# Lecture 5: The Transport Layer

**Kyle Jamieson**

COS 461: Computer Networks

# Transport Layer: Context & Motivation


- Most applications want to exchange messages between different remote processes

- Further, many applications want a **reliable stream of bytes between different remote processes**

# Transport Protocols

- Provide **logical communication** between **remote application processes**
  - Sender application divides a message into segments
  - Receiver application reassembles segments into message

- Transport layer services
  - (De)multiplexing packets
  - Detecting corrupted data
  - **Optional:** reliable byte stream delivery, flow control, congestion avoidance…

# User Datagram Protocol (UDP)

- Lightweight communication between processes
  - Send and receive messages
  - Avoid overhead of ordered, reliable delivery
    - No connection setup delay, no in-kernel connection state
- Used by popular apps
  - Query/response for DNS
  - Real-time data in VoIP

**8 byte header**

| | |
|---|---|
| SRC port | DST port |
| checksum | length |
| DATA | |

# Advantages of UDP

- Fine-grain control
  - UDP sends as soon as the application writes
- No connection set-up delay
  - UDP sends without establishing a connection
- No connection state in host OS
  - No buffers, parameters, sequence #s, etc.
- Small header overhead
  - UDP header is only eight-bytes long

# Identifying Sender and Receiver Apps

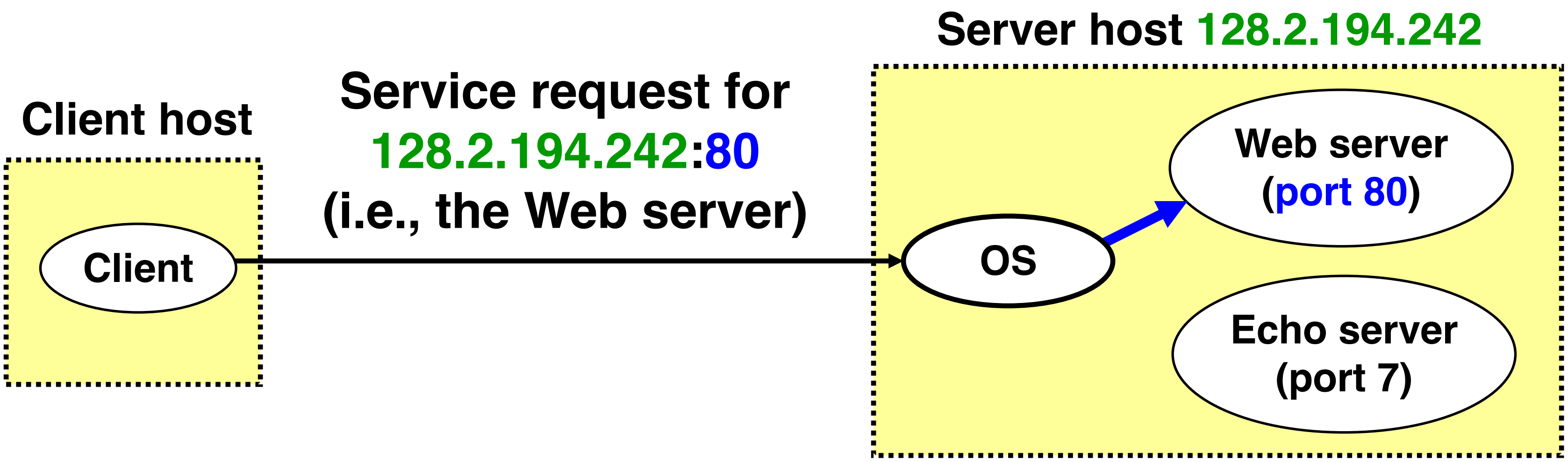
- Host may run multiple, concurrent applications

- Typical layered multiplexing: transport protocol **multiplexed (shared)** by applications above

- Transport protocol must identify sending and receiving application instance

- Application instance identifier: **port**
  - Port owned by one application **instance** on host

# Demultiplexing with Ports



# Transmission Control Protocol (TCP)

- Reliable byte stream service
  - **all data** reach receiver: **in order** they were sent, with **no data corrupted**
- Reliable, in-order delivery
  - Corruption: checksums
  - Detect loss/reordering: sequence numbers
  - Reliable delivery: acknowledgments and retransmissions
- Connection oriented
  - Explicit set-up and tear-down of TCP connection
- Flow control
  - Prevent overflow of the receiver's buffer space
- Congestion control
  - Adapt to network congestion for the greater good

# Outline

**Today:**

- **Fundamentals, Data transmission**
- Connection establishment

**Lecture 6:**

- Retransmit timeouts
- RTT estimator
- Slow Start and Self-clocking
- AIMD Congestion control

# Problem:

**Reliable (*i.e.*, Exactly Once) Delivery, over an Unreliable Network**

# An Analogy

- Alice is saying something to Bob
  - What if Bob couldn't understand Alice?
    - Bob asks Alice to repeat what she said
- What if Bob hasn't heard Alice for a while?
  - Is Alice just being quiet? Or, has he lost reception?
- How does Alice know her words are understood?
  - How long should she just keep on talking?
  - Maybe Bob should periodically say "uh huh"
    - … or Bob should ask "Can you hear me now?"

