Internet Technology

06. TCP: Transmission Control Protocol

Paul Krzyzanowski

Rutgers University

Spring 2016

Last time: Reliable Data Transfer

- Checksum: so we can determine if the data is damaged
- ARQ (Automatic Repeat reQuest) protocols
  - Use acknowledgements to request retransmission
- Acknowledgement (receiver feedback)
  - **Retransmit** if NAK or corrupt ACK
- Sequence numbers
  - Allow us identify duplicate segments
  - No need for NAK if we use sequence numbers for ACKs
- Timeouts
  - Detect segment loss
  - time expiration = assume that a segment was lost

Last time: Reliable Data Transfer

- Stop-and-wait protocol
  - Do not transmit a segment until receipt of the previous one has been acknowledged
  - Leads to *extremely poor* network utilization
- Use a pipelining protocol
  - **Go-back-N** (**GBN**)
    - Window size *W* – no more than *W* unacknowledged segments can be sent
    - Cumulative acknowledgement
      - Receipt of a sequence number *n* means that all segments up to and including *n* have been received
    - Timeout: retransmit <u>all</u> unacknowledged segments
  - **Selective Repeat** (**SR**)
    - Acknowledge individual segments
    - Sender's window: *N* segments starting from the earliest unacknowledged segment
    - Per-segment timer on sender: retransmit only that segment on timeout
    - Receiver's window: buffer for *N* segments starting from the first missing segment
      - Receiver must buffer acknowledged out-of-order segments
      - Deliver segments to application in order

TCP: Transmission Control Protocol

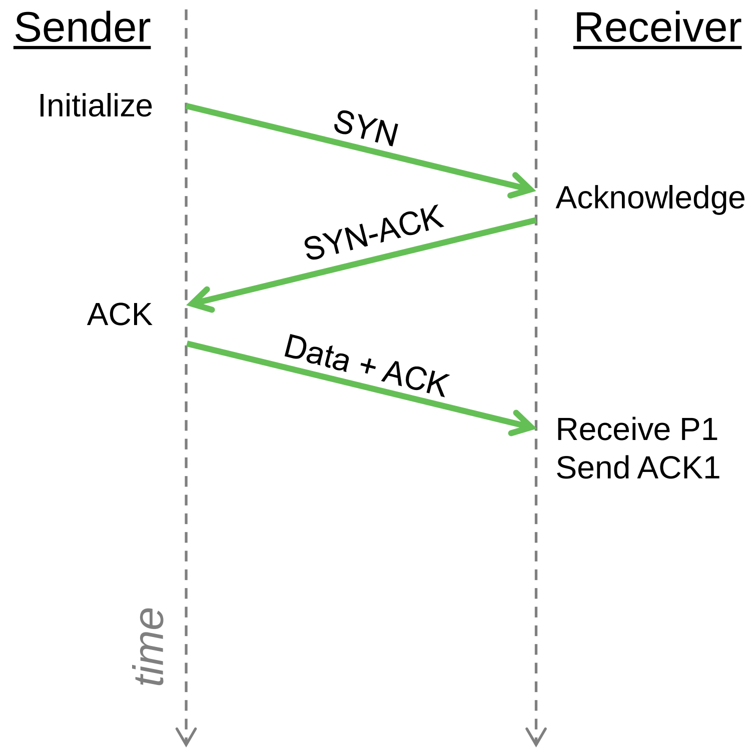
TCP

- Transport-layer protocol … like UDP
- But:
  - Connection-oriented
  - Bidirectional communication channel
  - Reliable data transfer
  - Flow control
- Network stacks on both end systems keep state
  - “Connection” managed only in end systems
  - Routers are not aware of TCP

TCP: Connection Setup

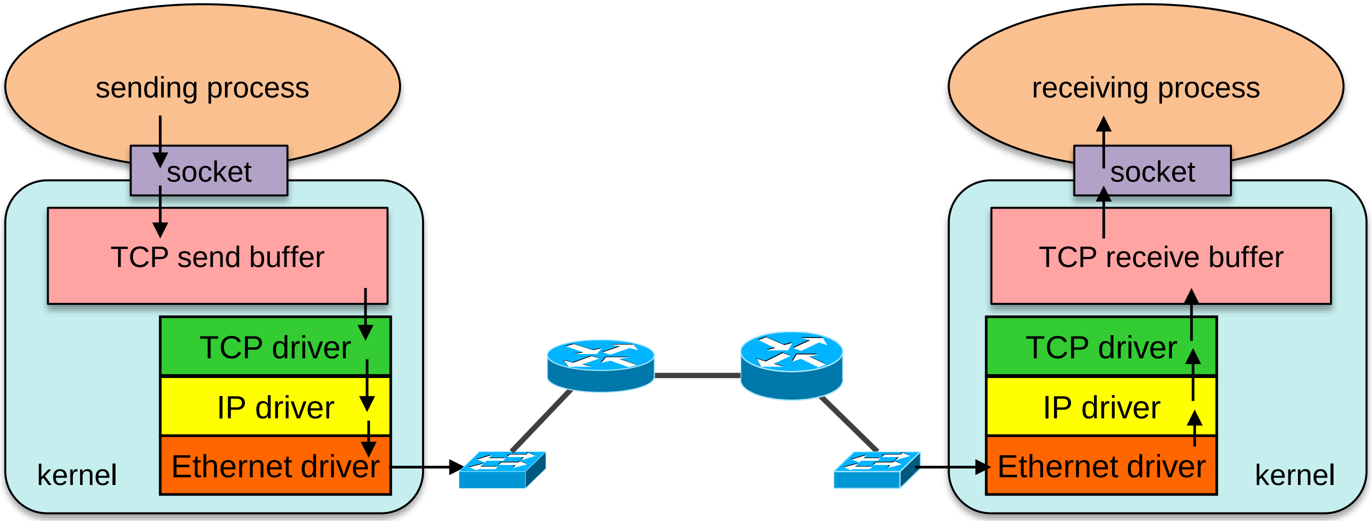
- Connection setup
  - Three way handshake
  - Negotiate parameters
  - Initialize state variables

(more details later!)

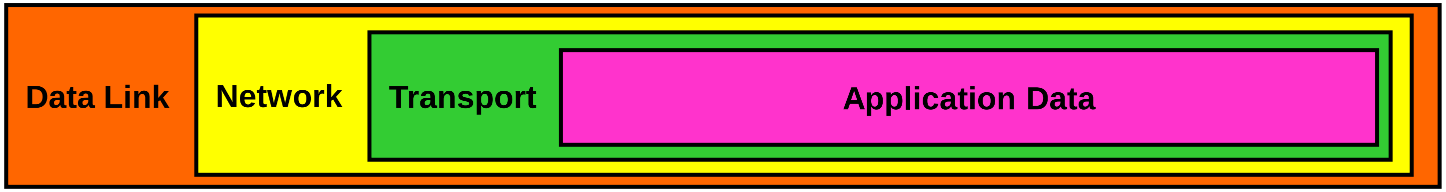


TCP Data Exchange

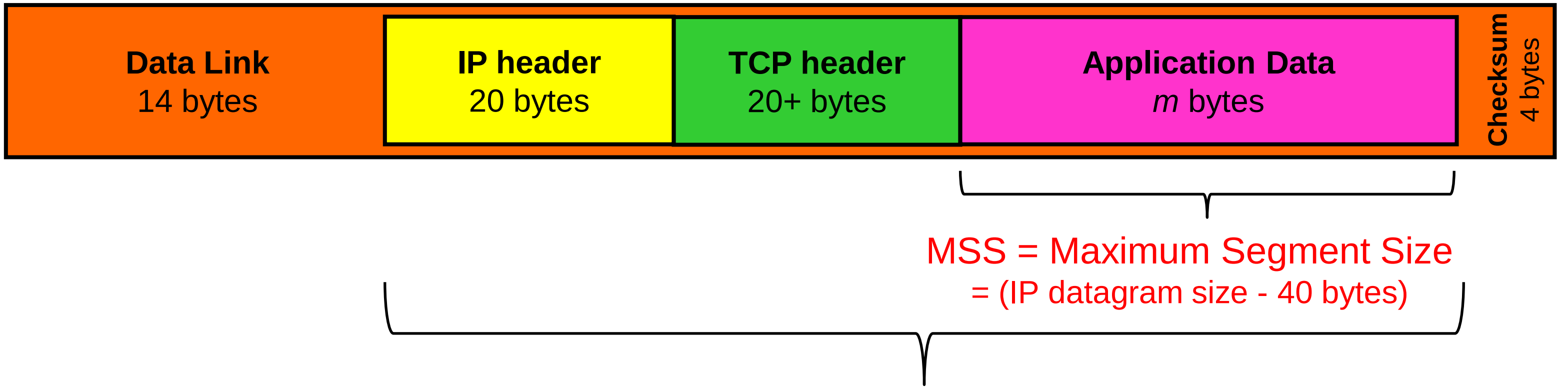
- TCP provides full duplex service
  - If a TCP connection has been established between processes *A* and *B*, *A* can send messages to *B* and *B* can send messages to *A* over the same connection
- Outgoing data is placed in TCP's send buffer
  - TCP takes data from here, creates segments, and sends them out
  - Data grabbed must be ≤ maximum allowable segment size (MSS)



TCP Segment Size



Protocol encapsulation: logical view



MTU = Maximum Transmission Unit
1500 bytes for Ethernet v2 (→MSS = 1460 bytes)
9000 bytes for Jumbo frames in gigabit Ethernet (→MSS = 8960 bytes)

Maximum Segment Size (MSS) is dependent on MTU (=MTU-40)

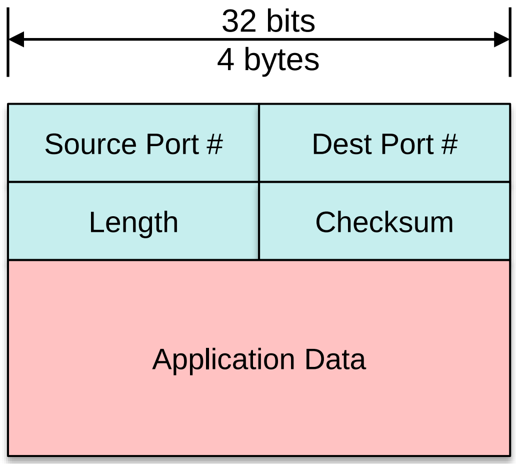
Path MTU Discovery

- What do we use for MTU?
  - No greater than the link layer's MTU (typically 1500 or 9000 bytes)
- Path MTU = Smallest MTU of any of the hops along the path to the destination
  - No easy (foolproof) way of determining this
- Path MTU Discovery (RFC 1191, 1981)
  - Send ICMP (Internet Control Message Protocol) packets (TCP in later versions)
  - Use MTU of 1st hop and set DF "don't fragment" bit on the IP packet
  - If the MTU of any hop is smaller, the router will
    - Discard the packet
    - Return "ICMP Destination Unreachable" message with a code indicating "fragmentation needed"
    - Place the MTU of the next hop in a 16-bit field in the ICMP header
  - The source will reduce its MTU and try again until it gets to the destination
  - Repeat the discovery process periodically: default = 10 minutes on Windows & Linux
- Routers must handle an MTU of at least 576 bytes (512 bytes + headers)
  - Minimum MTU for IPv6 = 1280 bytes

