting Lost and Out-of-Order Segments: Fast Retransmit

- simply discard segments – ineffective
- keep segments but do not send any ACK – ineffective
  (must wait entire timeout to receive missing segment)
- keep segment and re-acknowledge last in-order byte received ⇒ enables **fast retransmit**

Example [TCP error control – fast retransmit]



(a) receiver - on every out-of-order seg. acknowledge the last in-order byte received

(b) sender - in case that 3 duplicate ACKs are received immediately retransmit the missing segment (fast retransmit) even if the segment's timer has not expired; then proceed with fast recovery ...

Why should we wait for (at least) 3 duplicate ACK before initiating 'fast recovery'?!

TCP Error Control (cont.)

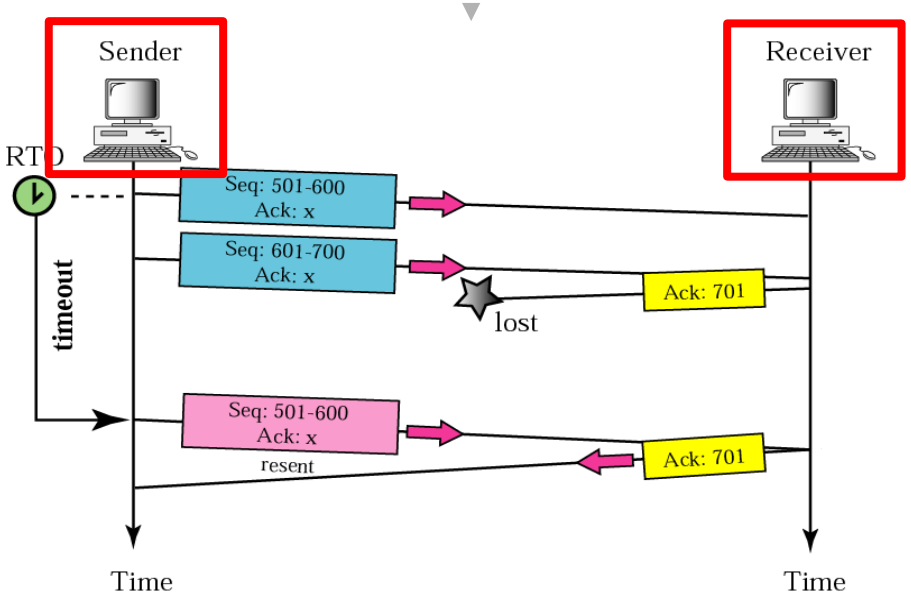
Detection of Lost Acknowledgments

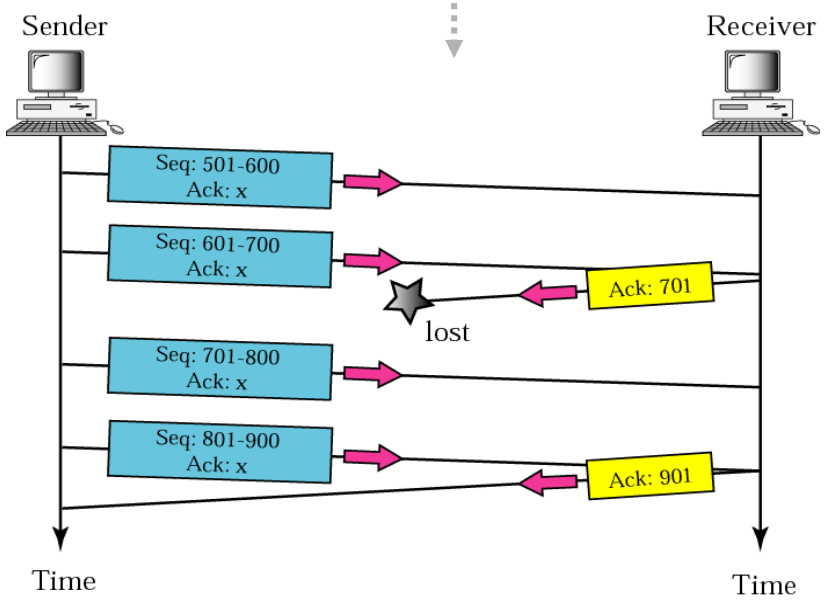
1) **Receiver** Detects lost ACKs by receiving duplicate segments
   - **action**: resend last ACK, discard duplicate segment