# Take-Aways from the Example

- Acknowledgments from receiver
  - Positive: “okay” or “uh huh” or “ACK”
  - Negative: “please repeat that” or “NACK”
- Retransmission by the sender
  - After *not* receiving an “ACK”
  - After receiving a “NACK
  - You can use both (as TCP does implicitly)
- Timeout by the sender (“stop and wait”)
  - Don’t wait forever without some acknowledgment

# Challenges of Reliable Data Transfer

- Over a network that **may cause bit errors**
  - Receiver detects errors and requests retransmission
- Over a **lossy** network with bit errors
  - Some **packets missing**, others corrupted
  - Receiver cannot easily detect loss
- Over a network that **may reorder packets**
  - How can the receiver distinguish loss from out of order delivery?

# Automatic Repeat Request (ARQ): Ensuring At-Least-Once Delivery

- **Sender** attaches a **unique number** (*nonce*) to each data packet sent; keeps copy of sent packet
- **Receiver** returns acknowledgement (ACK) for each data packet received, **containing nonce**
- Sender sets a timer on each transmission
  - timer expires before ACK returns → retransmit the packet
  - ACK returns before timer expires → cancel timer, discard saved packet copy

# Fundamental Problem: Estimating RTT

- **Round-Trip Time (RTT):** the end-to-end delay for data to reach receiver and ACK to reach sender, comprised of:
  - propagation delay on links
  - serialization delay at each hop
  - queuing delay at routers

- Design alternative: use fixed timer (*e.g.*, 250 ms)
  - What if **the route changes?**
  - What if **congestion at one or more routers?**

# 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

- Use just a single sample?
  **Too brittle (queuing, routing dynamics)**

- **Instead: adapt over time,** using EWMA:
  - **measurements:** $m_0, m_1, m_2, \ldots$
  - fractional weight for new measurement, $\alpha$
  - $RTT_i = ((1-\alpha) \times RTT_{i-1} + \alpha \times m_i)$

# Retransmission and Duplicate Delivery

- When sender's retransmit timer expires, two indistinguishable cases (why?):
  - **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 identifier (*nonce*)
- Design alternative: receiver stores a set of nonces that it has previously seen ("*tombstones*")
  - if received packet seen before, drop, but resend ACK to sender
- How many tombstones must receiver store?

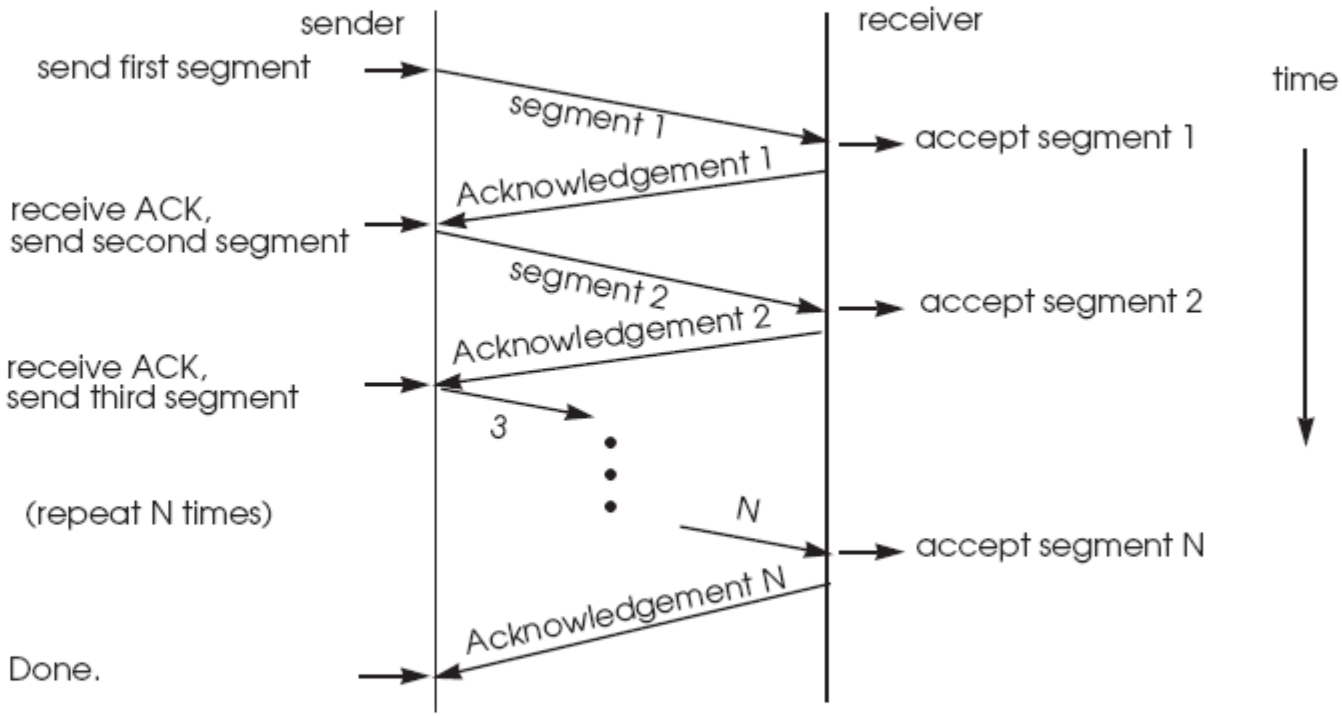
- **Better plan: sequence numbers**
  - Sender marks each packet with **monotonically increasing integer** *sequence number* (non-random nonce)
  - sender includes greatest ACKed sequence number in its packets
  - receiver remembers only greatest received sequence number, drops received packets with smaller ones