Try `tracepath` on Linux or `mturoute` on Windows

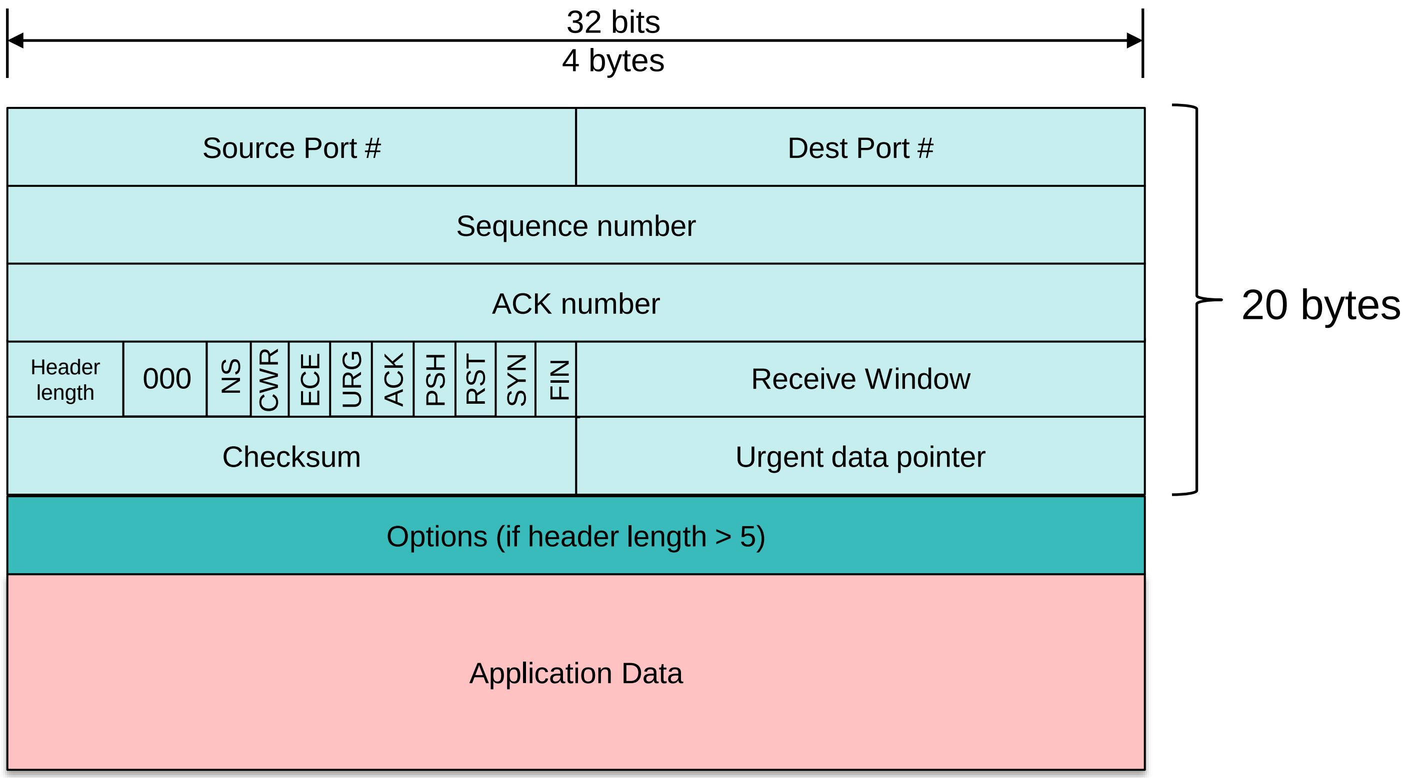
UDP Segment Structure

- Defined in RFC 768
- Eight byte header



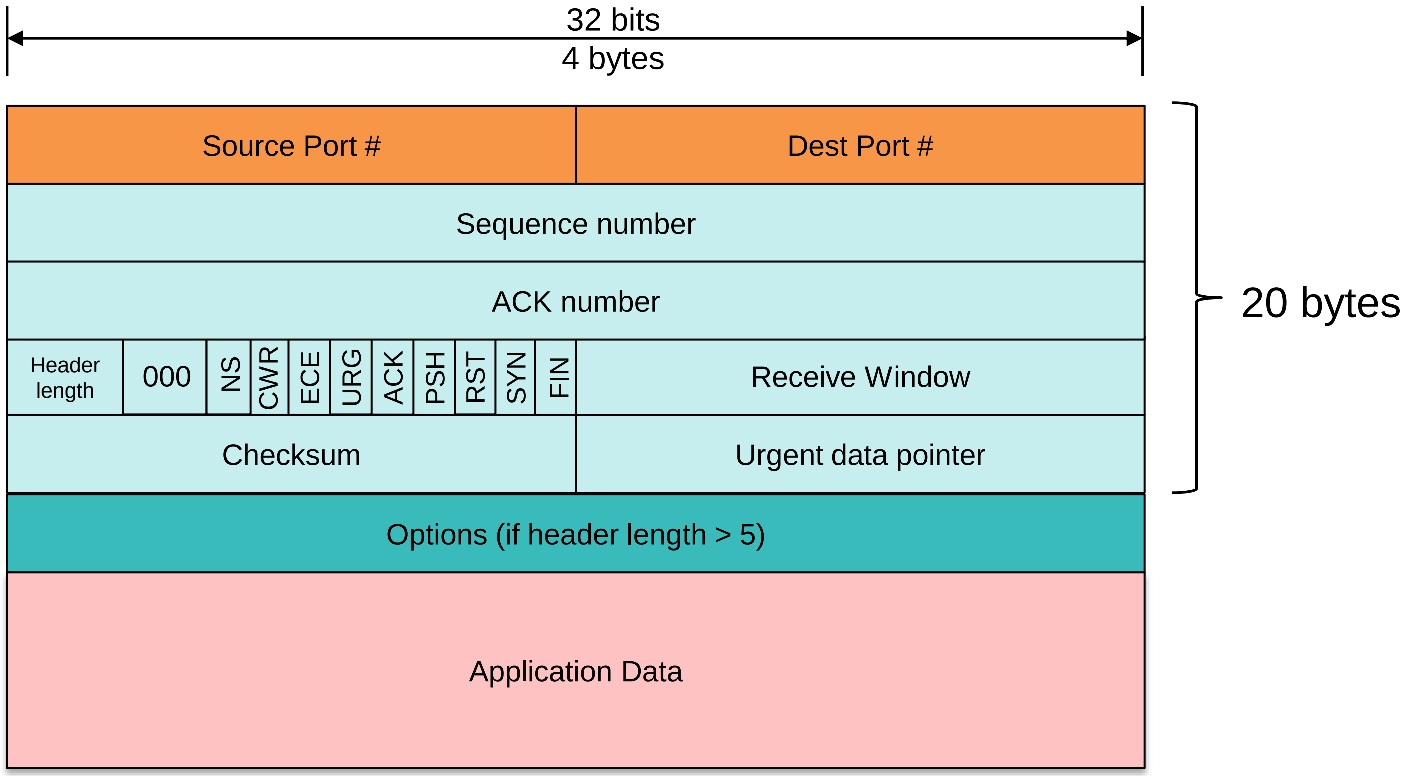
TCP Segment Structure

- Defined in RFC 1122 (and others)
- 20-byte header



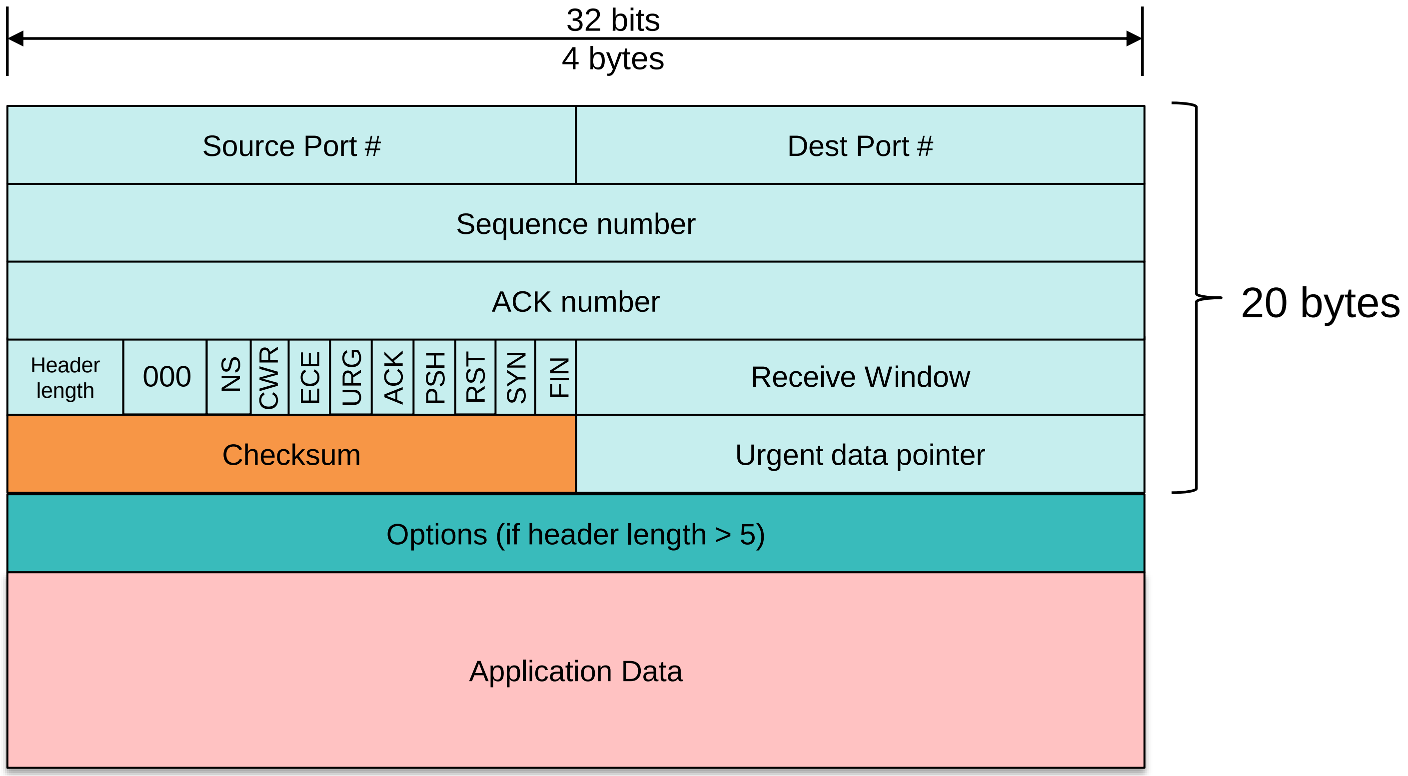
TCP Segment Structure: port numbers

- Source & Destination port numbers
  - Used for multiplexing & demultiplexing



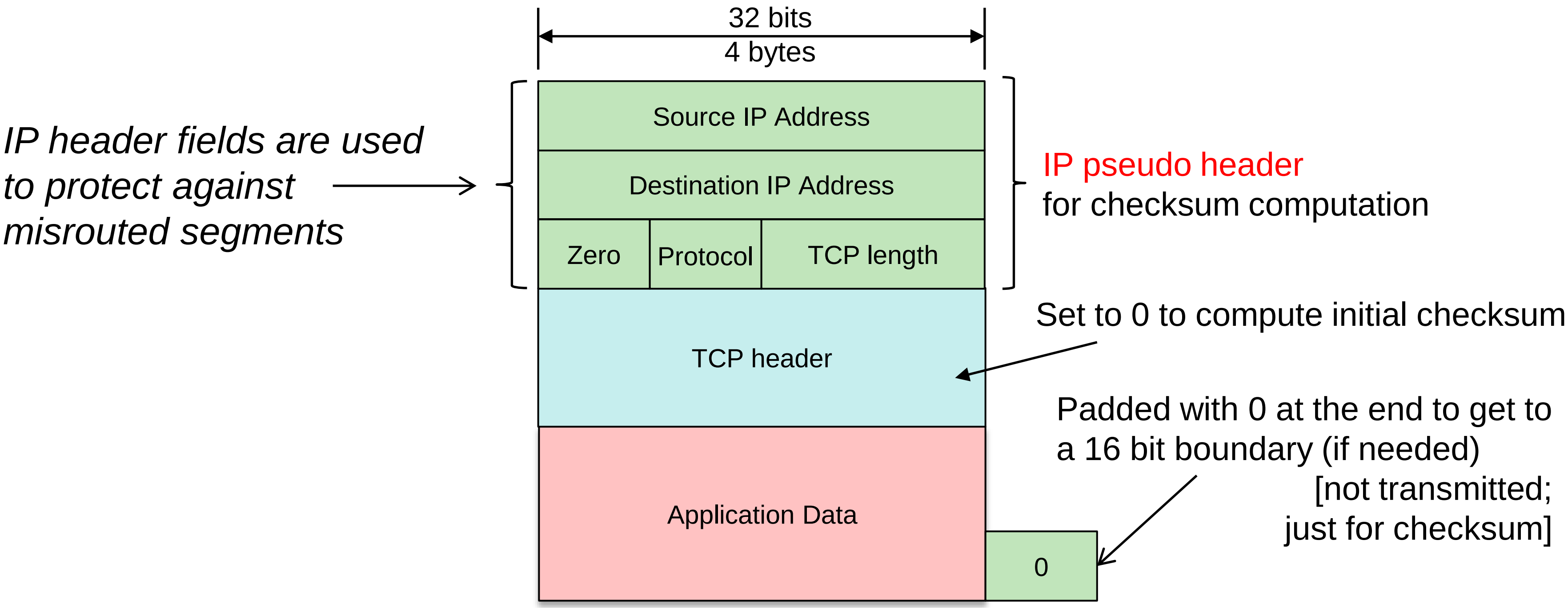
TCP Segment Structure: checksum

- 16-bit checksum checks for data corruption in transmission



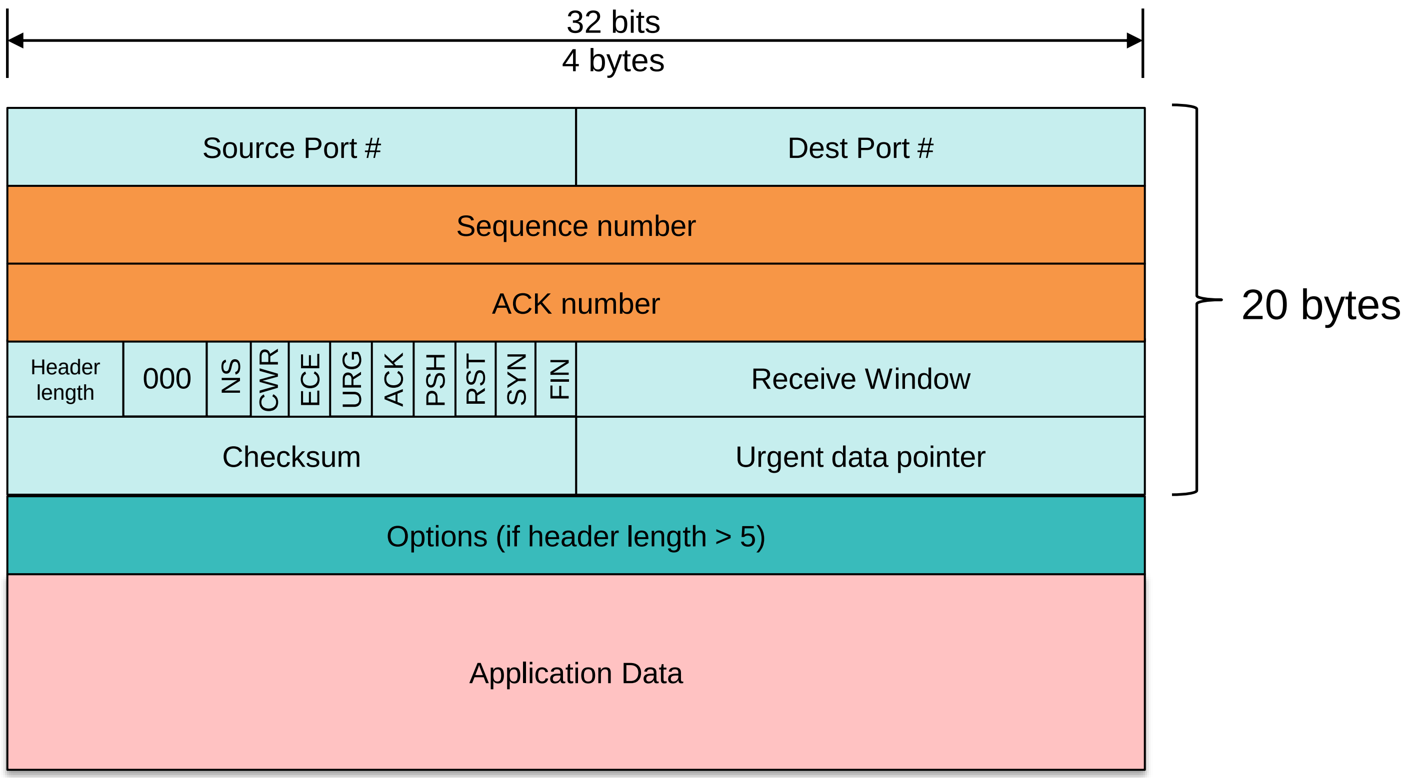
TCP Checksum

- As with UDP, the TCP header contains a 16-bit checksum
  - Checks for data corruption ⇒ *same computation as for IP and UDP checksums*
