Reliable Transport I: Concepts and TCP Protocol

Brad Karp
UCL Computer Science

CS 3035/GZ01

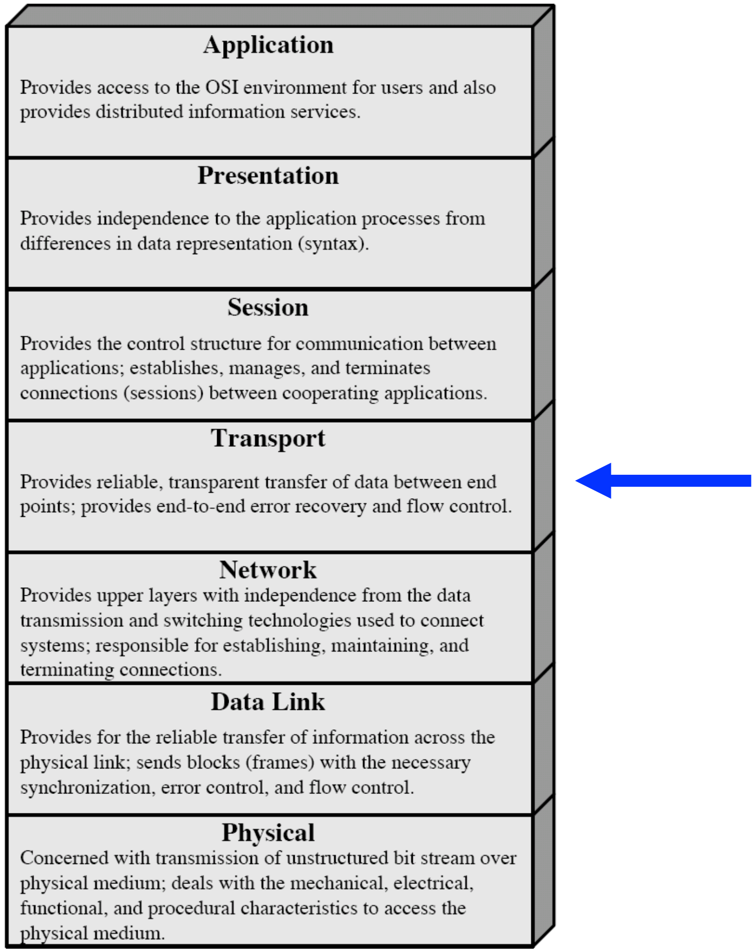
29th October 2013

Part I: Transport Concepts

- Layering context
- Transport goals
- Transport mechanisms

Context: Transport Layer

- Best-effort network layer
  - drops packets
  - delays packets
  - reorders packets
  - corrupts packet contents
- Many applications want reliable transport
  - all data reach receiver…
  - …in order they were sent
  - no data corrupted
  - “reliable byte stream”
- Need a transport protocol, *e.g.*, Internet’s Transmission Control Protocol (TCP)



Ports:
Identifying Senders and Receivers

- Host may run multiple, concurrent apps
- Typical layered multiplexing: transport protocol multiplexed by applications above
- Transport protocol must identify sending and receiving application instance
- Application instance ID: **port**
- Port owned by one application **instance** on host
- Servers often run on well-known ports
  - e.g., HTTP tcp/80, SMTP tcp/25, ssh tcp/22
- TCP port number: 16 bits, one each for sender and receiver

TCP: Connection-Oriented, Reliable Byte Stream Transport

- Sending application offers a sequence of bytes: d_0, d_1, d_2, …
- Receiving application sees all bytes arrive in same sequence: d_0, d_1, d_2…
  - not all applications need in-order behavior (e.g., ssh does, but does file transfer, really?)
  - result: reliable byte stream transport
- Each byte stream: connection, or flow
- Each connection uniquely identified by:
  - <sender IP, sender port, receiver IP, receiver port>

TCP's Many End-to-End Goals

- Recover from data loss
- Avoid receipt of duplicated data
- Preserve data ordering
- Provide integrity against corruption
- Avoid sending faster than receiver can accept data
- Prevent (most) third party hosts from originating connections as other hosts
- Avoid congesting network

Fundamental Problem: Ensuring At-Least-Once Delivery

- Network drops packets
- Strategy to ensure delivery:
  - Sender attaches unique number, or **nonce,** to each data packet sent; keeps copy of sent packet
  - Receiver returns acknowledgement (ACK) to sender for each data packet received, containing nonce
  - Sender sets timer on each transmission
    - if timer expires before ACK returns, retransmit that packet
    - if ACK returns, cancel timer, discard saved copy of that packet
  - Sender limits maximum number of retransmissions
- How long should retransmit timer be?

Fundamental Problem: Estimating RTT

- Expected time for ACK to return is **round-trip time (RTT)**
  - end-to-end delay for data to reach receiver and ACK to reach sender
  - propagation delay on links
  - serialization delay at each hop
  - queuing delay at routers
- Straw man: use fixed timer (e.g., 250 ms)
  - what if the route changes?
  - what if congestion occurs at one or more routers?
- Too small a value: needless retransmissions
- Too large a value: needless delay detecting loss

Fundamental Problem: Estimating RTT

- Expected time for ACK to return is **round-trip time (RTT)**
  - end-to-end delay for data to reach receiver and ACK to reach sender
  - propagation delay on links

Fixed timer violates end-to-end argument; details of link behavior should be left to link layer!

Hard-coded timers lead to **brittle behavior** as technology evolves

- Too small a value: needless retransmissions
- Too large a value: needless delay detecting loss

Estimating RTT: Exponentially Weighted Moving Average (EWMA)

- Measurements of RTT readily available
  - note time t when packet sent
  - corresponding ACK returns at time t'
  - RTT measurement = m = t'-t
- Single sample too brittle
  - queuing, routing dynamic
- Adapt over time, using EWMA:
  - measurements: m_0, m_1, m_2, ...
  - fractional weight for new measurement, α
  - $RTT_i = ((1-\alpha) \times RTT_{i-1} + \alpha \times m_i)$

Estimating RTT: Exponentially Weighted Moving Average (EWMA)

- Measurements of RTT readily available
  - note time t when packet sent
  - corresponding ACK returns at time t'
  - RTT measurement = m = t'-t

EWMA weights newest samples most

How to choose α? (TCP uses 1/8)

Is mean sufficient to capture RTT behavior over time? (more later)

  - fractional weight for new measurement, α
  - $RTT_i = ((1-\alpha) \times RTT_{i-1} + \alpha \times m_i)$

Retransmission and Duplicate Delivery

- When sender’s retransmit timer expires, two indistinguishable cases:
  - data packet dropped en route to receiver, or
  - ACK dropped en route to sender
- In both cases, sender retransmits
- In latter case, duplicate data packet reaches receiver!
- How to prevent receiver from passing duplicates to application?