# 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)

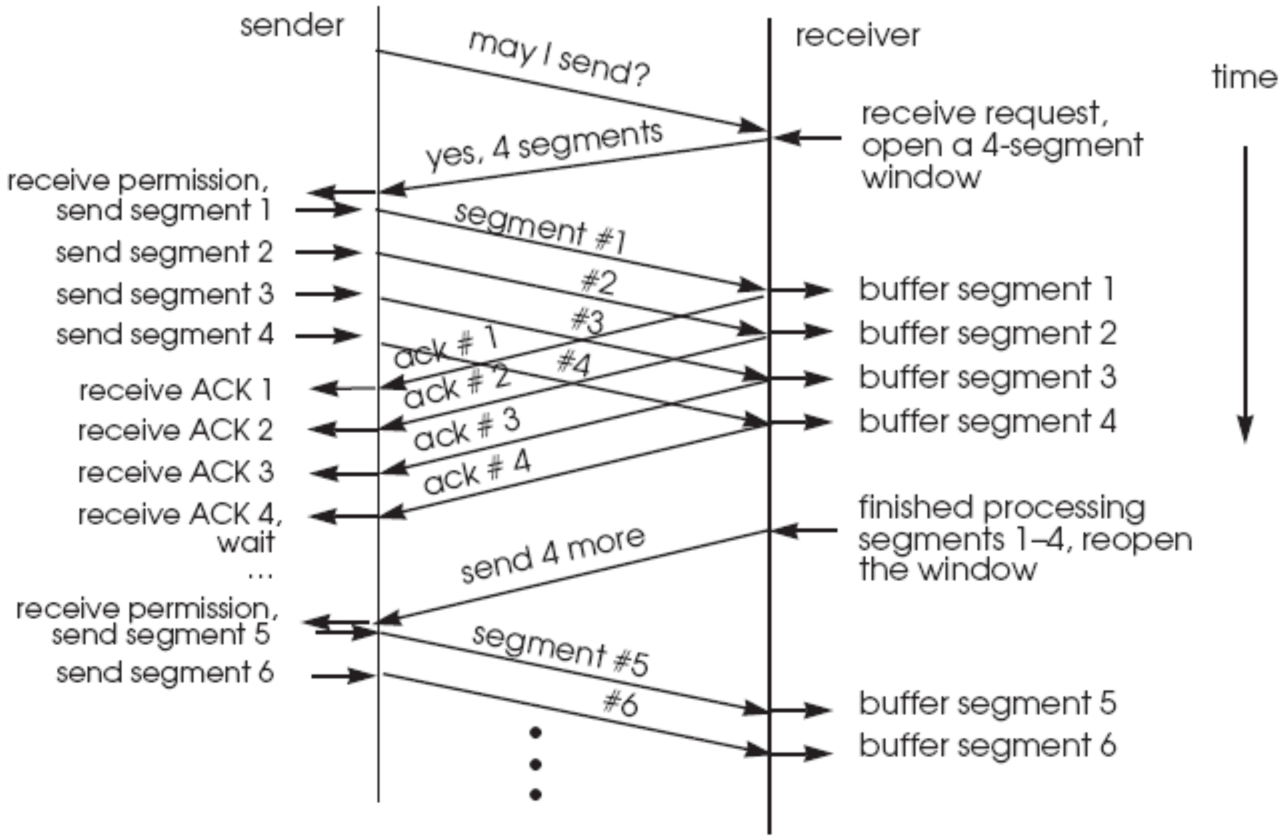
# Flow Control:
# TCP Sliding Window

# Window-Based Flow Control: Motivation




- **Previous scheme (*Stop and Wait*):** 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