- Checksum is generated by the sender and validated only by the receiver
- Checksum is a 16-bit one's complement sum of:
  IP pseudo header, TCP header, and data



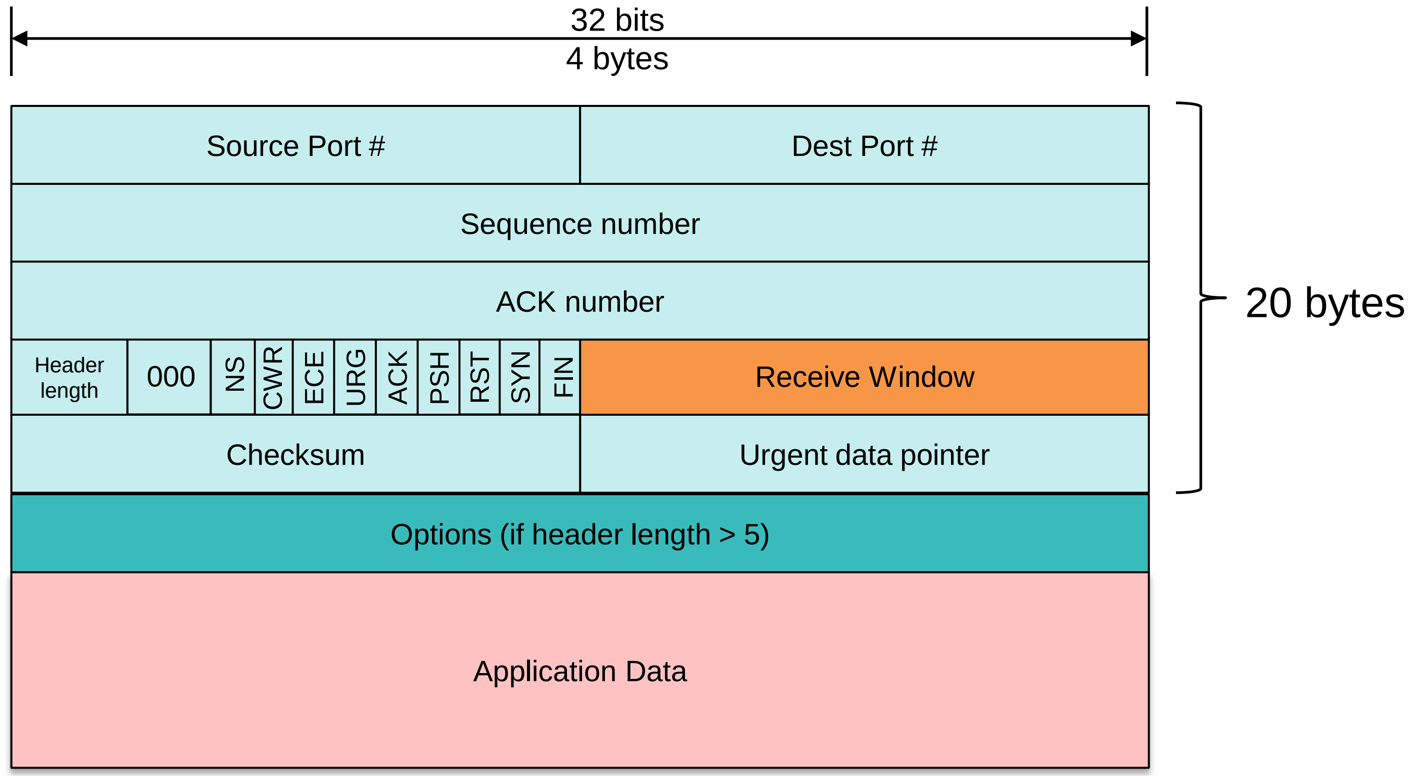
TCP Segment Structure: sequence numbers

- 32 bit sequence # and acknowledgement #
  - used for creating a reliable data transfer service



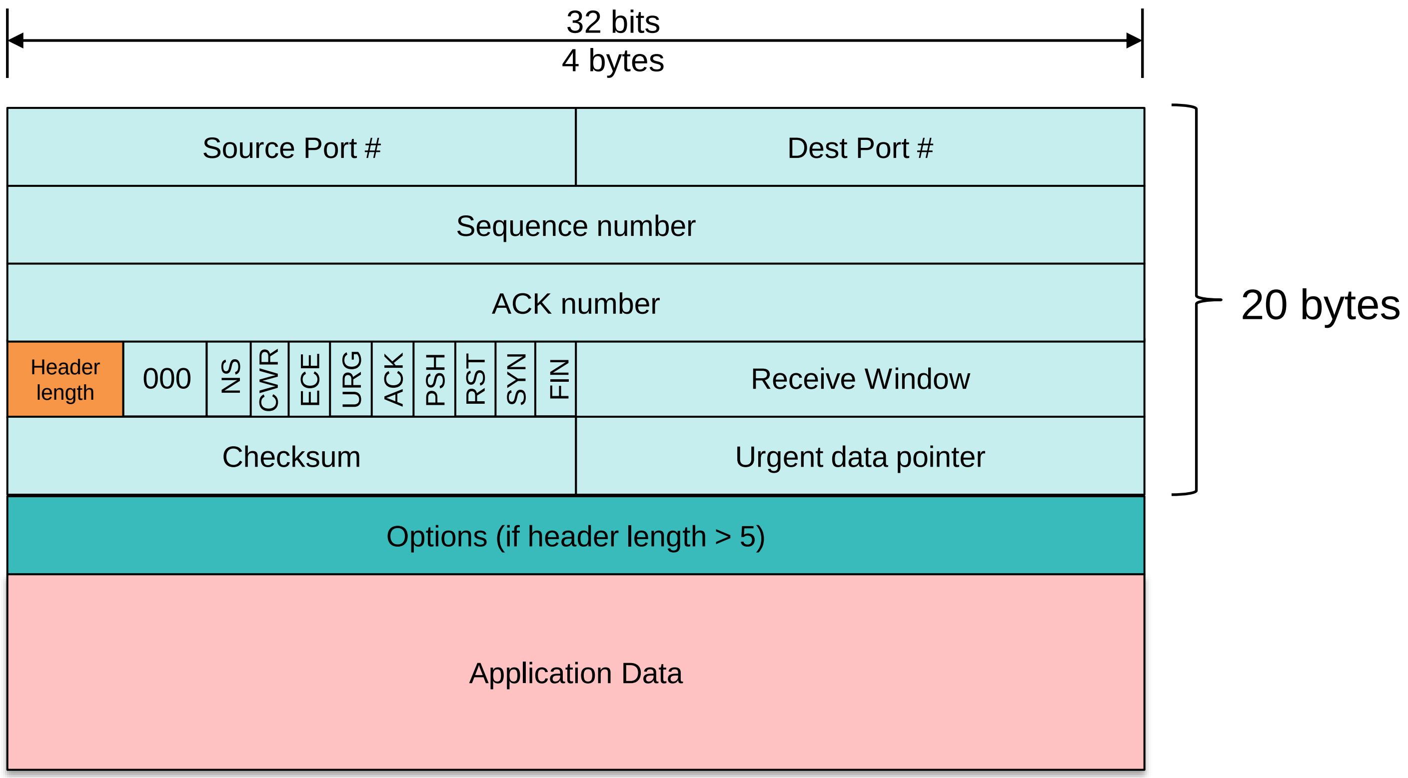
TCP Segment Structure: receive window

- number of bytes the receiver is willing to accept
  - used for flow control



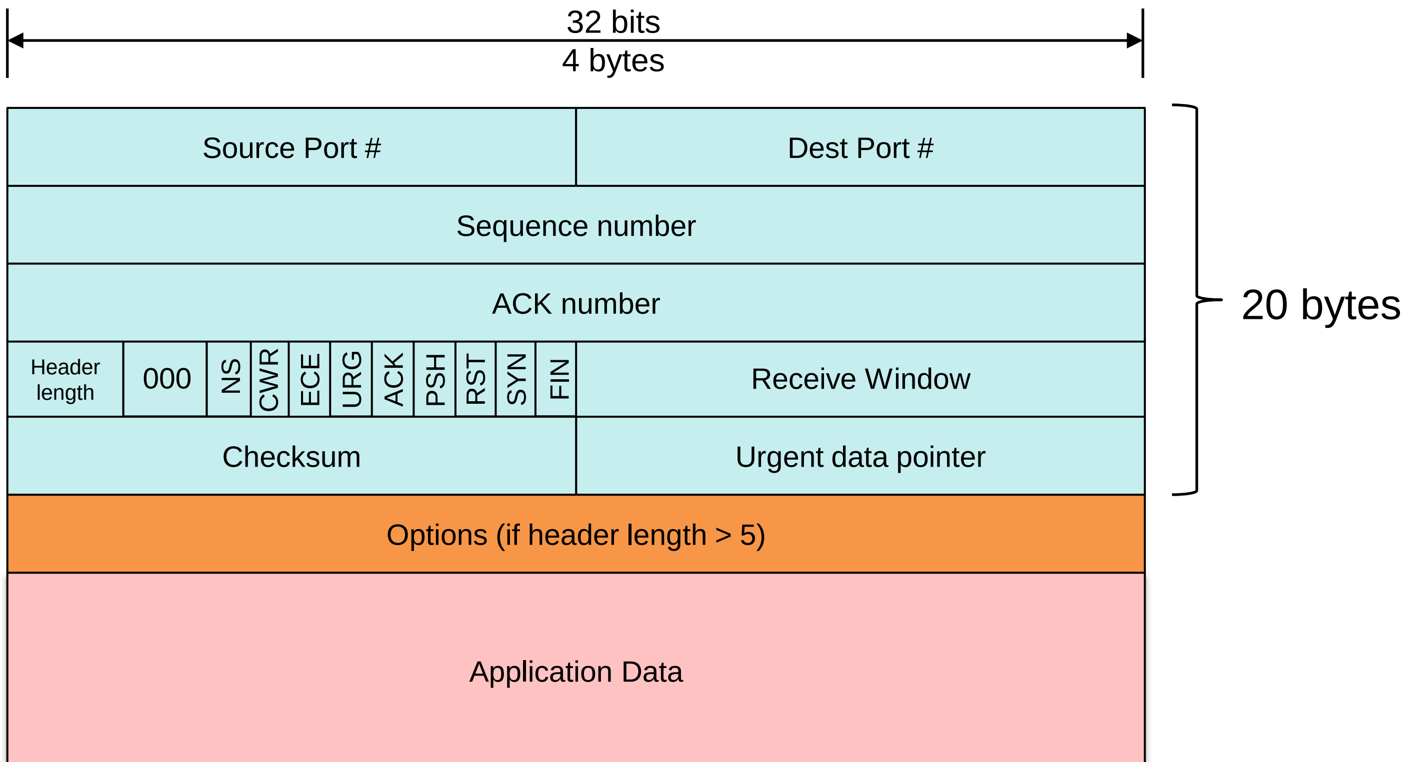
TCP Segment Structure: header length

- 4-bit header length: length of TCP header in 32-bit words
  - This is almost always 5 (20 bytes)