Eliminating Duplicates: Exactly Once Delivery

- Each packet sent with unique nonce
- Straw man: receiver remembers nonces previously seen
  - if received packet seen before, drop, but resend ACK to sender
- How many **tombstones** must receiver store?
  - Longest gap between duplicates unknown!
  - Unbounded storage…
- Better plan: **sequence numbers**
  - sender marks each packet with monotonically increasing sequence number (non-random nonce)
  - sender includes greatest ACKed sequence number in its packets
  - receiver remembers only greatest received sequence number, drops received packets with smaller ones
  - still results in one tombstone per connection
  - (partial) fix: expire state at receiver after maximum retry delay

Eliminating Duplicates: Exactly Once Delivery

- Each packet sent with unique nonce
- Straw man: receiver remembers nonces previously seen

Doesn't guarantee delivery!

Properties:

If delivered, then only once.

If undelivered, sender will not think delivered.

If ACK not seen, data may have been delivered, but sender will not know.

  - sender includes greatest ACKed sequence number in its packets
  - receiver remembers only greatest received sequence number, drops received packets with smaller ones
  - still results in one tombstone per connection
  - (partial) fix: expire state at receiver after maximum retry delay

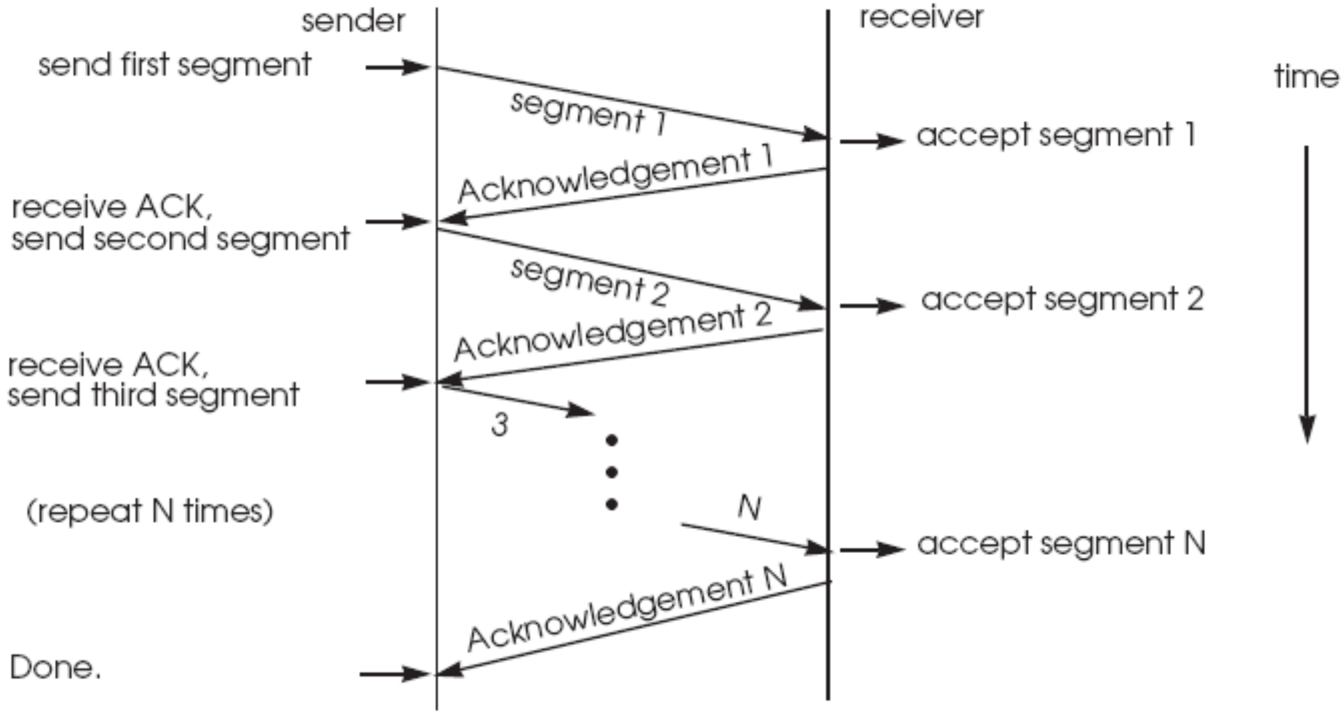
End-to-End Integrity

- Achieved by using transport checksum
- Protects against things link-layer reliability cannot:
  - router memory corruption, software bugs, &c.
- Covers data in packet, transport protocol header
- Also should cover layer-3 source and destination!
  - misdelivered packet should not be inserted into data stream at receiver, nor should be acknowledged
  - receiver drops packets w/failed transport checksum
  - TCP "pseudo header" covers IP source and destination (more later)

Segmentation and Reassembly

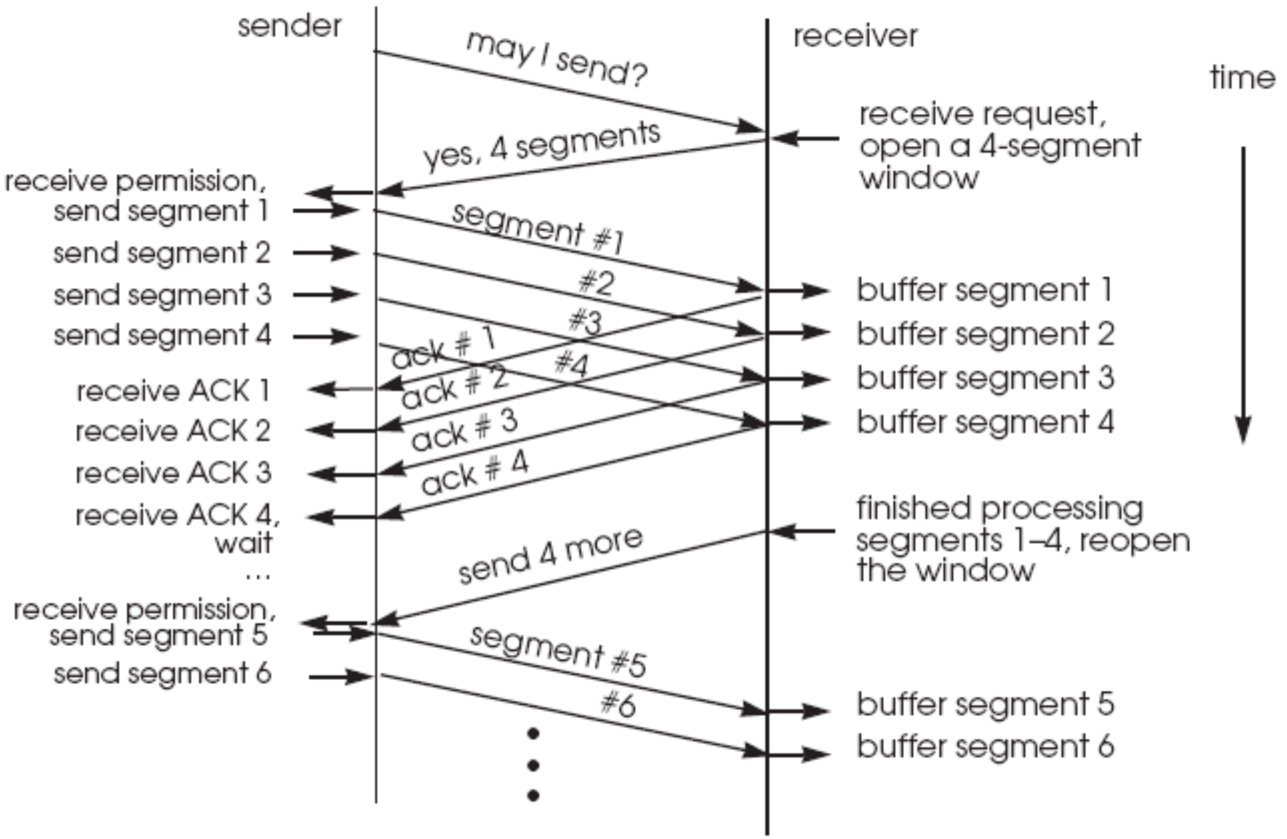
- Application data unbounded in length
- Link layers typically enforce maximum length
- Transport layer must
  - at sender, segment data too long for one packet into multiple packets
  - at receiver, reassemble these packets into original data
- Segmentation: divide into packets; mark each with range of bytes in original data
- Reassembly: buffer received packets in correct order; track which have arrived; pass to application only when all received