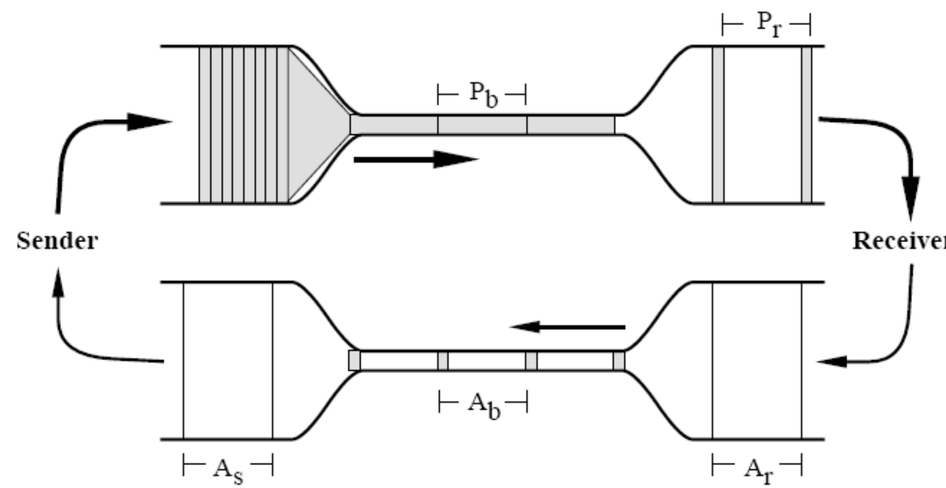



- Pipeline transmissions to "keep pipe full"; overlap ACKs with data
- Sender sends **window** of packets sequentially, without awaiting ACKs
- Sender retains packets until ACKed, tracks which have been ACKed
- Sender sets retransmit timer for each window; when expires, resends all unACKed packets in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

# Choosing Window Size: Bandwidth-Delay Product

- Network bottleneck: point of slowest rate along path between sender and receiver

- To keep pipe full:
  - window size ≥ RTT × bottleneck rate

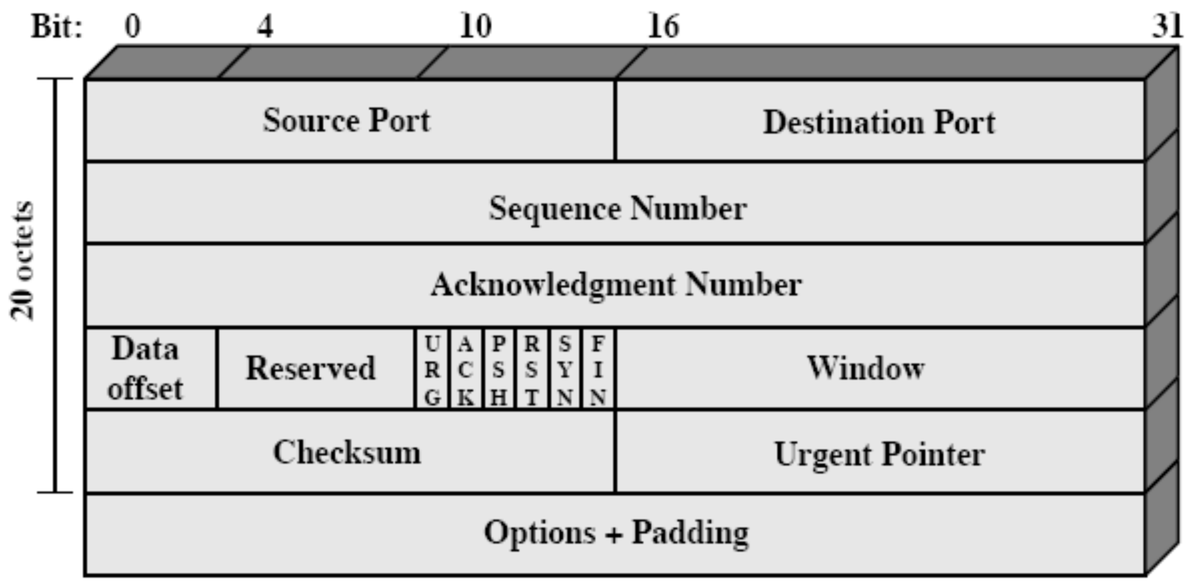



- Window too small: **can't fill pipe**
- Window too large: **unnecessary network load/queuing/loss**