TCP Segment Structure: options

- Variable size *options* field
  - empty in most segments
  - maximum segment size negotiation, window scaling factor, timestamps, alternate checksum, selective acknowledgements

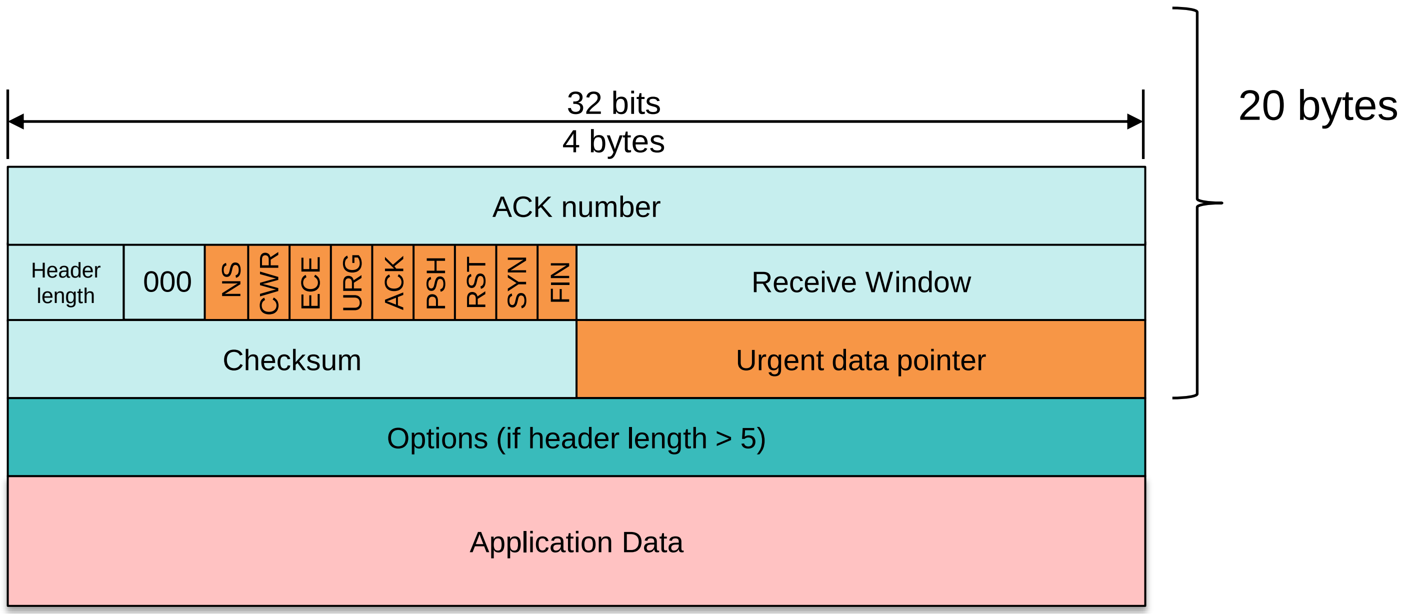


TCP Segment Structure: flags

- ACK: acknowledgement number contains valid data
- RST, SYN, FIN: used for connection setup/teardown
- PSH (push): pass data to upper layer immediately
- URG: application data contains a region of “urgent” data
  - 16-bit urgent data pointer points to last byte of this data

Push and Urgent are not used in practice

- NS, CWR, ECE: used for congestion notification



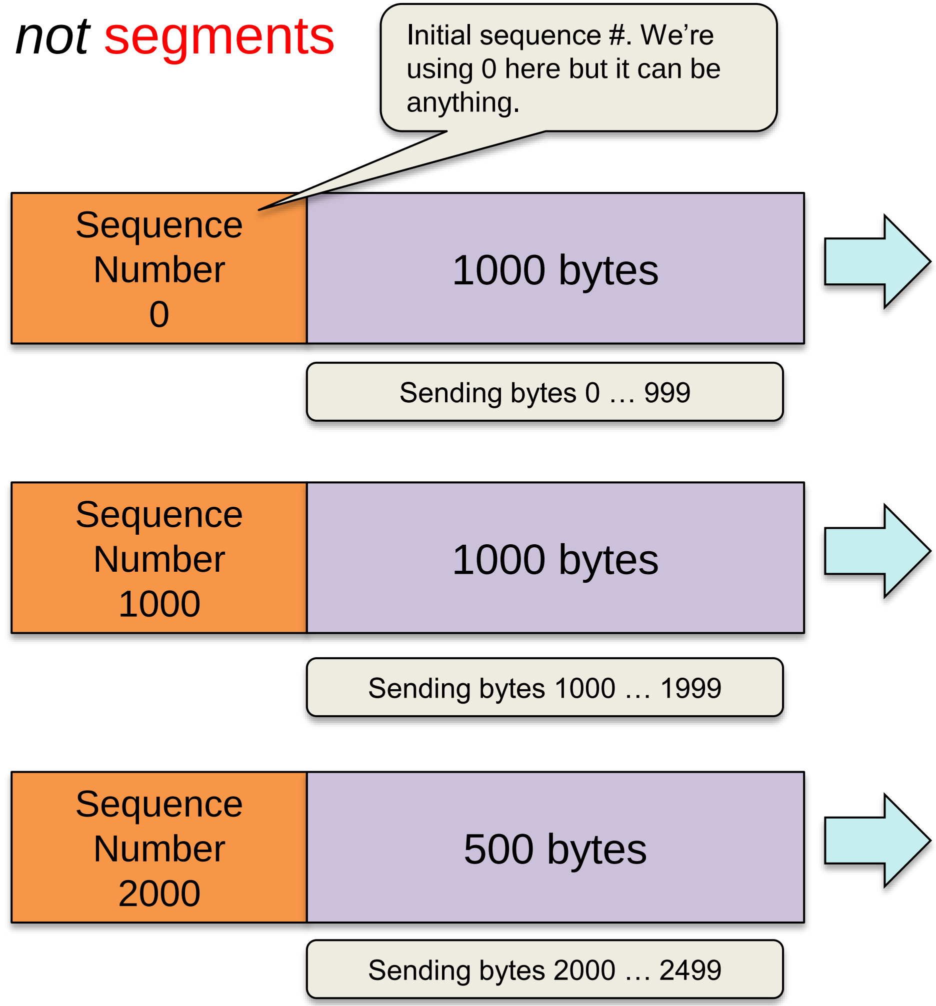
TCP sequence numbers

- TCP views application data as an ordered stream of bytes
- Sequence numbers count bytes, *not* segments

Suppose initial sequence # = 0
and we send a segment with 1000 bytes

Send next segment with 1000 bytes

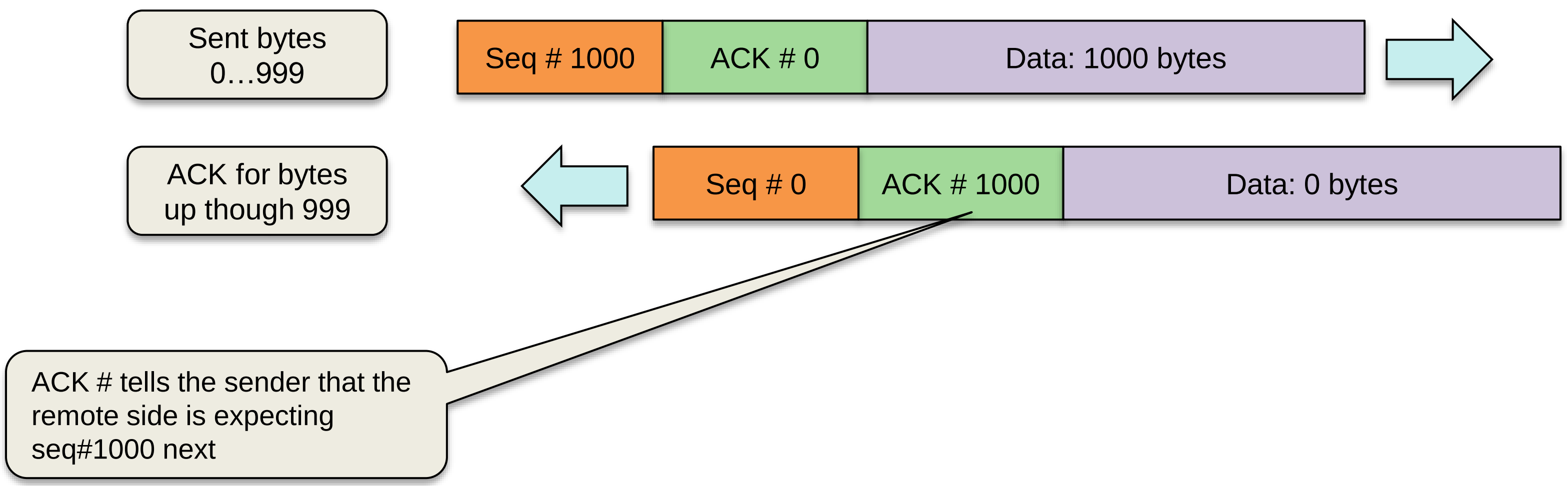
Send next segment with 500 bytes



TCP acknowledgement numbers

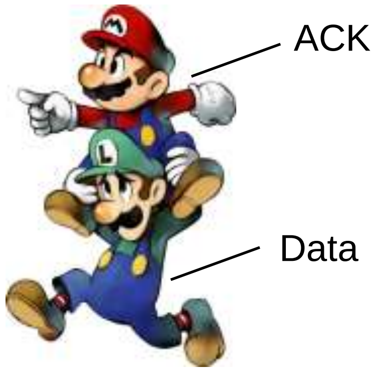
Acknowledgement number

– Number of the next byte the host is expecting from the other side (starting from the initial sequence number at the start of the connection)



Piggybacking acknowledgements

- If a host has TCP data to transmit on a connection
  - Acknowledgement placed in that TCP header (piggyback)
  - No need to send a separate acknowledgement message
- If there is no data to transmit
  - Acknowledgement sent with no data



Sent bytes 1000…1999

Seq # 1000	ACK # 0	Data: 1000 bytes

→

Received bytes 0…235 and ACK for bytes up to #1999

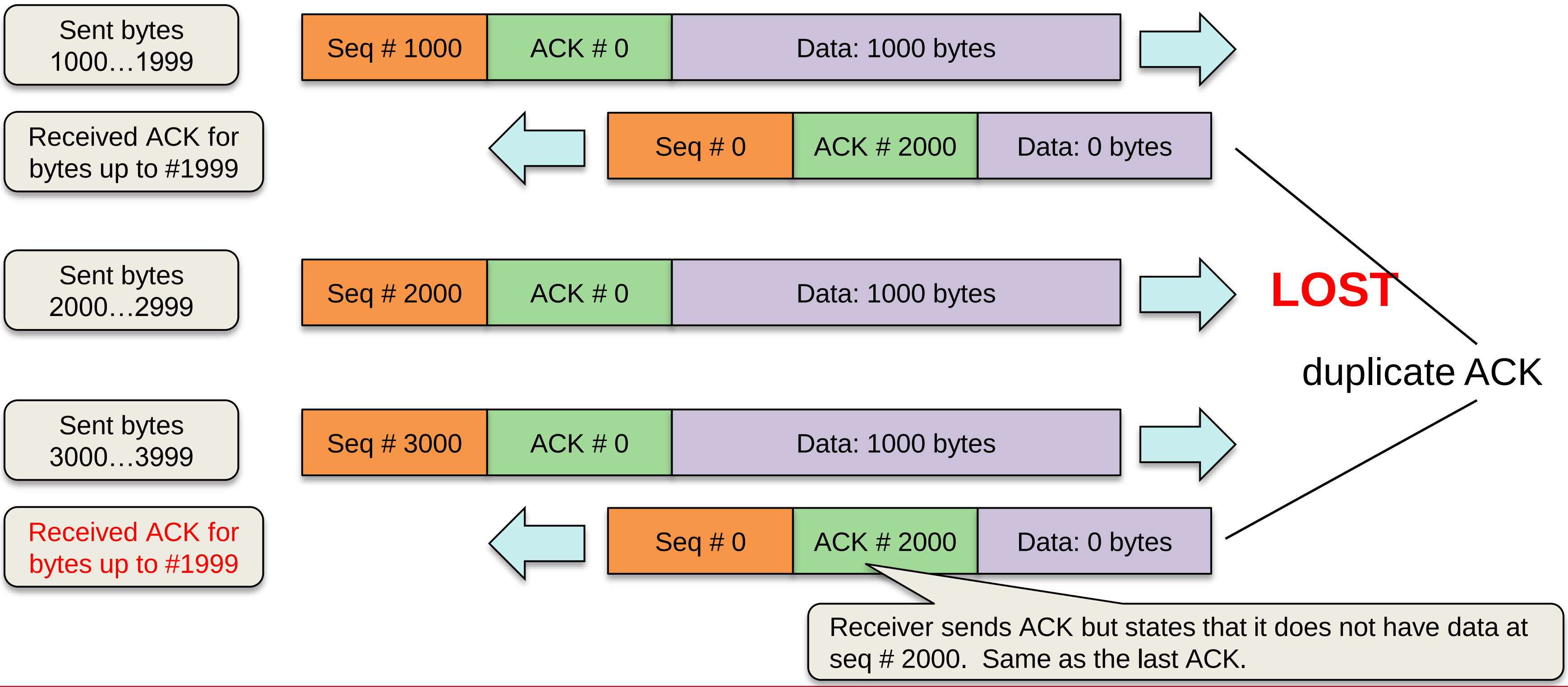
←

Seq # 0	ACK # 2000	Data: 236 bytes

This TCP segment contains 236 bytes of data *and* acknowledges that bytes up to #1999 have been received. It wants a segment starting from #2000 next.