Window-Based Flow Control: Motivation



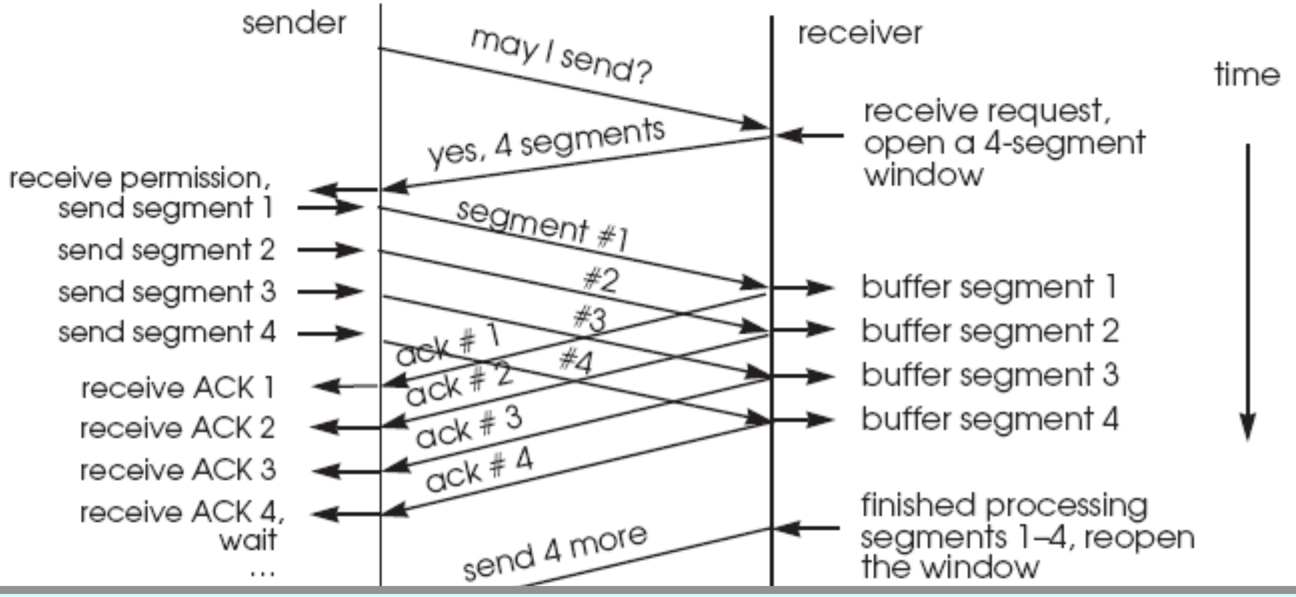
- Suppose sender sends one packet, awaits ACK, repeats…
- Result: one packet sent per RTT
- e.g., 70 ms RTT, 1500-byte packets
  - **Max throughput: 171 Kbps**

Fixed Window-Based Flow Control



- Pipeline transmissions to "keep pipe full"; overlap ACKs with data
- Sender sends **window** of packets sequentially, without awaiting ACKs
- Sender retains packets until they are ACKed, tracks which have been ACKed
- Sender sets retransmit timer for each window; when expires, resends all unACKed packets in window

Fixed Window-Based Flow Control



1 RTT idle time between grant of new window and arrival of data at receiver

Better approach, used by TCP: **sliding window,** extends on-the-fly as ACKs return; no idle time!

ACKs

- Sender retains packets until they are ACKed, tracks which have been ACKed
- Sender sets retransmit timer for each window; when expires, resends all unACKed packets in window

Choosing Window Size: Bandwidth-Delay Product

- How large a window is required at sender to keep the pipe full?
- Network bottleneck: point of slowest rate along path between sender and receiver
- To keep pipe full
  - window size $\geq$ RTT $\times$ bottleneck rate
- Window too small: can't fill pipe
- Window too large: unnecessary network load/queuing/loss

Choosing Window Size: Bandwidth-Delay Product

- How large a window is required at sender to keep the pipe full?
- Network bottleneck: point of slowest rate along path between sender and receiver

Goal: window size = RTT × bottleneck rate

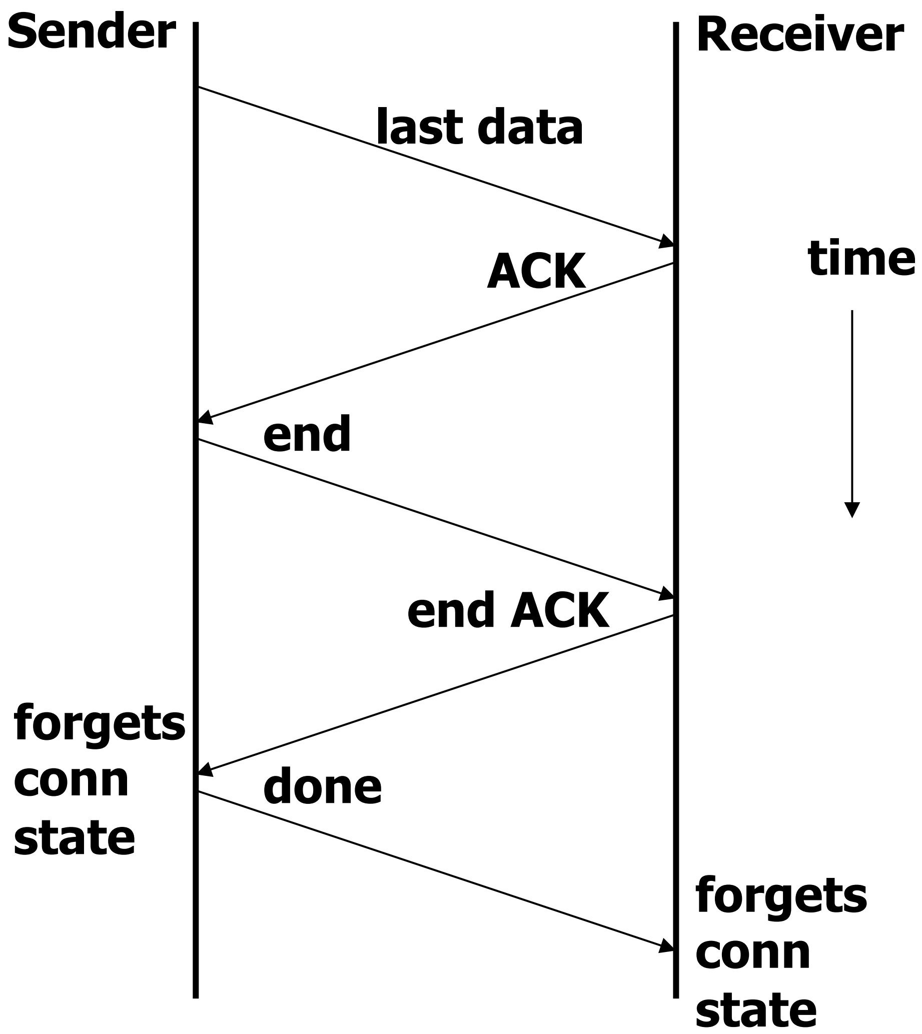
e.g., to achieve bottleneck rate of 1 Mbps, across a 70 ms RTT, need window size:

$$W = (10^6 \text{ bps} \times .07 \text{ s}) = 70 \text{ Kbits} = 8.75 \text{ KB}$$

- Window too large: unnecessary network load/queuing/loss

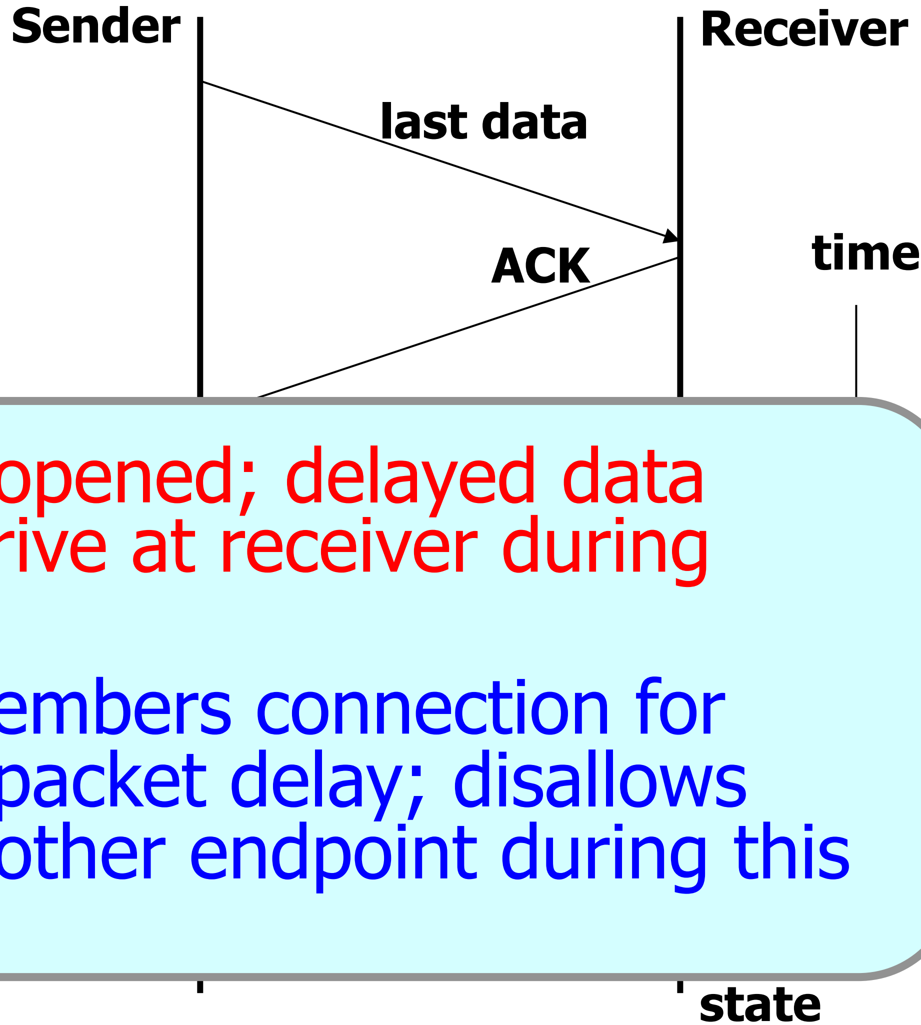
Closing of Connections

- Connection life cycle:
  - Open connection
  - Send/receive data
  - Close connection
- Criteria for connection close:
  - Receiver must know all data received
  - Sender and receiver must **agree** last packet reached receiver, and connection ended



Closing of Connections

- Connection life cycle:
  - Open connection
  - Send/receive data
  - Close connection



Risk: new connection opened; delayed data from old connection arrive at receiver during new one

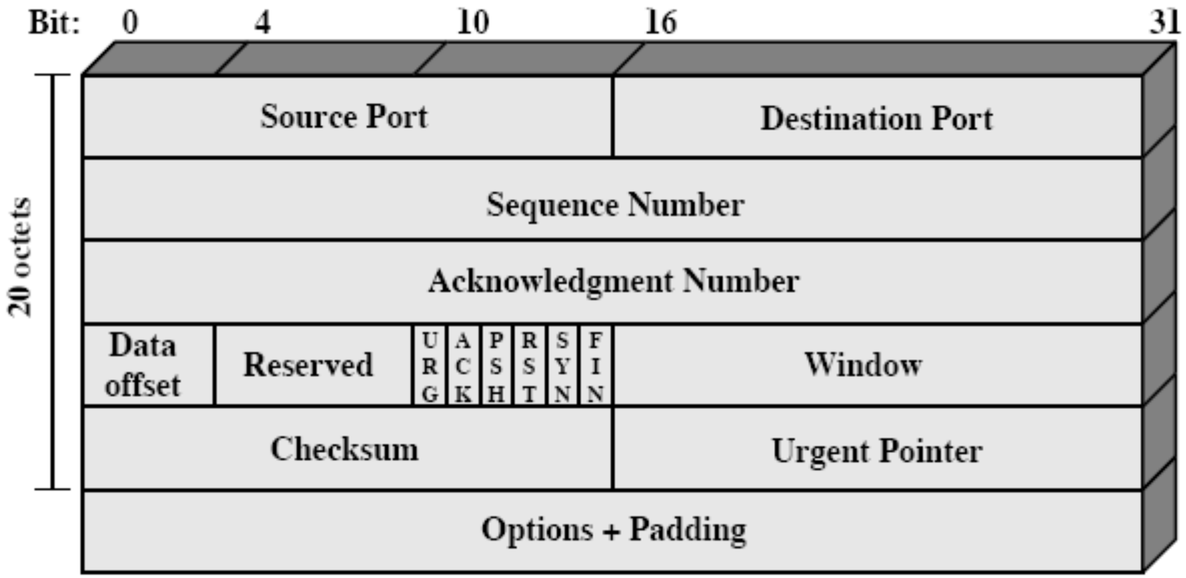
Fix: one endpoint remembers connection for longer than maximum packet delay; disallows new connections from other endpoint during this period

receiver, and
connection ended

Part II: TCP Protocol

- Packet header format
- Connection establishment
- Data transmission
- Retransmit timeouts
- RTT estimator
- AIMD Congestion control
- Throughput, loss, and RTT equation
- Connection teardown
- Protocol state machine