# TCP Support for Reliable Delivery

- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
  - … and for putting the data back in order
- **Recover from lost data:** retransmission
  - Sender retransmits lost or corrupted data
  - Two main ways to detect lost packets

# 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)
- Window: advertisement of number of bytes advertiser willing to accept

# 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

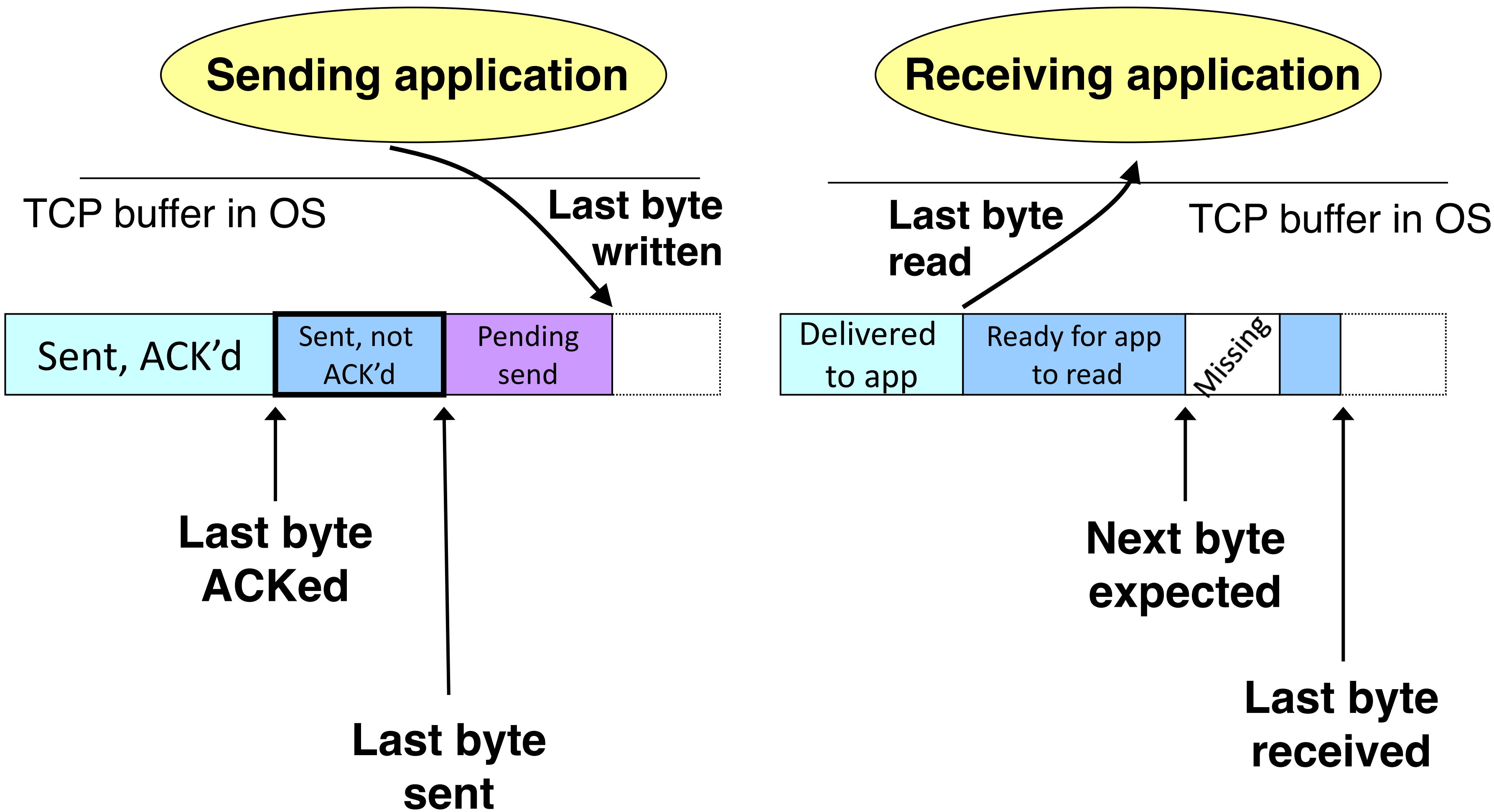
# TCP: Data Transmission (II)

- 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)

- ***Window*** at sender tracks bytes not yet ACK'd
  - Left edge advances as packets acknowledged
  - Right edge advances as updates arrive from receiver