Cumulative & Duplicate acknowledgements

- TCP uses **cumulative acknowledgements**
  - Every packet that is received without error is acknowledged
  - The ACK # is the byte number that the receiver wants to see next
- Let's assume that we sent 3 TCP segments but one gets lost: we get 2 ACKs
  - The second ACK is a **duplicate acknowledgement**



Out of order data

- A segment that arrives out of order is not acknowledged
  – Instead, a duplicate ACK is sent asking for the missing sequence

- TCP protocol does not define what happens to the received segment

- Two options:
  1. Discard it
  2. Hold on to out of order segments and wait for missing data
     - More complex
       … but much more efficient for the network
     - This is the preferred approach

TCP ACK generation

Event	Receiver action
Arrival of in-order segment. All data up to this sequence # has been acknowledged.	**Delayed ACK**. Wait up to 500 ms for the arrival of another in-order segment. Otherwise send ACK.
Arrival of in-order segment. One other in-order segment waiting for ACK transmission.	Send a single **cumulative ACK**. This acknowledges both segments.
Arrival of out-of-order segment with higher sequence #.	Send **duplicate ACK** with sequence number of next expected byte.
Arrival of out-of-order segment that fills in a gap	Send ACK with sequence number of next unfilled byte (might be duplicate).

TCP Timeouts

Round-trip time estimation

- **Round trip time**:
  - elapsed time from sending a segment to getting an ACK
- RTT helps us determine a suitable timeout value

- TCP measures RTT for each non-retransmitted segment
- RTTs fluctuate
  - SRTT = "**Smoothed Round Trip Time**" = weighted average

$$SRTT = (1 - \alpha) \cdot SRTT + \alpha \cdot RTT$$

$\alpha = 0.125$

  - Exponential weighted moving average (**EWMA**)
  - Greater weight on recent measurements

Round-trip time variation estimation

- Compute the average variation in round-trip time from the estimate (smoothed average)
- Another exponential weighted moving average

$$\mathrm{RTTVAR} = (1 - \beta) \cdot \mathrm{RTTVAR} + \beta \cdot (\mathrm{SRTT} - \mathrm{RTT})$$

Round Trip Time Variation

Smoothed Round Trip Time

$\beta = 0.25$

- RTTVAR = estimate of how much RTT typically deviates from SRTT

See RFC 6298

Setting the TCP timeout interval

- Timeout ≥ SRTT
  - Otherwise we'll time out too early and retransmit too often
  - But don't want a value that's too high
    - Because we will introduce excessive delays for retransmission
- Use SRTT + *x*
  - *x* should be large when there is a lot of variation in RTT
  - *x* should be small when there is little variation in RTT
  - This is what RTTVAR gives us!
- TCP sets retransmission timeout to:

  Timeout interval = SRTT + 4 · RTTVAR

  - Initial value of 1 second
- When timeout occurs, the timeout interval is doubled until the next round trip

TCP Reliable Data Transfer

TCP reliable data transfer

- TCP uses a single timer
  - Even if there are multiple transmitted unacknowledged segments
  - Less overhead than a timer per segment
- Timer is associated with **oldest unacknowledged segment**
- Receiver sends cumulative acknowledgements

Receiver tells us it correctly received all bytes up to *y-1*

If received data from application

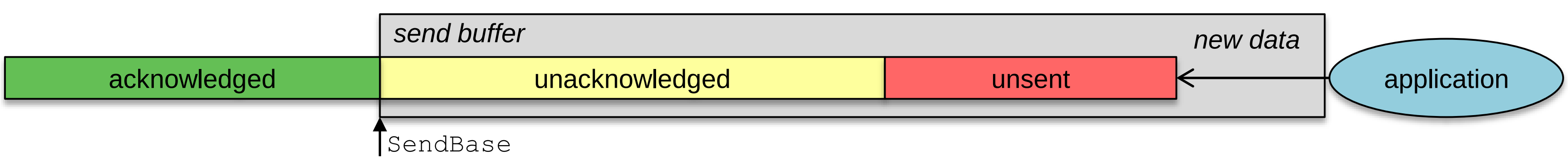
- Create TCP segment
- Set sequence #
- Start timer (=timeout interval) if not already running
- Send data to IP layer
- next sequence # = sequence # + data size

If timeout

- Retransmit non-acknowledged segment with smallest sequence #
- Start timer

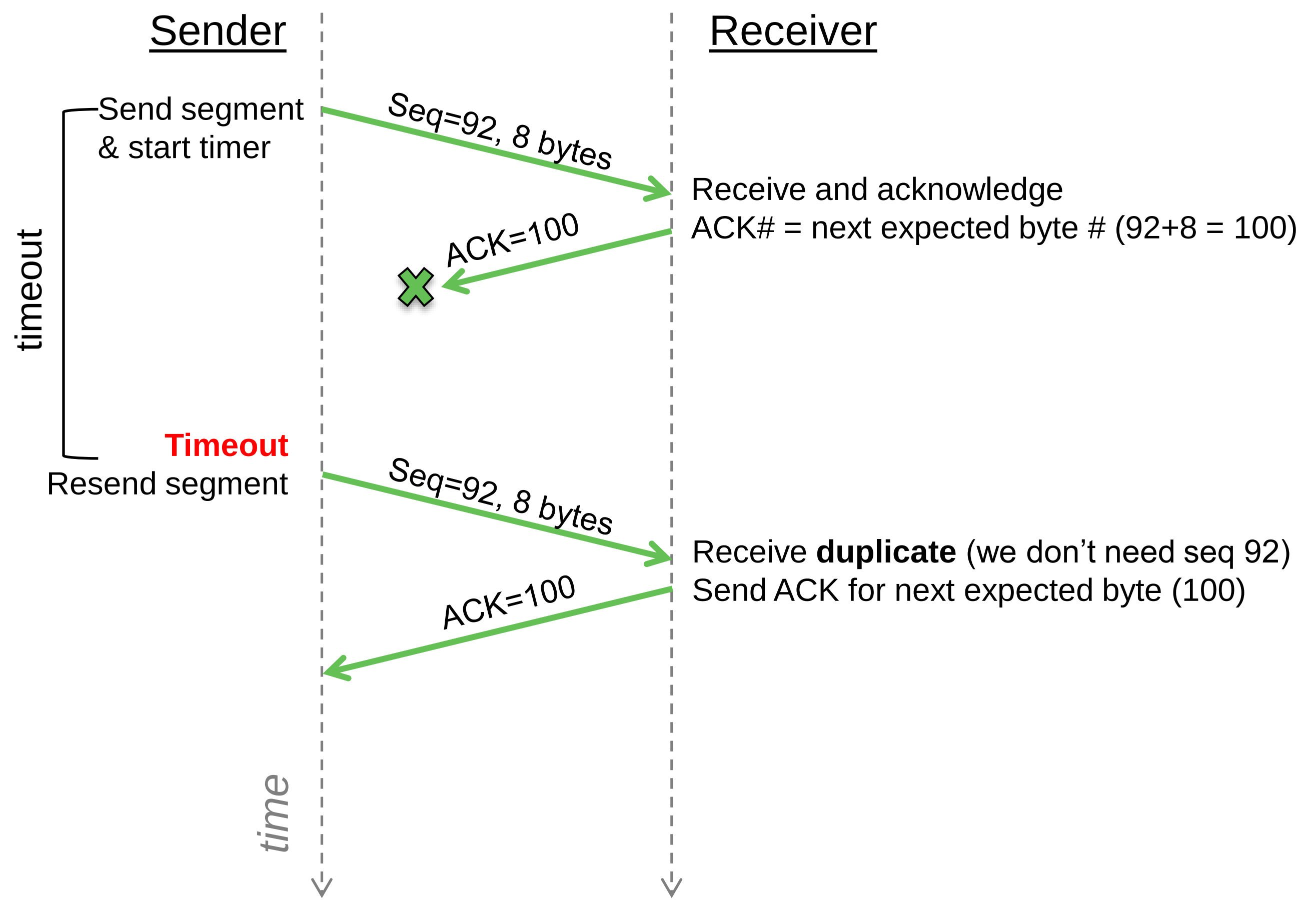
If receive ACK value *y*

- if (y > `SendBase`)
  `SendBase` = y
- if any non-acknowledged segments remaining, start timer