2) **Sender** Detects lost acknowledgments

(a) if corresponding timer expires, as in lost segments
(sender cannot tell the difference between a lost segment and a lost ACK – the same recovery mechanism applies)

(b) may not even notice a lost acknowledgment - TCP uses **cumulative acknowledgment system**, so each ACK is confirmation that everything up to the byte specified ☺



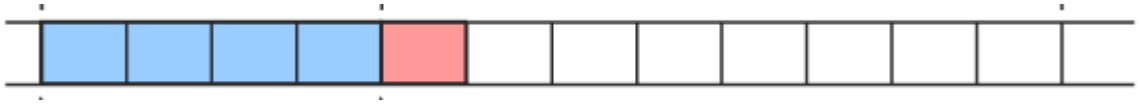


TCP Error Control (cont.)

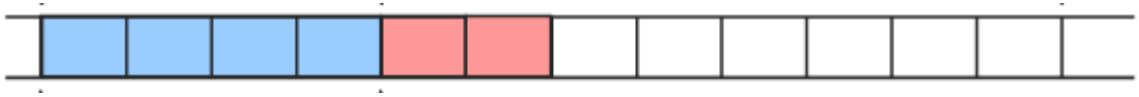
TCP ACK Generation Recommendation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed.	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK. (prevents creating extra traffic)
Arrival of in-order segment with expected seq #. One other segment has ACK pending.	Immediately send single cumulative ACK, ACKing both in-order segments. (do not let sender wait too long)
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected.	Immediately send duplicate ACK, indicating seq. # of next expected byte. (enables fast retransmit)
Arrival of segment that partially or completely fills gap.	Immediately send ACK, provided that segment starts at lower end of gap.

Many of TCP error correction procedures have evolved as a result of decades of experience!

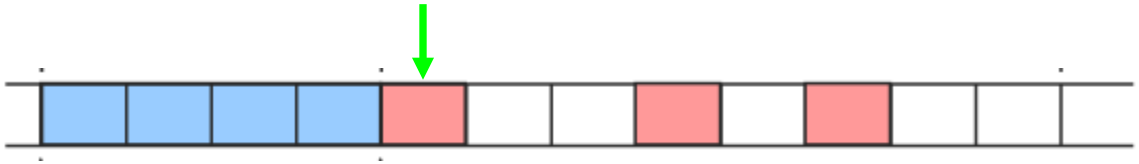


Delayed ACK. Wait up to 500ms.

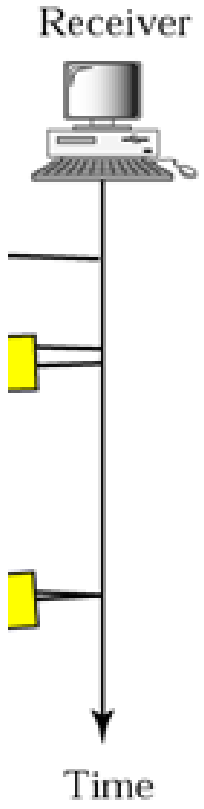


Immediately send single cumulative ACK.

Immediately send duplicate ACK, indicating seq. # of next expected byte.