TCP Packet Header



- TCP packet: IP header + TCP header + data
- TCP header: 20 bytes long
- Checksum covers header + "pseudo header"
  - IP header source and destination addresses, protocol
  - Length of TCP segment (TCP header + data)

TCP Header Details

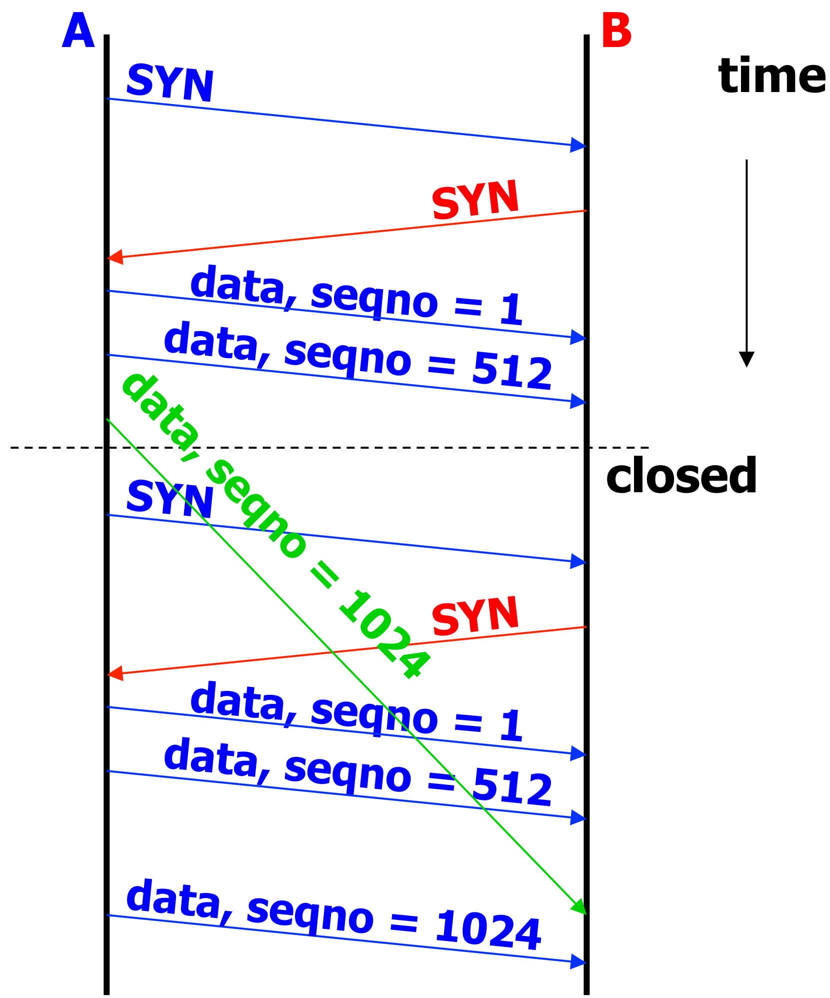
- Connections inherently bidirectional; all TCP headers carry both data and ACK sequence numbers
- 32-bit sequence numbers are in units of bytes
- Source and destination ports
  - multiplexing of TCP by applications
  - UNIX: local ports below 1024 reserved (only root may use them)
- Window: advertisement of number of bytes advertiser willing to accept

TCP Connection Establishment: Motivation

- Goals:
  - Start TCP connection between two hosts
  - Avoid mixing data from old connection in new connection
  - Avoid confusing previous connection attempts with current one
  - Prevent (most) third parties from impersonating **(spoofing)** one endpoint
- SYN packets (SYN flag in TCP header set) used to establish connections
- Use retransmission timer to recover from lost SYNs
- What protocol meets above goals?

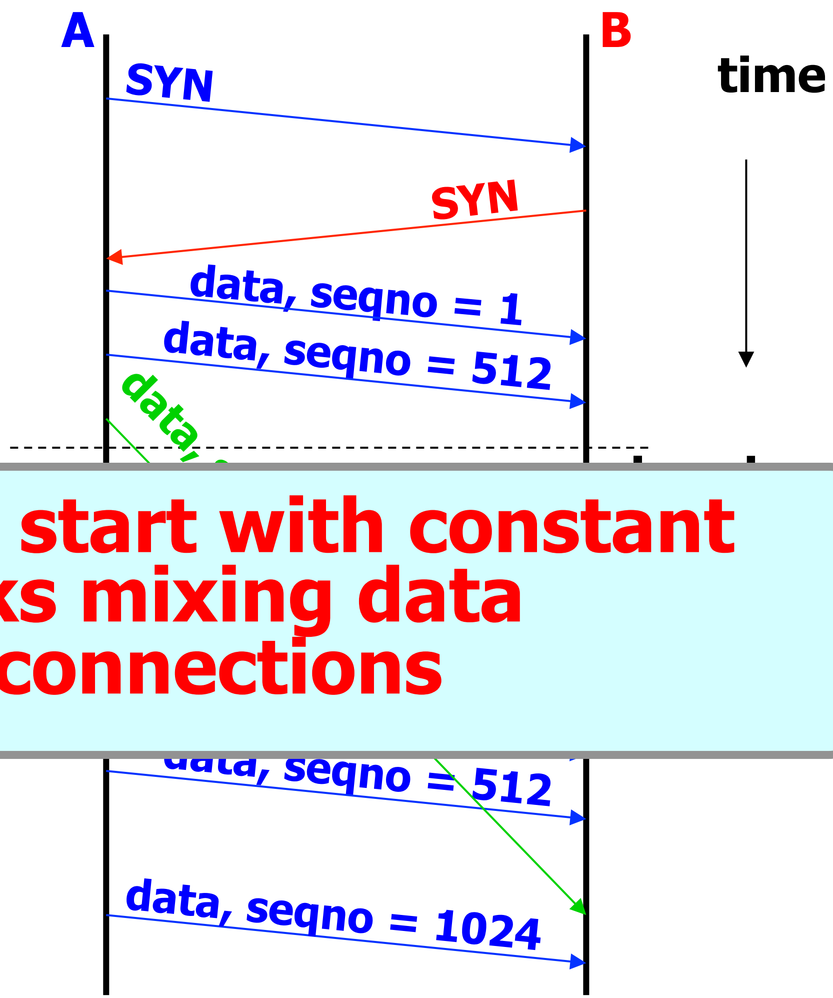
TCP Connection Establishment: Non-Solution (I)

- Use two-way handshake
- A sends SYN to B
- B accepts by returning SYN to A
- A retransmits SYN if not received
- A and B can ignore duplicate SYNs after connection established
- What about delayed data packets from old connection?