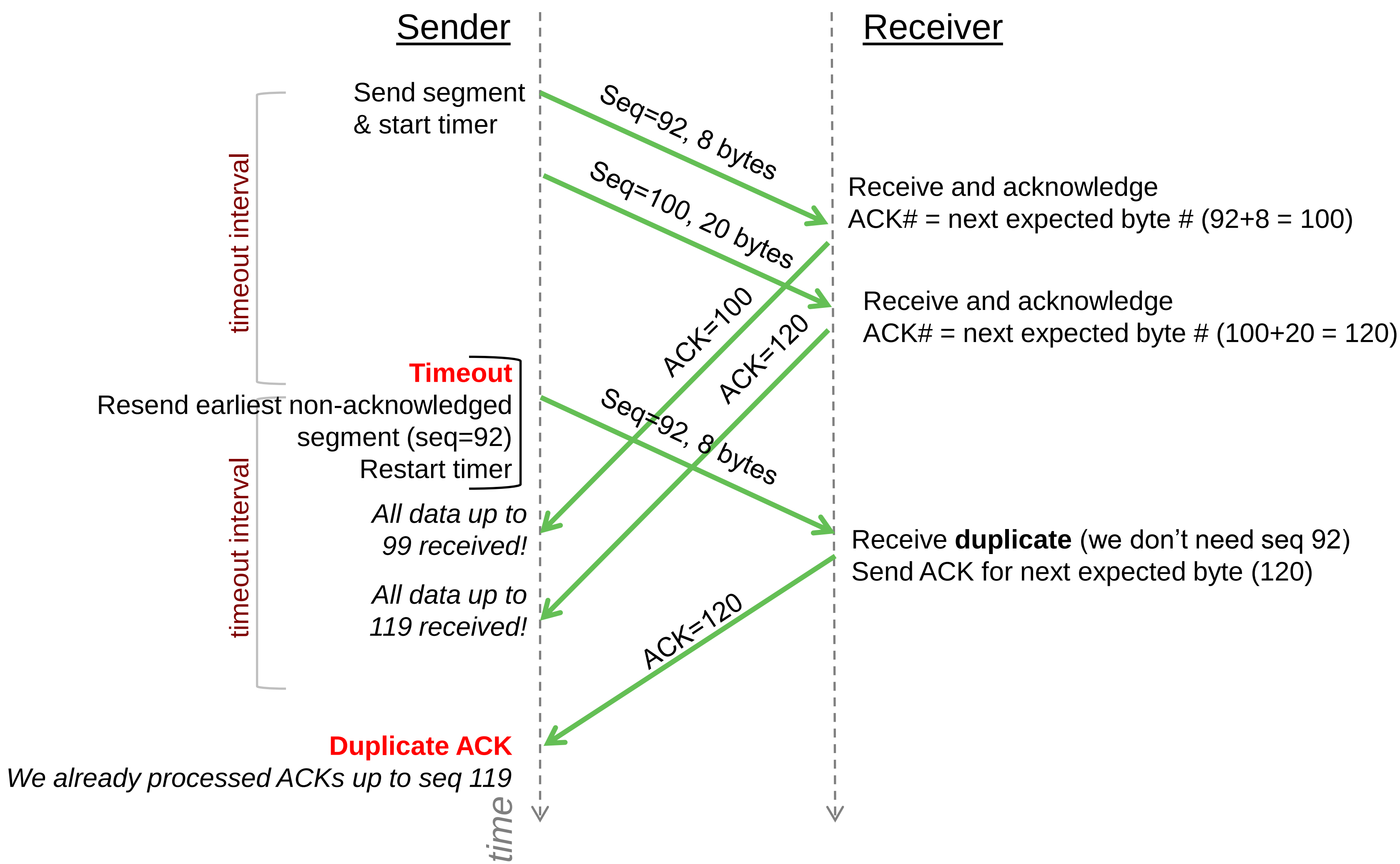
Example: Lost ACK

On timeout, sender retransmits segment with the same sequence #



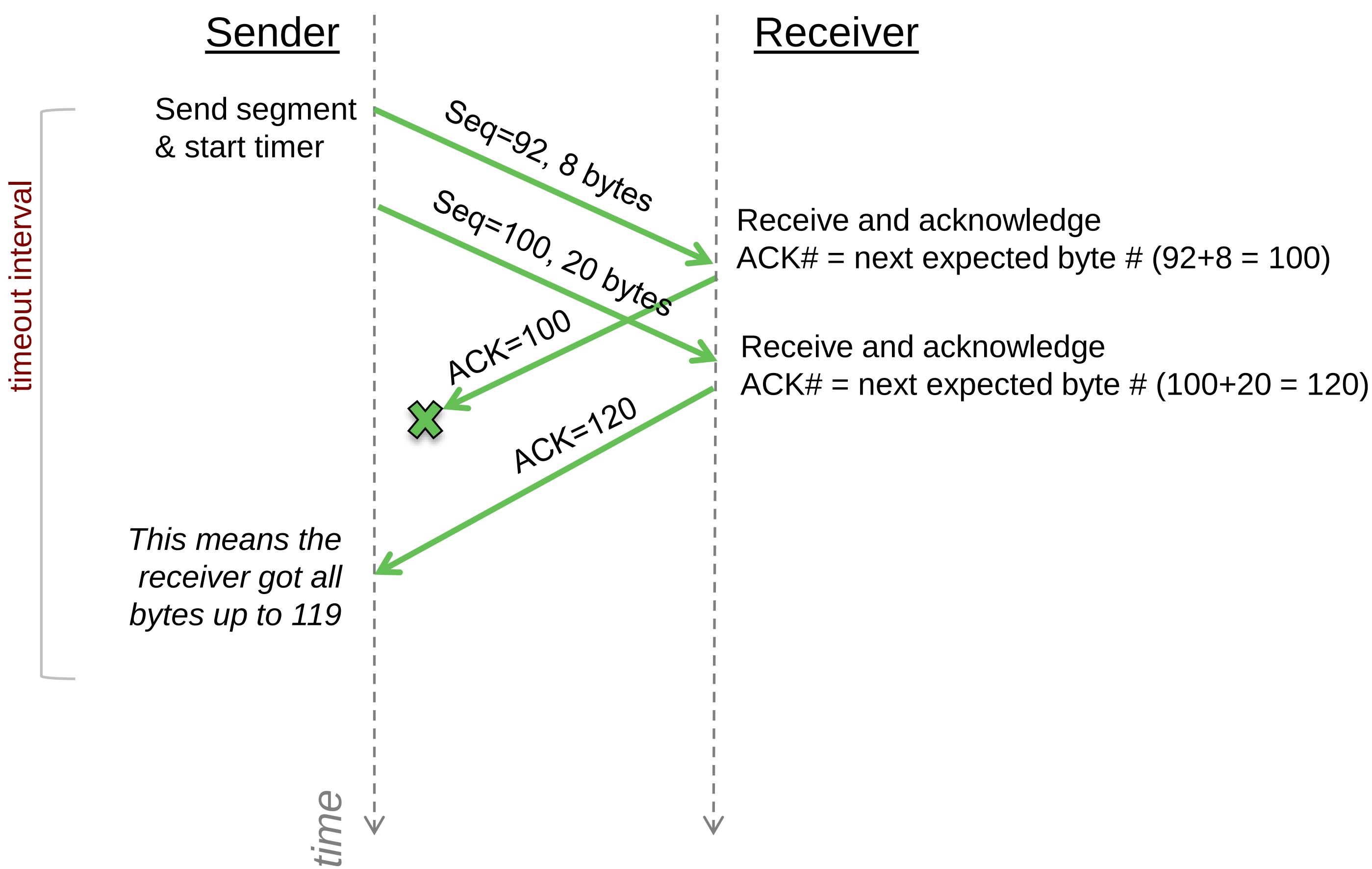
Example: Delayed ACKs

Pipelined transmits; delayed ACKs. What happens?



Example: Lost ACK for one segment

ACKs are cumulative; it's OK if we miss some



Timeouts

- Timeout interval is normally set to

  Timeout interval = SRTT + 4 · RTTVAR

- But if a timeout occurs
  - Retransmit unacknowledged segment with smallest seq #
  - Set timer to

    Timeout interval = 2 · previous timeout interval

  - If timer expires again, do the same thing:
    - Retransmit & double the timeout
  - This gives us exponentially longer time intervals
    - This is a form of congestion control
- Any other even that requires a timer reset
  - Set normal time interval (SRTT + 4 · RTTVAR)

TCP Fast Retransmit

- TCP uses pipelining
  - Will usually send many segments before receiving ACKs for them
- If a receiver detects a missing sequence #
  - It means out-of-order delivery or a lost segment
  - TCP does not send NAKs
  - Instead, acknowledge every segment with the last in-order seq #
  - Each segment received after a missing one will generate replies with duplicate ACKs

TCP Fast Retransmit

- Waiting for timeouts causes a delay in retransmission
  - Increases end-to-end latency
- But a sender can detect segment loss via duplicate ACKs
  - **Duplicate ACK**:
    Sender receives an ACK for a segment that was already ACKed
  - That means that a segment was received but not the sequentially next one
- If a sender receives **three duplicate ACKs**
  - Sender assumes the next segment was lost
    (it could have been received out of order but we're guessing that's unlikely since three segments after it have been received)
  - Performs a **fast retransmit**
    - Sends missing segment before the retransmission timer expires

GBN or SR?

- TCP looks like a Go-Back-N protocol
  - Sender only keeps track of smallest sequence # that was transmitted but not acknowledged

- But not completely…
  - GBN will retransmit *all* segments in the window on timeout
  - TCP will retransmit at most one segment (lowest #)
  - TCP will retransmit no segments if it gets ACKs for higher-numbered segments before a timeout
  - Most TCP receivers will hold out-of-order segments in a buffer

- We can call it a modified Go-Back-N

SACK: Selective Acknowledgements

- Enhancement to TCP to make it be a Selective Repeat protocol

- RFC 2018: TCP Selective Acknowledgement Options

- When receiving an out-of-order segment:
  - Send duplicate ACK segment (as before)
  - But append TCP option field containing range of data received
    - List of (*start byte, end byte*) values
  - Negotiated between hosts at the start of a connection
    - SACK may be used if both hosts support it

Flow Control

Flow control

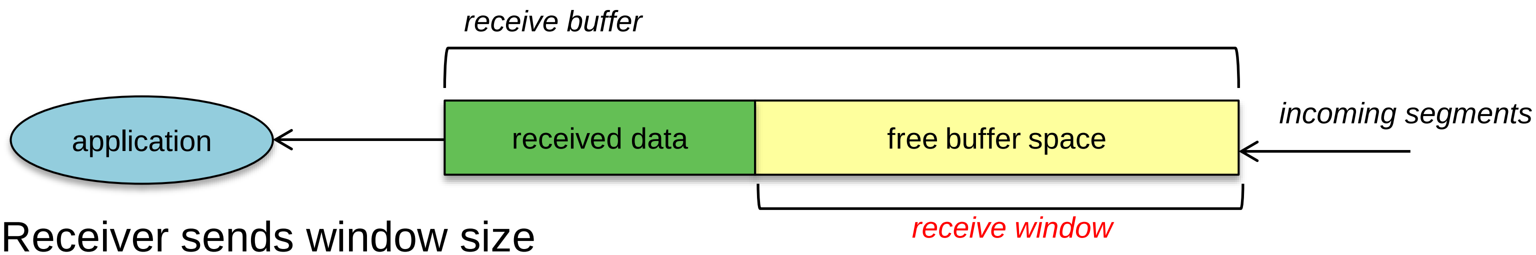
- Incoming data goes to receive buffer
- What if it comes in faster than the process reads it?
- We don’t want overflow!

- Flow control: match transmission rate with rate at which the app is reading data

Flow control

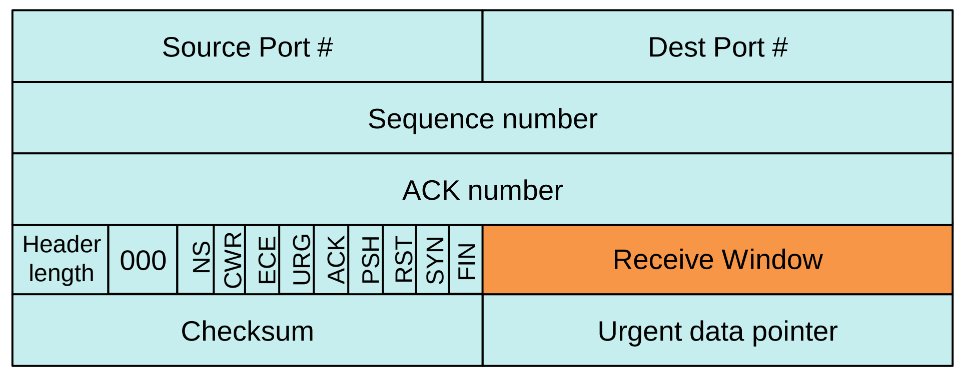
Receive window

Sender's idea of how much free buffer space is available at receiver



- Receiver sends window size to sender in reply segments

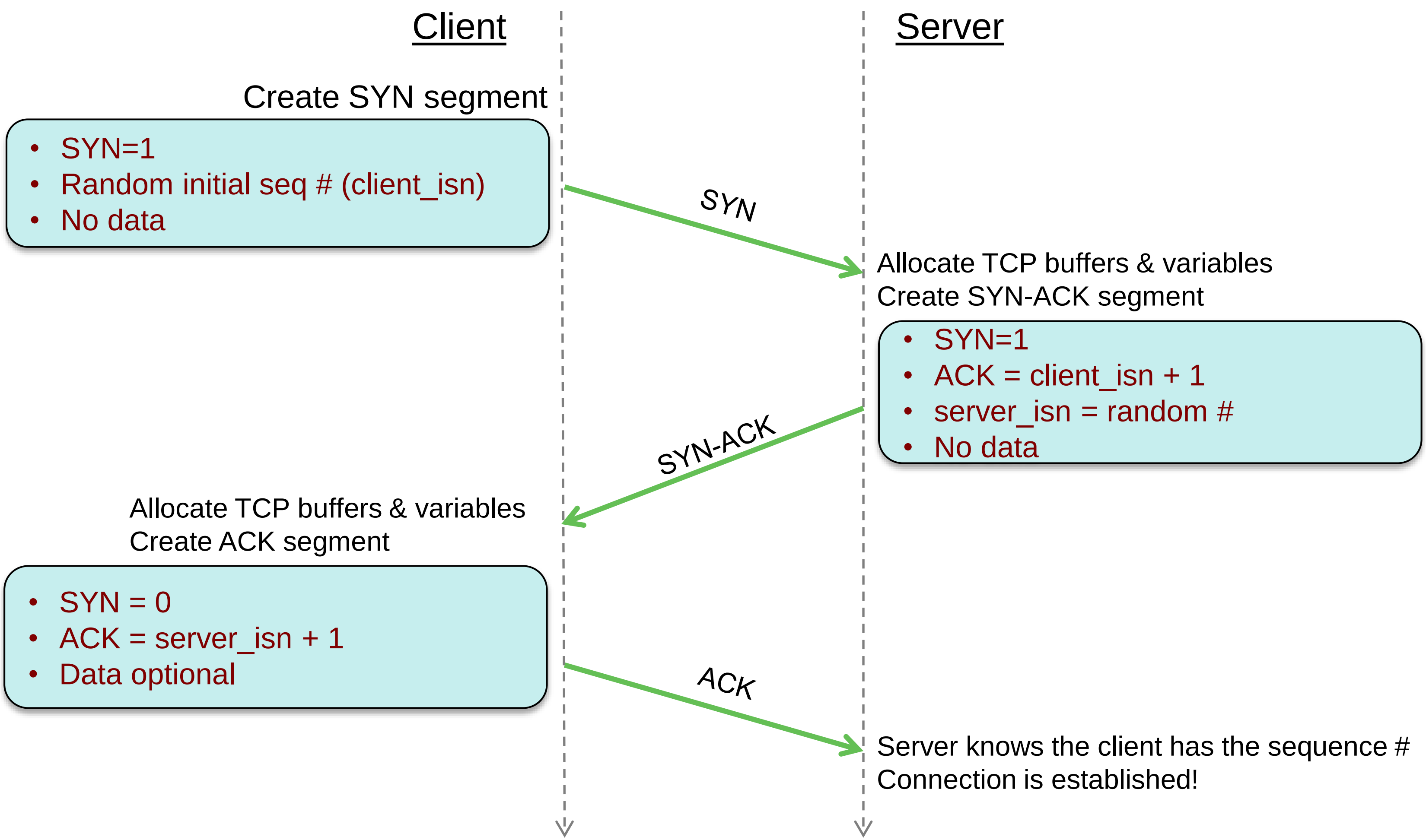
- If the receiver has no messages for the sender and the buffer was full, the sender won't know that the buffer is being emptied!