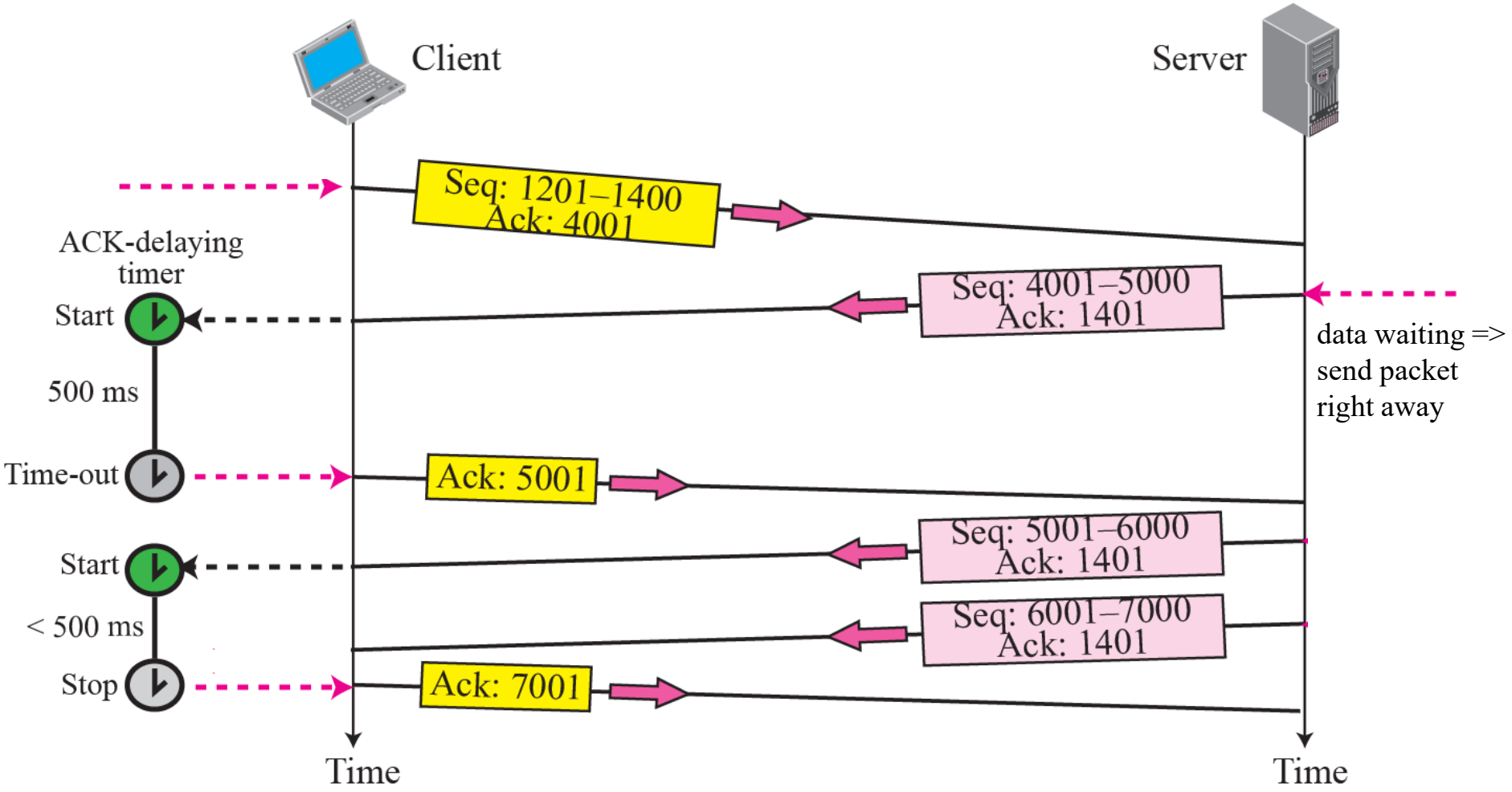


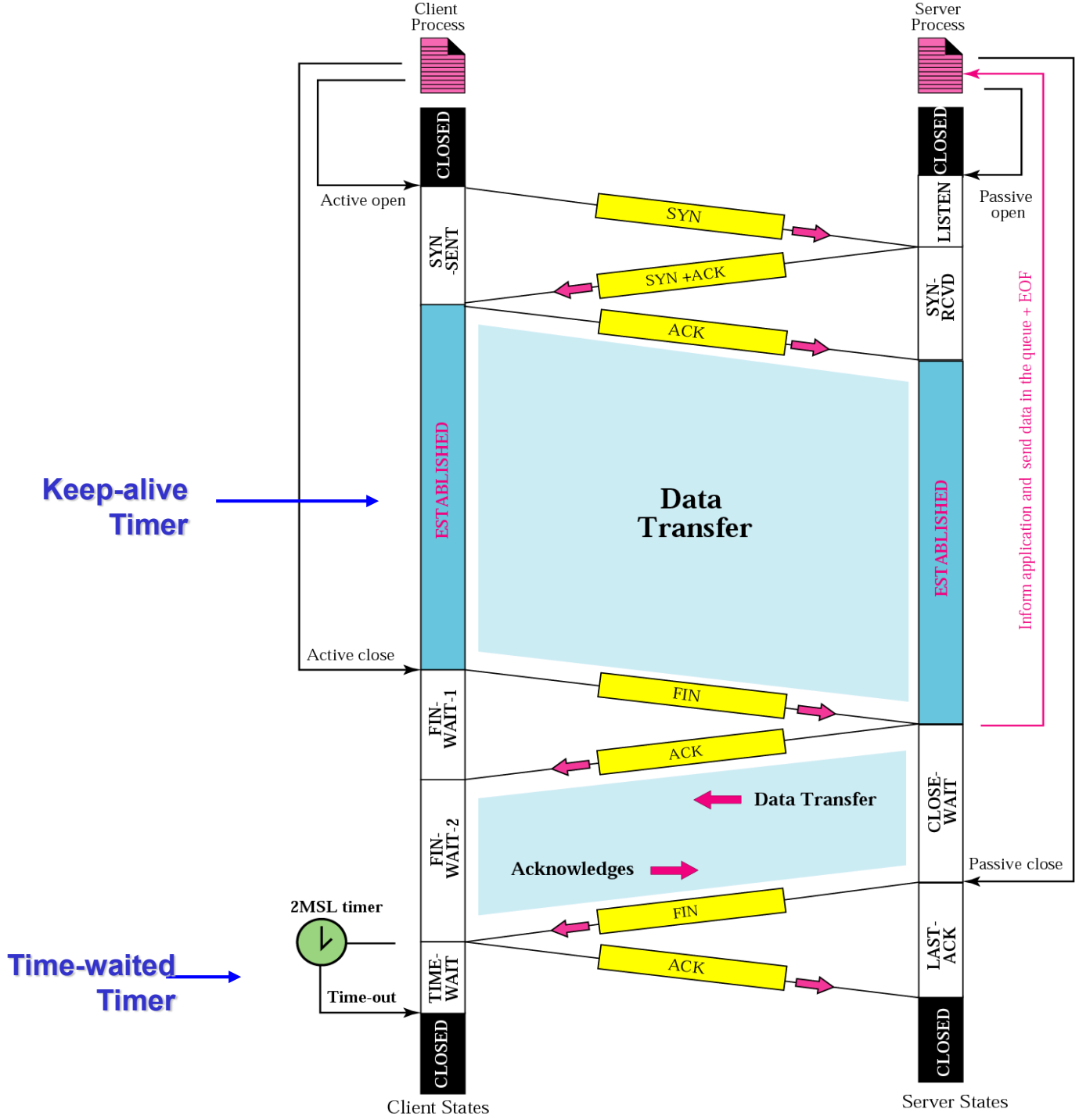
Immediately send new ACK, provided that segment starts at lower end of gap.



TCP Error Control (cont.)