TCP Connection Establishment: Non-Solution (I)

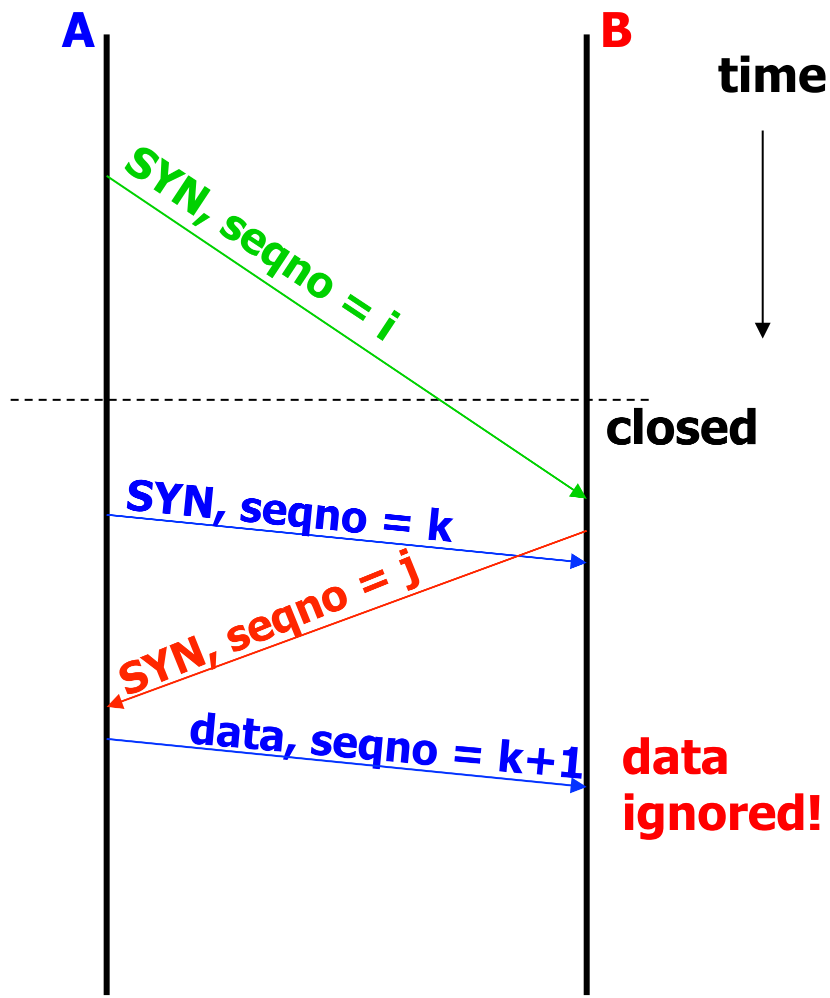
- Use two-way handshake
- A sends SYN to B
- B accepts by returning SYN to A
- A retransmits SYN if not received
- What about delayed data packets from old connection?



Connections shouldn't start with constant sequence number; risks mixing data between old and new connections

TCP Connection Establishment: Non-Solution (II)

- Two-way handshake, as before
- But enclose random initial sequence numbers on SYNs
- What about delayed SYNs from old connection?
  - A wrongly believes connection successfully established
  - **B will drop all of A's data!**

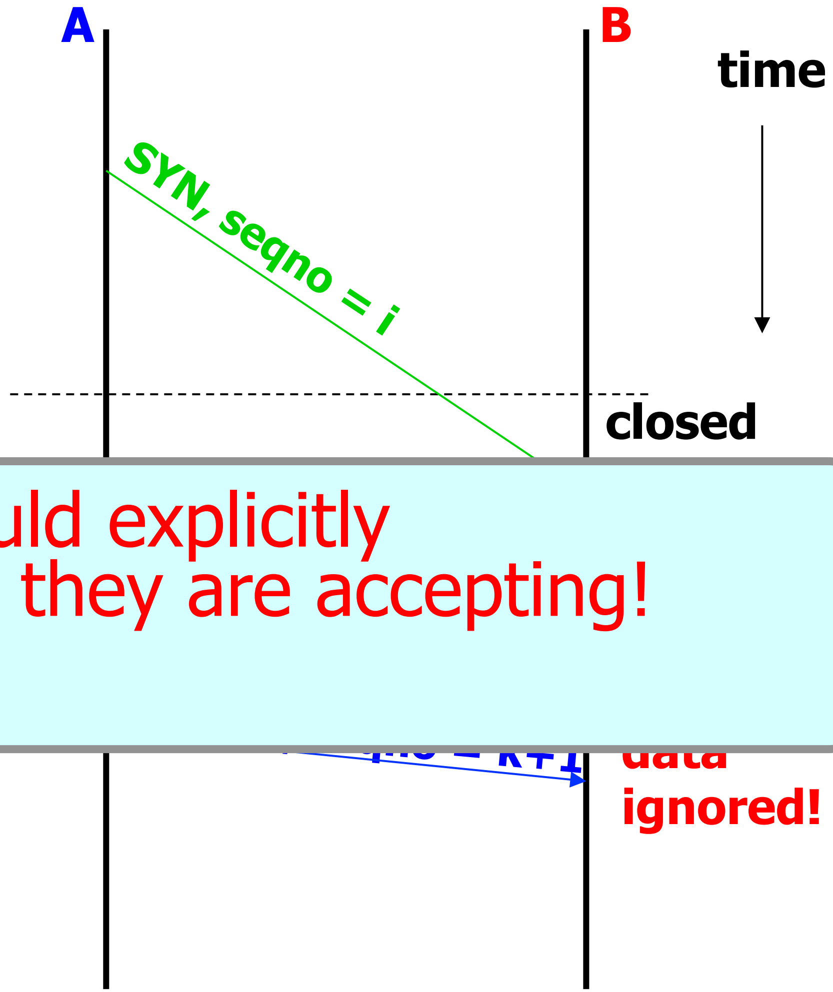


TCP Connection Establishment: Non-Solution (II)

- Two-way handshake, as before
- But enclose random initial sequence numbers on SYNs
- What about delayed