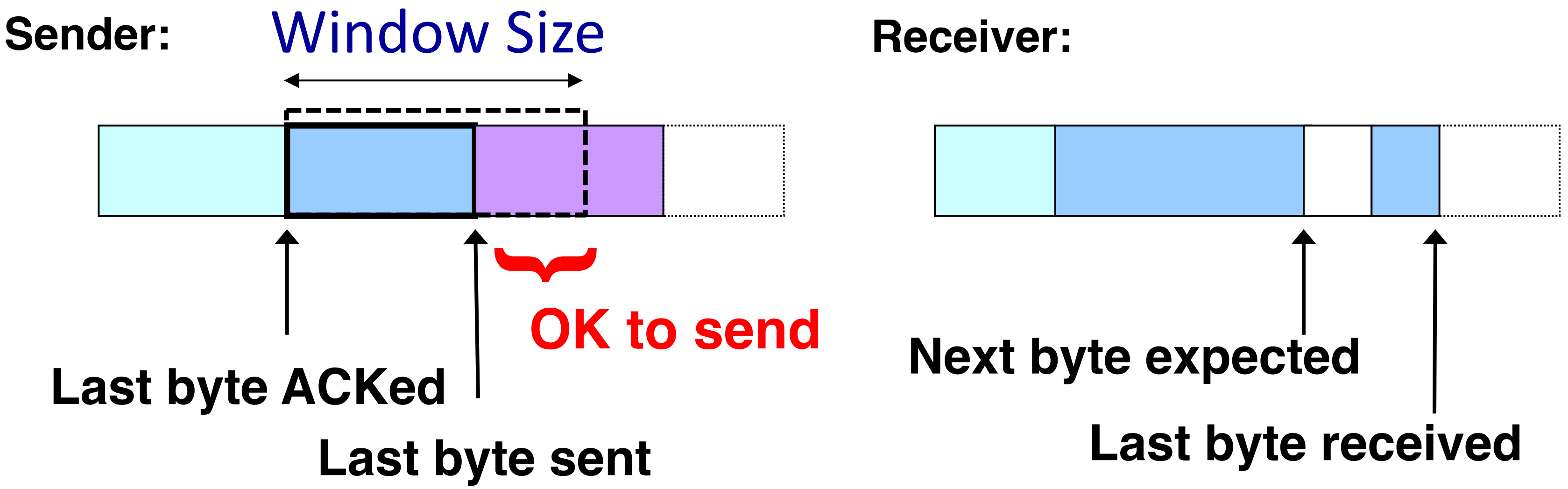
- This is called a ***sliding window***

# TCP's Sliding Window: High-Level View



# TCP's Sliding Window: Window Size

- Sender's ***transmit window size:*** avail. buffer space at sender
- Receiver indicates ***receive window size*** explicitly to sender in **`window`** field in TCP header
  - corresponds to available buffer space at receiver
  - Receiver must be able to store this amount of data
- Sender uses **window** = **min** of send & receive window sizes



# Outline

**Today:**

- Fundamentals, Data transmission
- **Connection establishment**

**Lecture 6:**

- Retransmit timeouts
- RTT estimator
- Slow Start and Self-clocking
- AIMD Congestion control

# Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not a de facto ISN of 0?
- Practical issue: reuse of port numbers
  - Port numbers must (eventually) get used again
  - … and an old packet may still be in flight
  - … and associated with the new connection
- So, TCP must change the ISN over time
  - Set from a 32-bit clock that ticks every 4 microsec
  - … which wraps around once every 4.55 hours!

# TCP Connection Establishment: Motivation

- Goals:
  - Start TCP connection between two hosts
  - Avoid mixing data from old connection in new one
  - Avoid confusing previous connection attempts with current one
  - Prevent (most) third parties from impersonating **(spoofing)** one endpoint