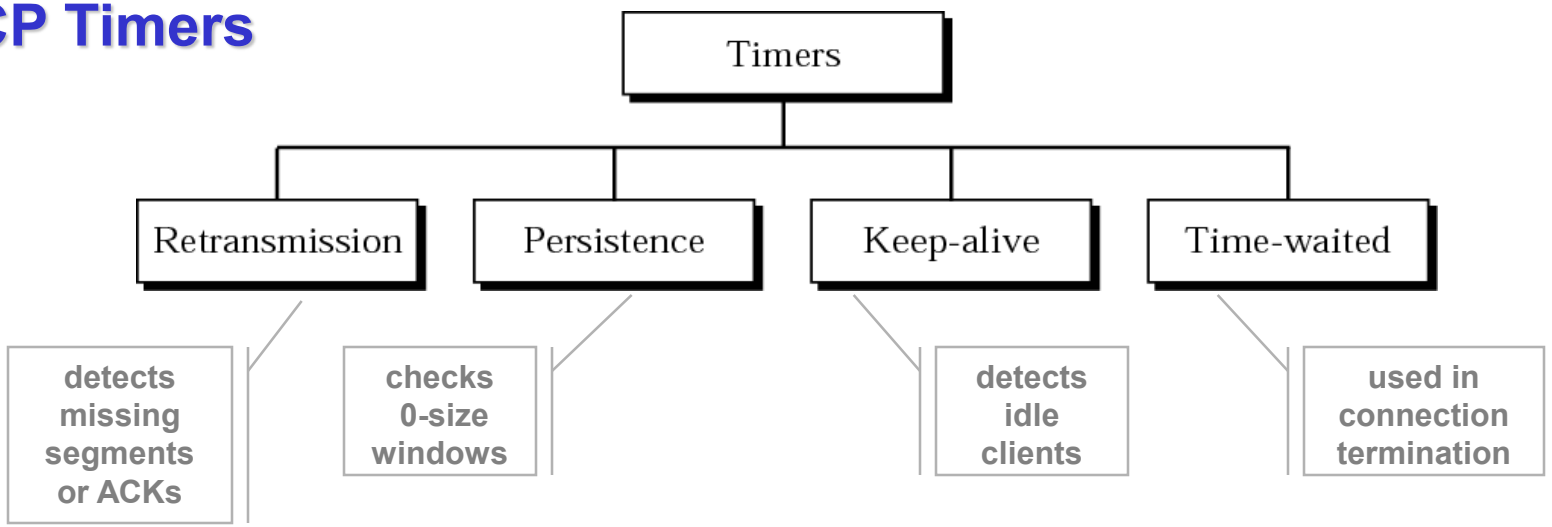
Example **[TCP error control – normal operation]**




Client Process
Server Process
CLOSED
CLOSED
Active open
Passive open
SYN
LISTEN
SYN -SENT
SYN +ACK
SYN-RCVD
ACK
Keep-alive Timer
ESTABLISHED
Data Transfer
ESTABLISHED
Inform application and send data in the queue + EOF
Active close
FIN
ACK
FIN-WAIT-1
Data Transfer
CLOSE-WAIT
FIN-WAIT-2
Acknowledges
Passive close
2MSL timer
FIN
Time-waited Timer
Time-out
ACK
TIME-WAIT
LAST-ACK
CLOSED
CLOSED
Client States
Server States

TCP Error Control (cont.)

TCP Timers



Time-waited Timer – during connection termination, connection is held limbo for time-waited period (usually 2*MSL) to make sure that all packets created by this connection have died off (see pp. 29 earlier lecture)