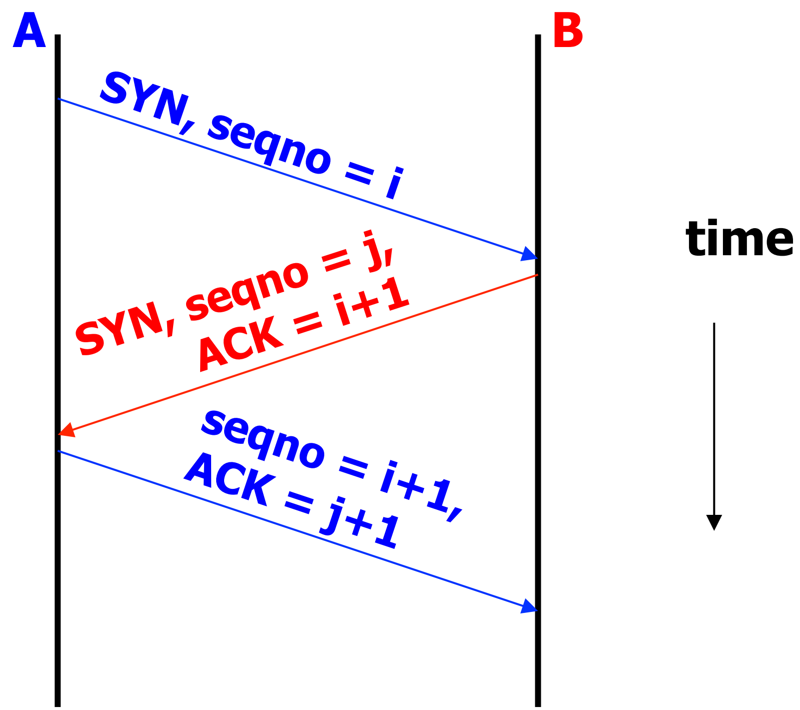
Connection attempts should explicitly acknowledge **which** SYN they are accepting!

connection successfully established
  - **B will drop all of A's data!**



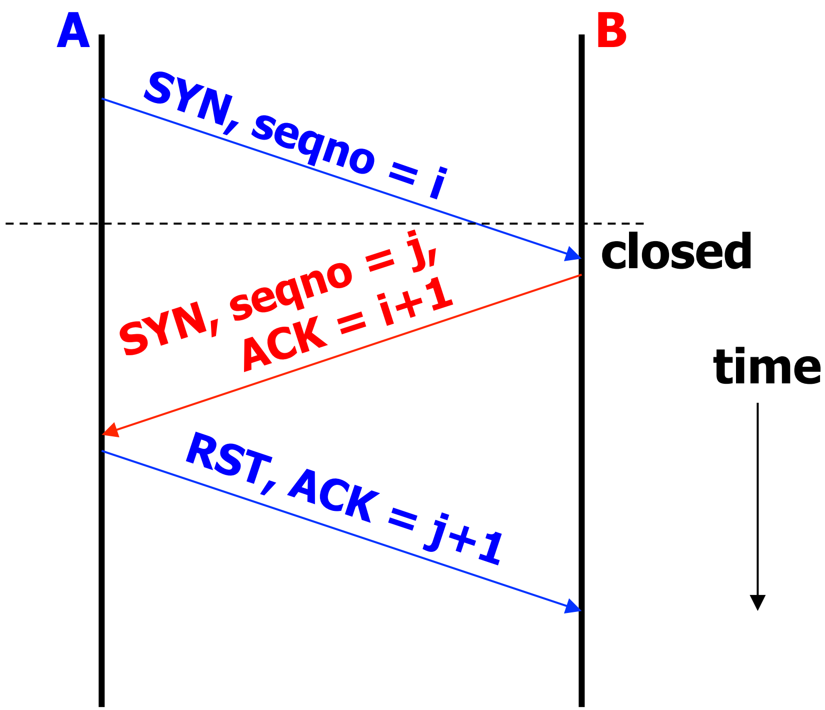
TCP Connection Establishment: 3-Way Handshake

- Set SYN on connection request
- Each side chooses random initial sequence number
- Each side explicitly ACKs the sequence number of the SYN it's responding to



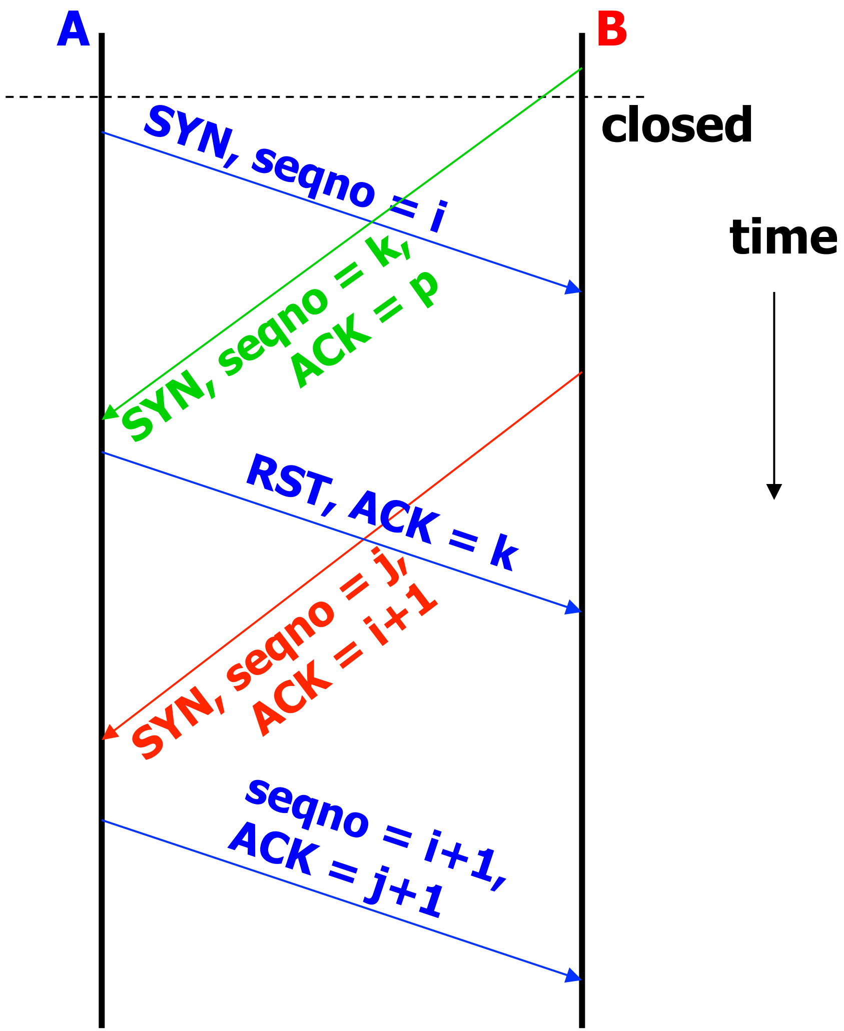
Robustness of 3-Way Handshake: Delayed SYN

- Suppose A's SYN i delayed, arrives at B after connection closed
- B responds with SYN/ ACK for i+1
- A doesn't recognize i +1; responds with **reset,** RST flag set in TCP header
- A rejects connection



Robustness of 3-Way Handshake: Delayed SYN/ACK

- A attempts connection to B
- Suppose B's SYN k/ ACK p delayed, arrives at A during new connection attempt
- A rejects SYN k; sends RST to B
- Connection from A to B succeeds unimpeded



Robustness of 3-Way Handshake: Source Spoofing

- Suppose host B trusts host A, based on A's IP address
  - e.g., allows any account creation request from host A
- Adversary M may not control host A, but may seek to impersonate, or **spoof,** host A
  - Adversary may not need to receive data from B; only send data (e.g., "create an account l33thax0r")
- Can M establish a connection to B as A?