- **Probing**
  – If the sender sees the receive window = 0, it will periodically send messages with 1 byte of data
  – Receiver will not accept them if the window size is really 0
  – Eventually one of them will cause an ACK reporting a non-zero window

Connection Management

Connection setup: Three-way handshake



SYN Flooding

- An OS will allocate only a finite # of TCP buffers
- SYN Flooding attack
  - Send lots of SYN segments but never complete the handshake
  - The OS will not be able to accept connections until those time out
- SYN Cookies: Dealing with SYN flooding attacks
  - Do not allocate buffers & state when a SYN segment is received
  - Create initial sequence # =

    hash(src_addr, dest_addr, src_port, dest_port, SECRET)
  - When an ACK comes back, validate the ACK #

    Compute the hash as before & add 1
  - If valid, then allocate resources necessary for the connection & socket

MSS Announcement

- Remember the Maximum Segment Size (MSS)?
- For direct-attached networks
  - MSS = MTU of network interface – protocol headers
    - Ethernet MTU of 1500 bytes yields MSS of 1460 (1500-20-20)
- For destinations beyond the LAN (routing needed)
  - Use TCP Options field to set Maximum Segment Size
    - Set MSS in SYN segment
      - MSS may be obtained from PATH MTU discovery
    - Other side receives this and records it as MSS for sent messages.
    - It can respond with the MSS it wants to use for incoming messages in the SYN-ACK message
  - All IP routers must support MSS ≥ 536 bytes

Special cases

- What if the host receives a TCP segment where the port numbers or source address do not match any connection?
  - Host sends back a "reset" segment (RST = 1)
    "I don't have a socket for this"

- For UDP messages to non-receiving ports
  - Send back an ICMP message to the sending host

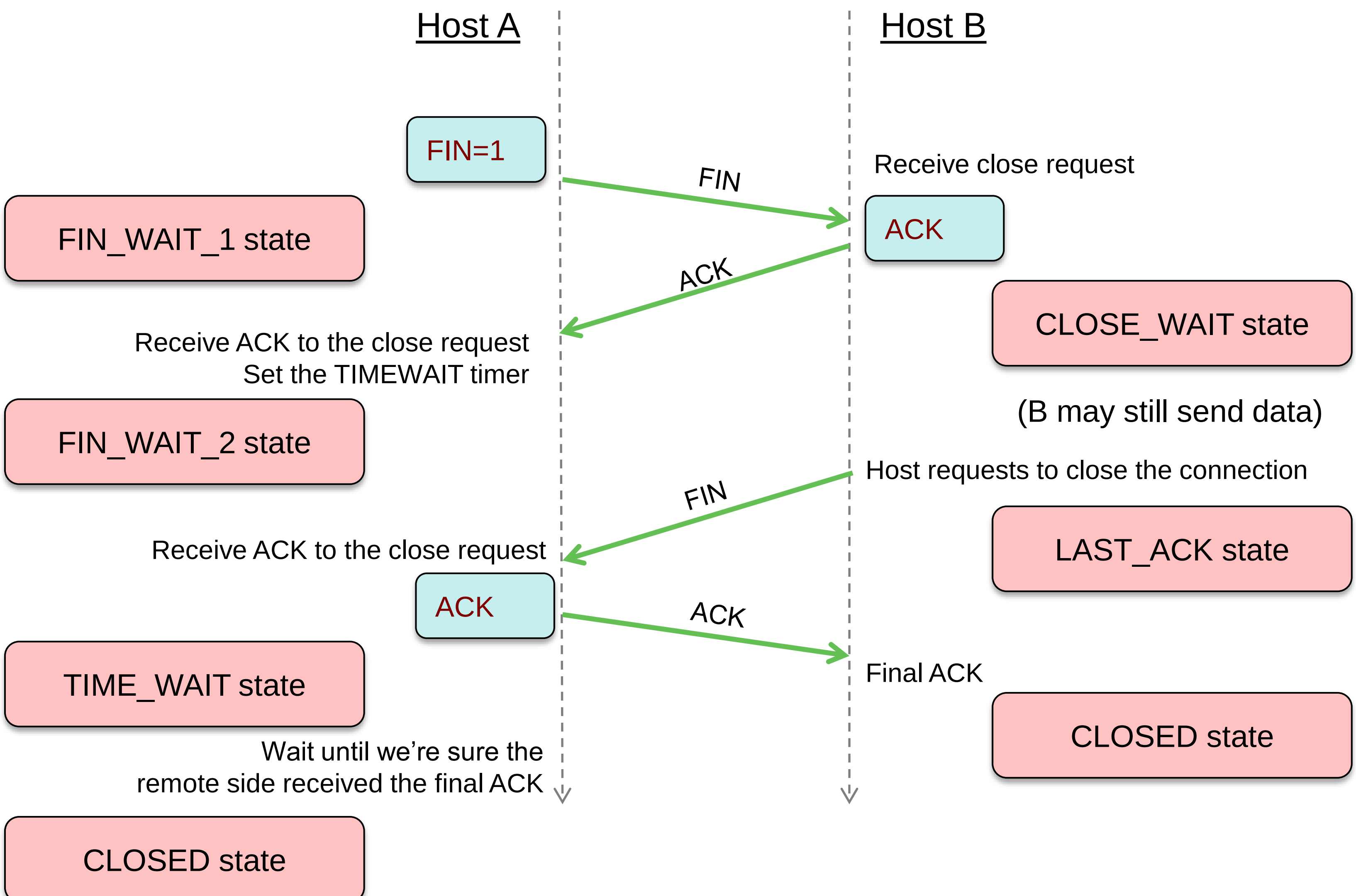
Connection teardown

- Either side can end a connection
- Buffers & state variables need to be freed
- Both sides agree to send no more messages

To close:

1. Send a TCP segment with the **FIN** bit set (FIN = Finish)
   - You are saying “I will not send any more data on this connection”
2. Other side acknowledges this
3. Other side then agrees to close the connection
   - Sends a TCP segment with the **FIN** bit set
4. You acknowledge receipt of this
   - Then wait (TIME_WAIT state) to ensure that your ACK had time to get to the other side and that any stray segments for the connection have been received
     – Wait time = 2 × maximum segment lifetime (timeout interval × 2)
     – Opportunity to resend final ACK in case it is lost

Connection teardown



TCP Congestion Control

Congestion control

- Congestion control goal

  Limit rate at which a sender sends traffic based on congestion in the network

  (Flow control goal was: limit traffic based on remote side's ability to process)

- Must use end-to-end mechanisms
  - The network gives us no information
  - We need to *infer* that the network is congested
  - Generally, more packet loss = more congestion

Regulating Rate: Congestion Window

- Window size = # bytes we can send without waiting for ACKs
- Receive Window (**`rwnd`**) – *flow control request from receiver*
  - # bytes that a receiver is willing to receive (reported in header)
- Congestion Window (**`cwnd`**) – *rate control by sender*
  - Window size to limit the rate at which TCP sender will transmit
- TCP will use window size = min(`rwnd`, `cwnd`)
  - These are per-connection values!

- How does a window regulate transmission rate?
  - If we ignore loss and delays, we transmit `cwnd` bytes before waiting
  - The time we wait is the round-trip time (RTT)

Transmission rate ≈ cwnd / RTT bytes/second

Basic mechanisms

- Timeout or three duplicate ACKs
  - Assume segment loss → decrease `cwnd` = *decrease sending rate*

- Sender receives expected ACKs
  - Assume no congestion → increase `cwnd` = *increase sending rate*

- ACKs pace the transmission of segments
  - ACKs trigger increase in `cwnd` size
  - If ACKs arrive slowly (slow network) → `cwnd` increases slowly
  - TCP is **self-clocking**
- **Bandwidth probing**
  - Increase rate in response to arriving ACKs
  - … until loss occurs; then back off and start probing (increasing rate) again

Basic Principle: Additive Increase (AI)

If we feel we have extra network capacity

- Increase window by 1 segment each RTT
  - If we successfully send `cwnd` bytes, increase window by 1 MSS
  - That means increase window fractionally for each ACK

$$\text{cwnd} = \text{cwnd} + [\ \text{MSS} \div (\text{cwnd}/\text{MSS})\ ]$$

- This is **Additive (linear) Increase**

Basic Principle: Multiplicative Decrease (MD)

If we feel we have congestion (timeout due to lost segment)

– Decrease cwnd by halving it

$$\text{cwnd} = \text{cwnd} \div 2$$

– This is **Multiplicative decrease**

Additive Increase / Multiplicative Decrease (AIMD)

AIMD is a *necessary* condition for TCP congestion control to be stable

TCP Congestion Control

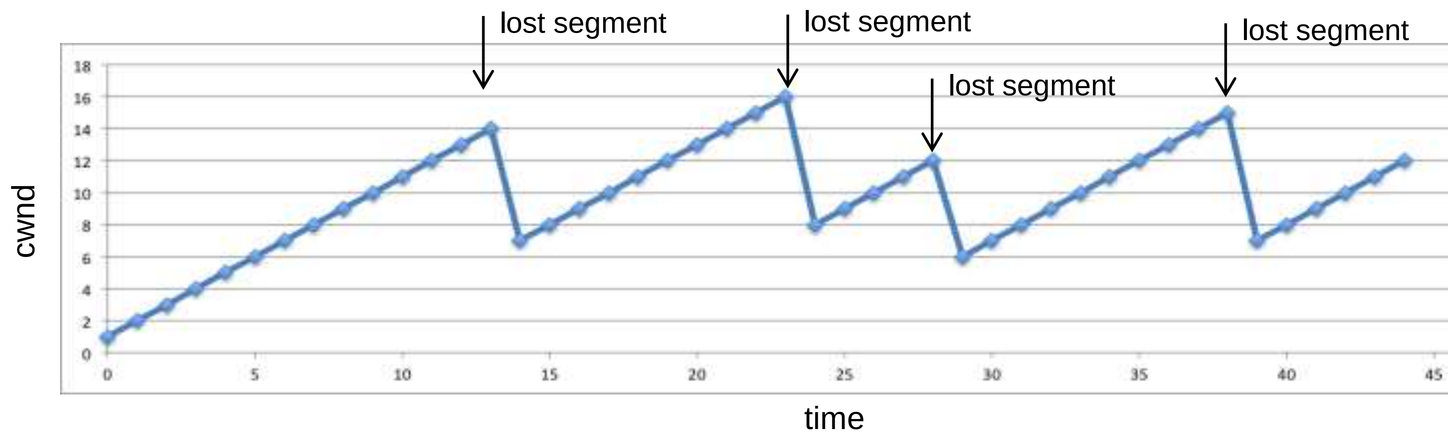
Three Parts:

1. Slow Start — REQUIRED
2. Congestion Avoidance — REQUIRED
3. Fast Recovery — RECOMMENDED

Speeding things up at the start

AIMD gives us linear ramps

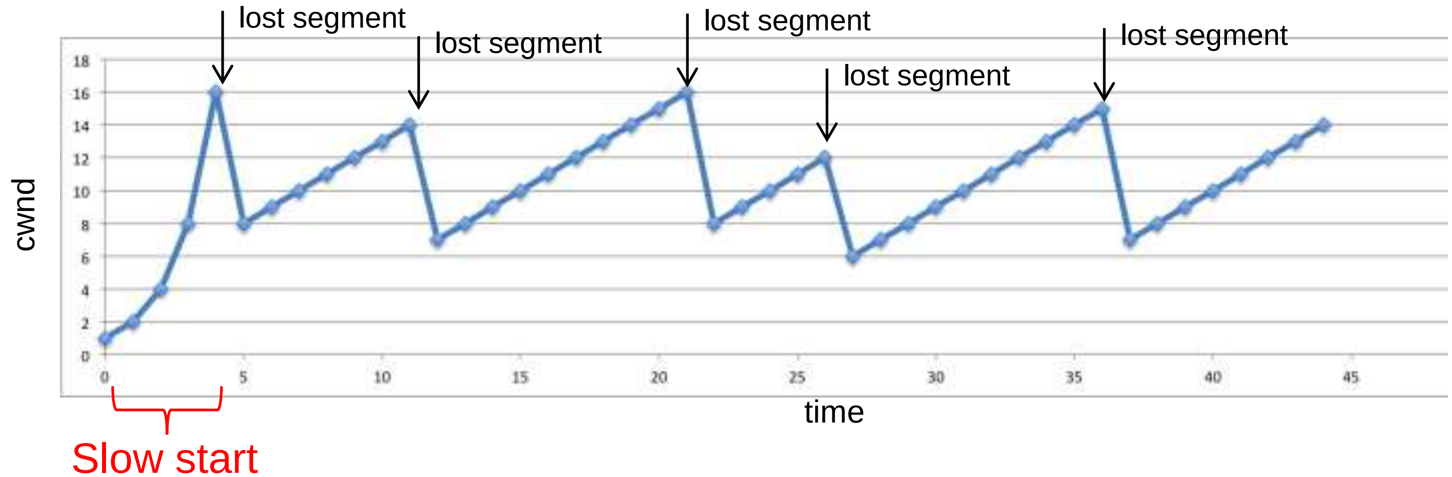
- Transmission follows a sawtooth pattern



- But it can take a long time to ramp up the transmission speed

TCP Slow Start

- Prevent the slow ramp at startup
- Start with an initial exponential increase in `cwnd` size



This is what **TCP Slow Start** is about … it's actually an *accelerated start*

- Avoid the slow start of a linear ramp
- … but it's still slower than just sending the `rwnd` # of bytes
- … but doing so might cause congestion and we won't know the threshold

TCP Slow Start

- Sender-based flow control
- Rate of acknowledgements determines rate of transmission
- For a new connection, initial `cwnd` = 1 MSS

  Example:

  If MSS = 1460 bytes and RTT = 90 ms

  Transmission rate ≈ 130 kbps

  This is stop-and-wait performance!