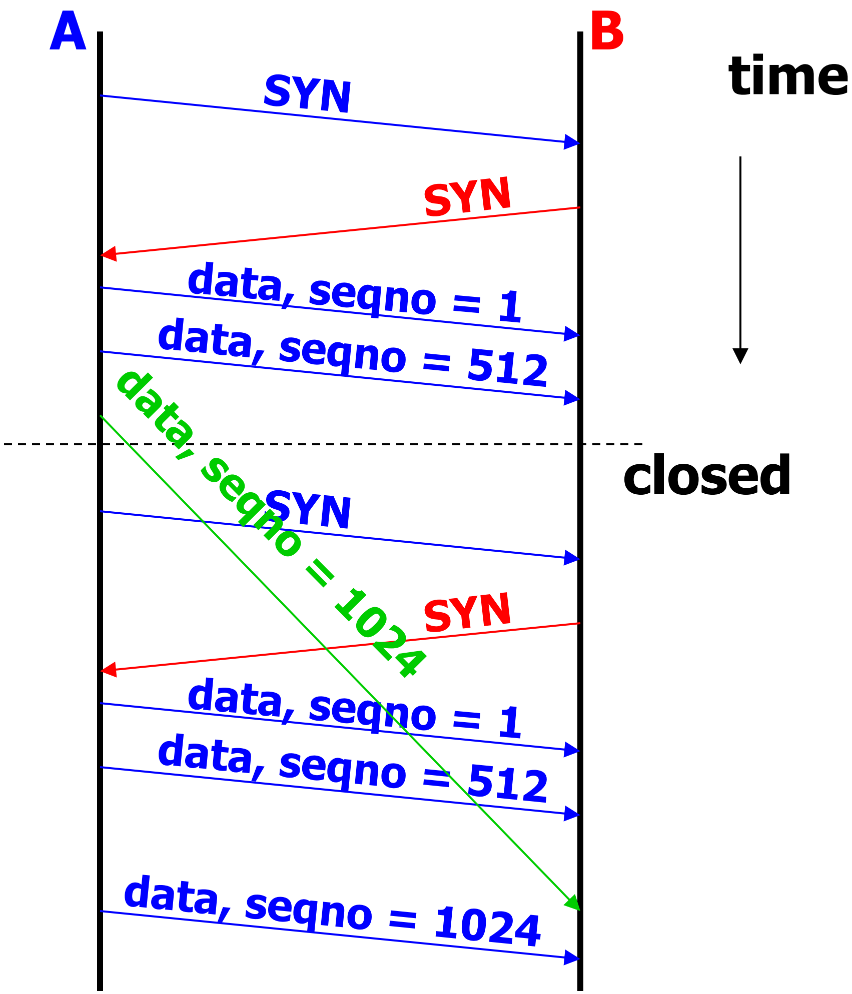
- SYN packets (SYN flag in TCP header set) used to establish connections

- Use retransmission timer to recover from lost SYNs

- What protocol meets above goals?

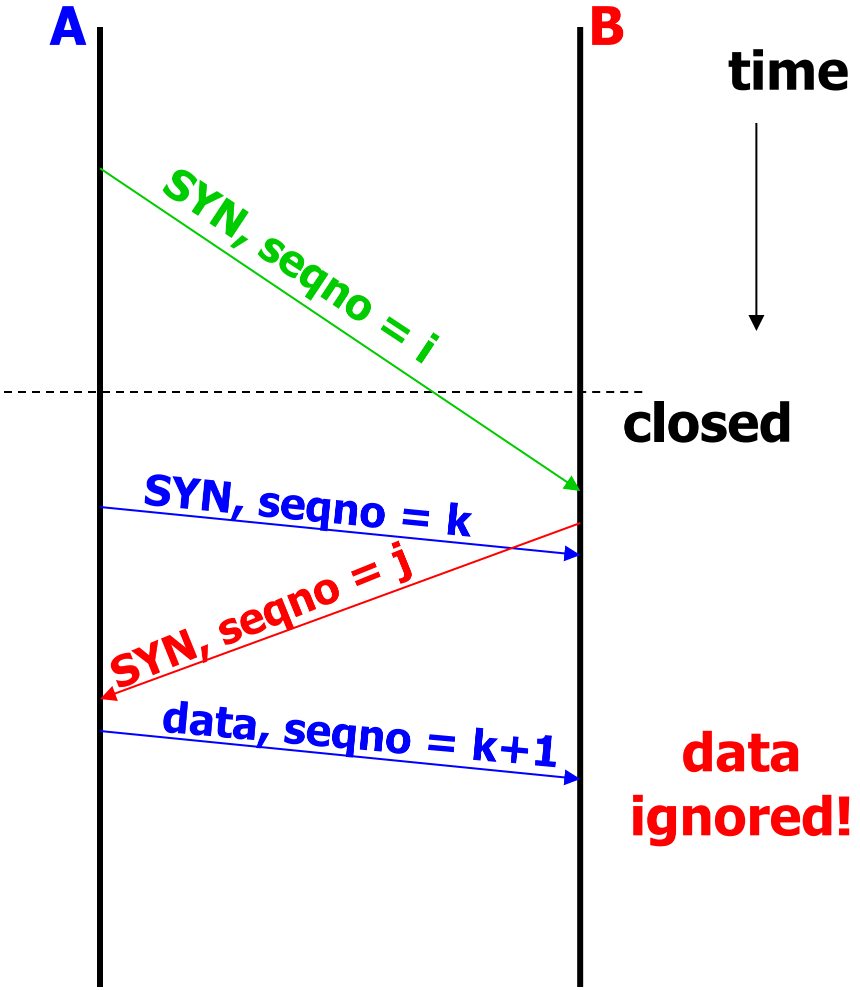
# TCP Connection Establishment: Non-Solution (I)