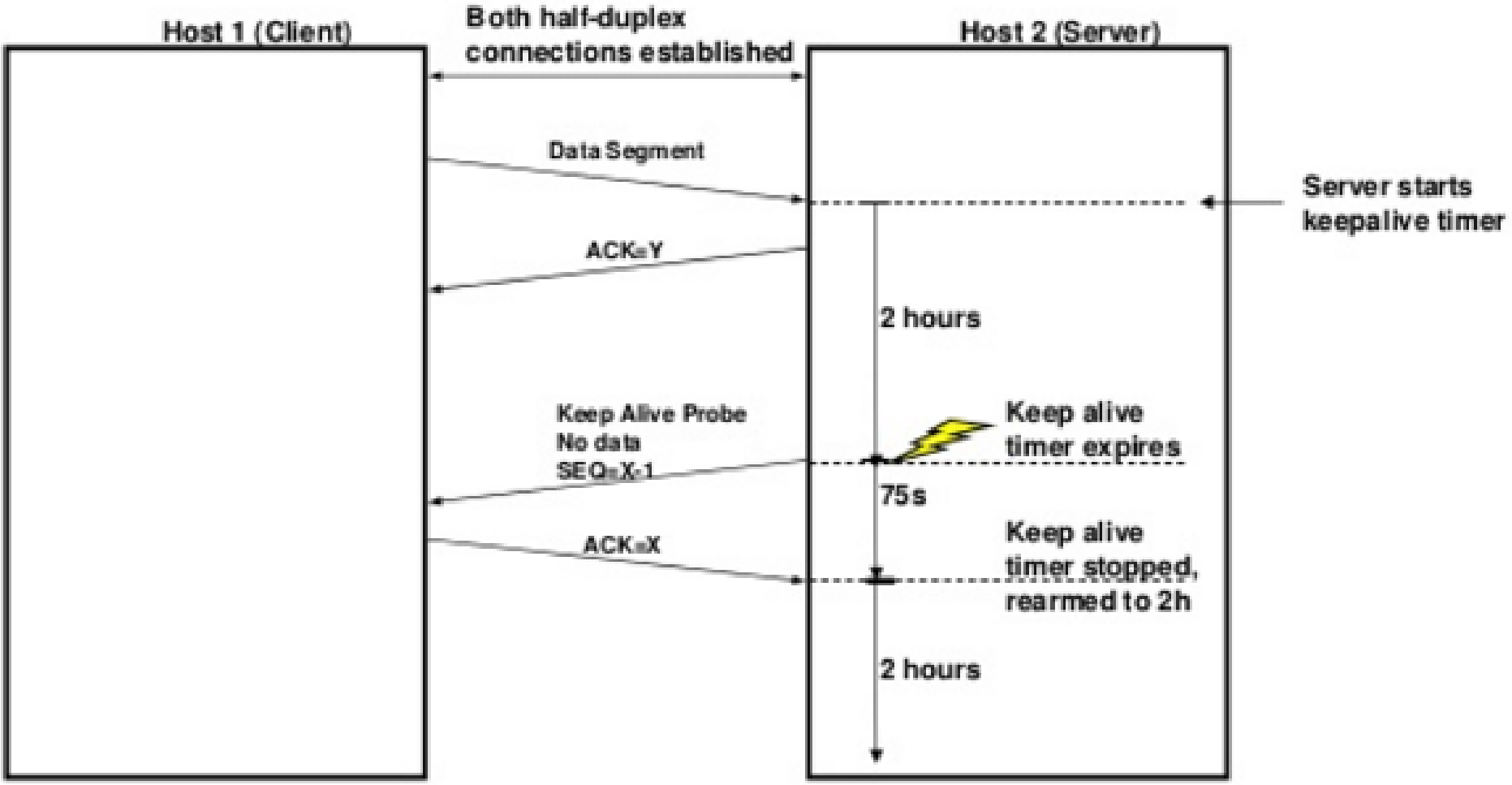
MSL ≈ 120 [sec] – may depend on OS implementation

Keep-alive Timer – used in some implementations to prevent a long idle connection between two processes (usually 2h)

- each time server hears from client, it resets this timer
- if server does not hear from client in 2h, it sends a probe segment; if there is no response in 10 probes, 75 sec apart, server terminates connection

TCP Error Control (cont.)

Example **[Keep-alive Timer]**



TCP Error Control (cont.)

Example **[Keep-alive Timer]**