- Increase `cwnd` by 1 MSS for each acknowledged segment

  Start with 1 MSS (get 1 ACK)
  - Then `cwnd` = 2 MSS (get 2 ACKs)
  - Then `cwnd` = 4 MSS (get 4 ACKs)
  - Then `cwnd` = 8 MSS …

Two events bring us to this state:

1. Cold start (start of connection)
2. Timeout

- Transmission rate grows exponentially
  - *Doubles every RTT*

TCP Slow Start

- “Slow Start” actually grows quickly!
- When do we stop going faster?
  - **On timeout** (we assume this is due to congestion)
    - Sender sets `cwnd`=1 and restarts Slow Start process
    - Set ***slow start threshold***, **`ssthresh`** = `cwnd`/2
  - **When `cwnd` ≥ `ssthresh`**
    - switch to **Congestion Avoidance** mode (slow the ramp)
    - This is not set at cold start; we will time out
  - **When three duplicate ACKs received (following a normal ACK for a segment)**
    - Perform **Fast Retransmit** of segment
    - Enter **Fast Recovery** State

Congestion Avoidance

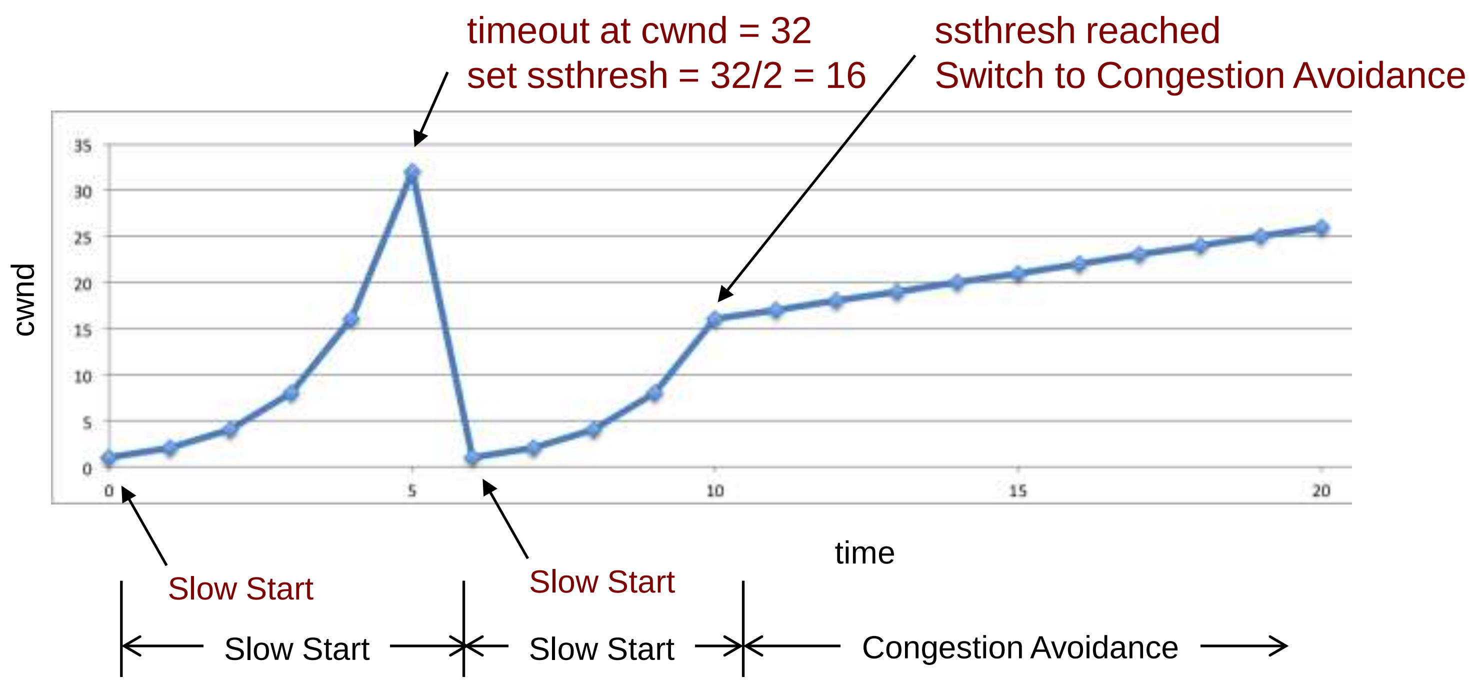
- `cwnd` is ½ of the size when we saw congestion
  - We think that's safe
  - … it worked before but doubling it gave a timeout – so we're close

- Increase rate additively: 1 MSS each RTT
  - # segments in window = `cwnd`/MSS
    - E.g., if MSS = 1460 bytes & `cwnd`= 23360 bytes, `cwnd`/MSS =16
  - Each ACK means we increase cwnd by MSS/(`cwnd`/MSS)
    - E.g., after 16 ACKs, cwnd increased by 1 MSS
      = increase cwnd by 1/16 MSS (~91 bytes) for each received ACK

- Now we have a linear growth in transmission speed

Slow Start + Congestion Avoidance

- Start with **Slow Start**
- On timeout, save `ssthresh`; restart **Slow Start**
- If `ssthresh` is reached, switch to **Congestion Avoidance**



Congestion Avoidance

- When do we stop increasing `cwnd`?
- When we get a timeout
  - Set `ssthresh` to ½ `cwnd` when the loss occurred
  - Set `cwnd` set to 1 MSS and do a Slow Start
- When we receive 3 duplicate ACKs
  - We're guessing segment loss BUT the network is delivering segments
  - Otherwise the receiver would not send ACKs
  - `ssthresh` = `cwnd` / 2
  - `cwnd` = `ssthresh` + (3 · MSS)
  - We essentially ½ our transmission rate
  - Enter **Fast Recovery** state

(3 · MSS) accounts for the three duplicate ACKs

Fast Recovery

- Fast Retransmit was used when duplicate ACKs received
  – Avoid waiting for a timeout
- Duplicate ACKs means data is flowing to the receiver
  – ACKs are generated only when a segment is received
- Might indicate that we don’t have congestion and the loss was a rare event.
- Don’t reduce flow abruptly by going into Slow Start
  – Adjust `cwnd` = `cwnd` / 2

Fast Recovery

- Increase `cwnd` by 1 MSS for each duplicate ACK received
  - Increase transmission rate exponentially – just like slow start
  - Each ACK means that the receiver received a segment *… data is flowing!*

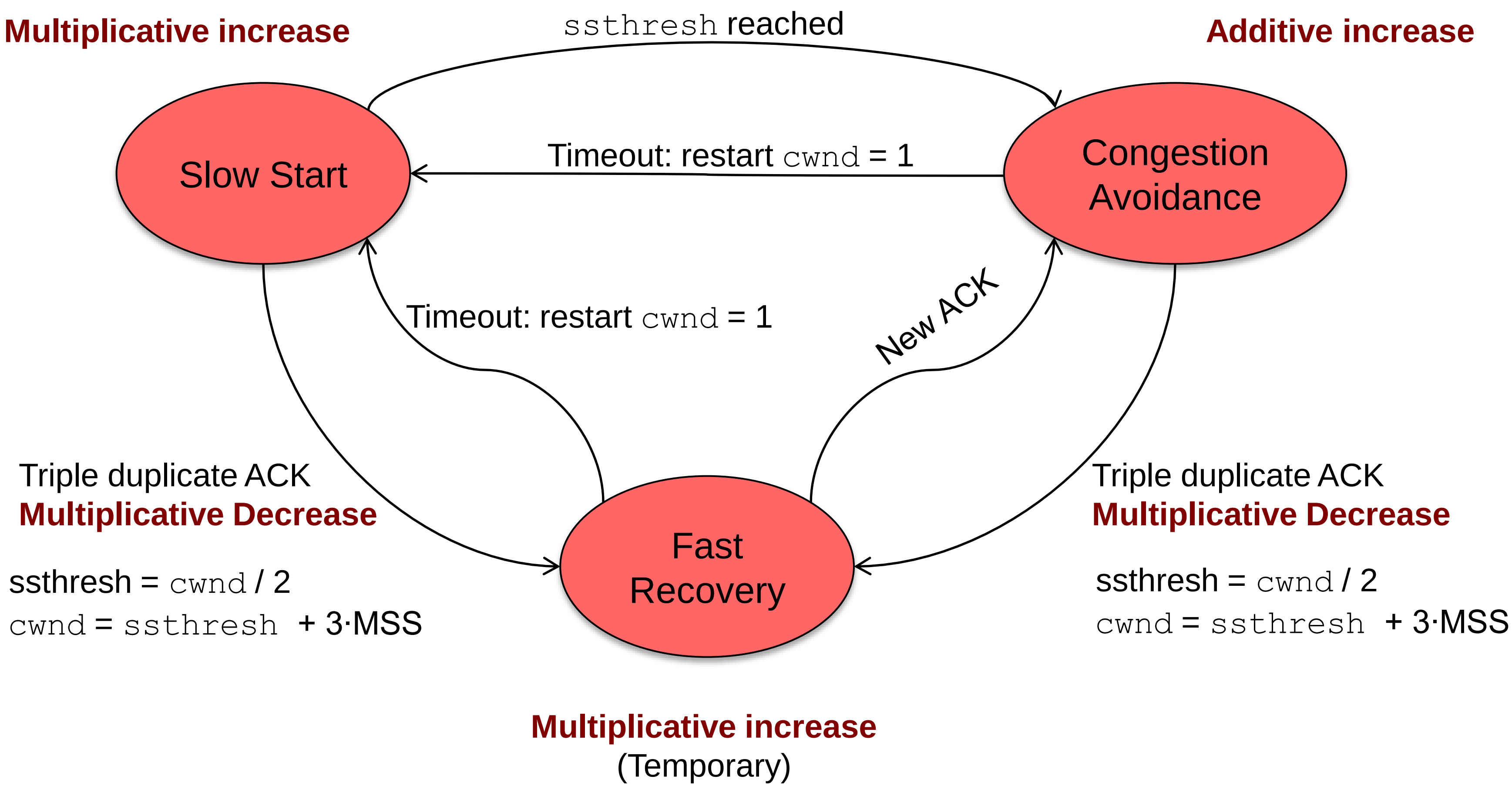
- When ACK arrives for the missing segment (non-duplicate ACK)
  - Reset `cwnd` to `ssthresh` (back to where it was)
  - Enter **Congestion Avoidance** state
    - Resumes transmission with linear growth of the window

- If timeout occurs
  - `ssthresh` = `cwnd` / 2
  - `cwnd` = 1
  - Do a **Slow Start**

Why the name?

- Why do we call it Fast Recovery?
  - Prior to its use, TCP would set `cwnd` = 1 and enter Slow Start for both timeouts as well as triple duplicate ACKs
- We try to distinguish casual packet loss from packet loss due to congestion

TCP congestion control state summary



Timeouts should be *rare*: we expect most segment losses to be detected by triple ACKs

TCP is effectively an **Additive Increase / Multiplicative Decrease** (**AIMD**) form of congestion control

The end