A

B

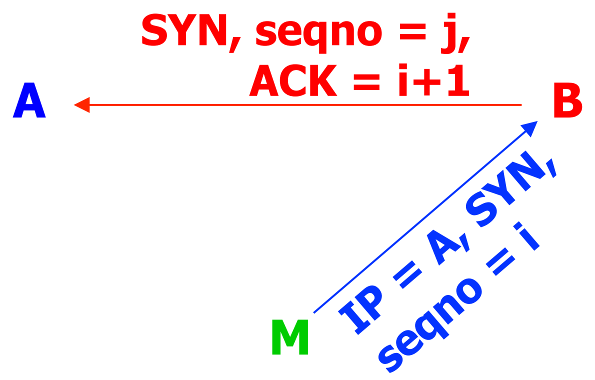
M

Robustness of 3-Way Handshake: Source Spoofing

- Suppose host B trusts host A, based on A's IP address
  - e.g., allows any account creation request from host A
- Adversary M may not control host A, but may seek to impersonate, or **spoof,** host A
  - Adversary may not need to receive data from B; only send data (e.g., "create an account l33thax0r")
- Can M establish a connection to B as A?

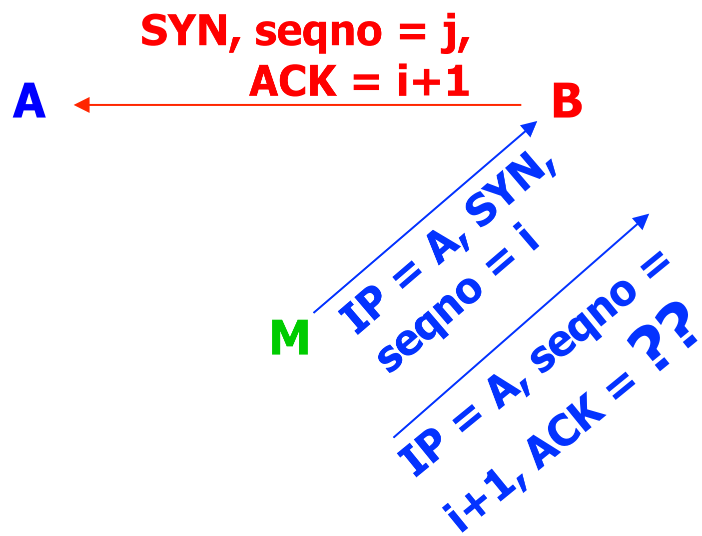
Robustness of 3-Way Handshake: Source Spoofing

- Suppose host B trusts host A, based on A's IP address
  - e.g., allows any account creation request from host A
- Adversary M may not control host A, but may seek to impersonate, or **spoof,** host A
  - Adversary may not need to receive data from B; only send data (e.g., "create an account l33thax0r")
- Can M establish a connection to B as A?



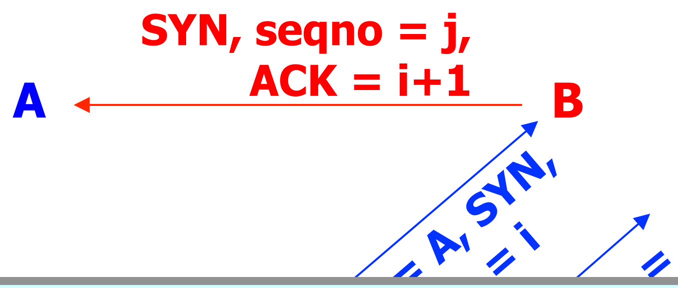
Robustness of 3-Way Handshake: Source Spoofing

- Suppose host B trusts host A, based on A's IP address
  - e.g., allows any account creation request from host A
- Adversary M may not control host A, but may seek to impersonate, or **spoof,** host A
  - Adversary may not need to receive data from B; only send data (e.g., "create an account l33thax0r")
- Can M establish a connection to B as A?



Robustness of 3-Way Handshake: Source Spoofing

- Suppose host B trusts host A, based on A's IP address
  - e.g., allows any account creation request from host A
- Adversary M may not



Unless he is on path between A and B, adversary cannot spoof A to B or vice-versa!

Why: **random ISNs on SYNs**

send data (e.g., "create an account l33thax0r")

- Can M establish a connection to B as A?

TCP: Data Transmission (I)

- Each byte numbered sequentially, mod 2^{32}
- Sender buffers data in case retransmission required
- Receiver buffers data for in-order reassembly
- Sequence number (seqno) field in TCP header indicates first user payload byte in packet
- Receiver indicates receive window size explicitly to sender in window field in TCP header
  – corresponds to available buffer space at receiver

TCP: Data Transmission (II)

- Sender's transmit window size: amount of buffer space at sender
- Sender uses window that is minimum of send and receive window sizes
- Receiver sends cumulative ACKs
  - ACK number in TCP header names highest contiguous byte number received thus far, +1
  - one ACK per received packet, OR
  - Delayed ACK also possible: receiver batches ACKs, sends one for every pair of data packets (200 ms max delay)
- Current window at sender:
  - low byte advances as packets sent
  - high byte advances as receive window updates arrive

Outline

- Packet header format
- Connection establishment
- Data transmission
- Retransmit timeouts
- RTT estimator
- AIMD Congestion control
- Throughput, loss, and RTT equation
- Connection teardown
- Protocol state machine

TCP: Retransmit Timeouts

- Sender sets timer for each sent packet
  - when ACK returns, timer canceled
  - if timer expires before ACK returns, packet resent
- Expected time for ACK to return: RTT
- TCP estimates round-trip time using EWMA
  - measurements m_i from timed packet/ACK pairs
  - $RTT_i = ((1-\alpha) \times RTT_{i-1} + \alpha \times m_i)$
  - Retransmit timeout: $RTO_i = \beta \times RTT_i$
  - original TCP: $\beta = 2$
- Is this accurate enough?
  - Recall dangers of too-short and too-long RTT estimates from previous lecture

Mean and Variance: Jacobson's RTT Estimator

- Above link load of 30% at router, $\beta \times RTT_i$ will retransmit too early!
- Response to increasing load: waste bandwidth on duplicate packets
- Result: **congestion collapse!**
- [Jacobson 88]: estimate v_i, mean deviation (EWMA of $|m_i - RTT_i|$), stand-in for variance

$$v_i = v_{i-1} \times (1-\gamma) + \gamma \times |m_i - RTT_i|$$

- Use $RTO_i = RTT_i + 4v_i$

Mean and Variance: Jacobson's RTT Estimator

- Above link load of 30% at router, $\beta \times RTT_i$ will retransmit too early!
- Response to increasing load: waste bandwidth on duplicate packets

Mean and Variance RTT estimator used by all modern TCPs

for variance

$$v_i = v_{i-1} \times (1-\gamma) + \gamma \times |m_i - RTT_i|$$

- Use $RTO_i = RTT_i + 4v_i$

Reminder:
Reading for Next Lecture

- Jacboson, V., Congestion Avoidance and Control
- Read before Thursday's lecture
- A true classic in networking—one of the most influential ideas in the area
- Paper is available in PDF on calendar page of class web site