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



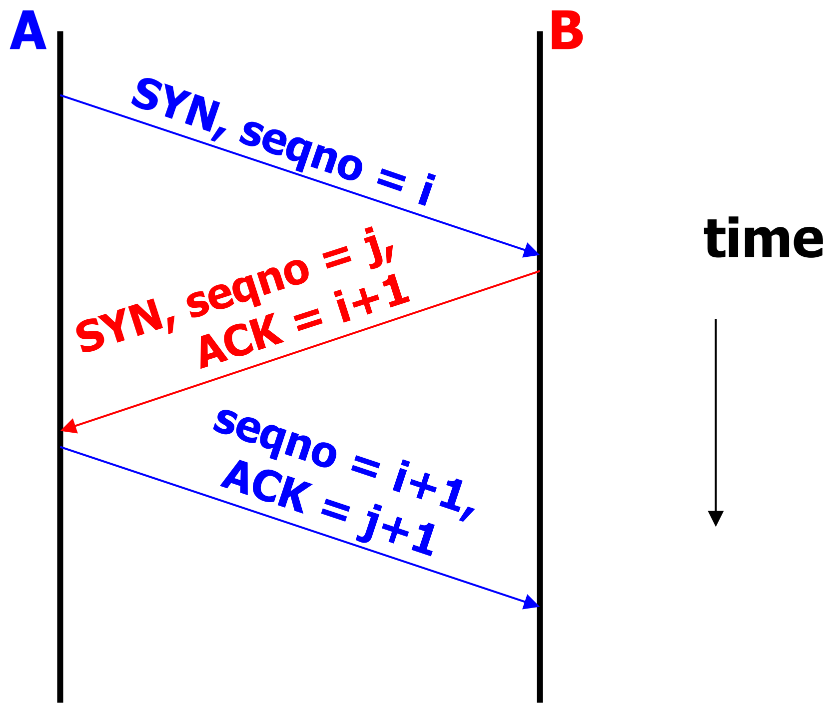
# TCP Connection Establishment: Non-Solution (II)

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



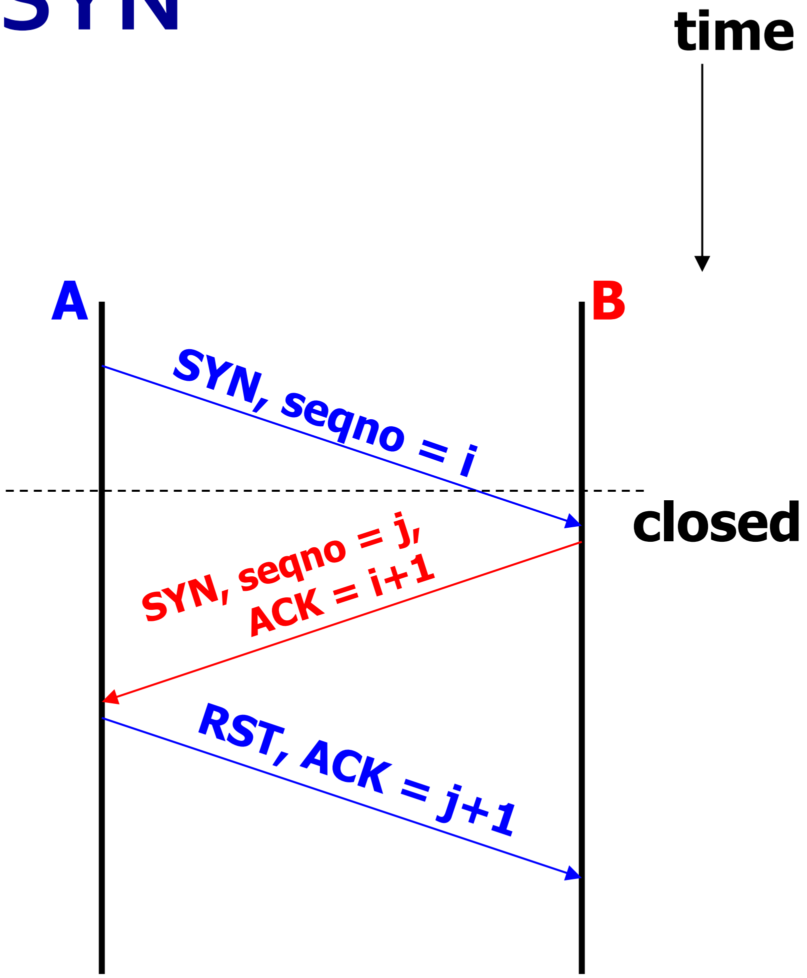
# TCP Connection Establishment: 3-Way Handshake

- Set SYN flag on connection request
- Each side chooses a random *initial sequence number* (ISN)
- Each side **explicitly ACKs the sequence number of the SYN it's responding to**



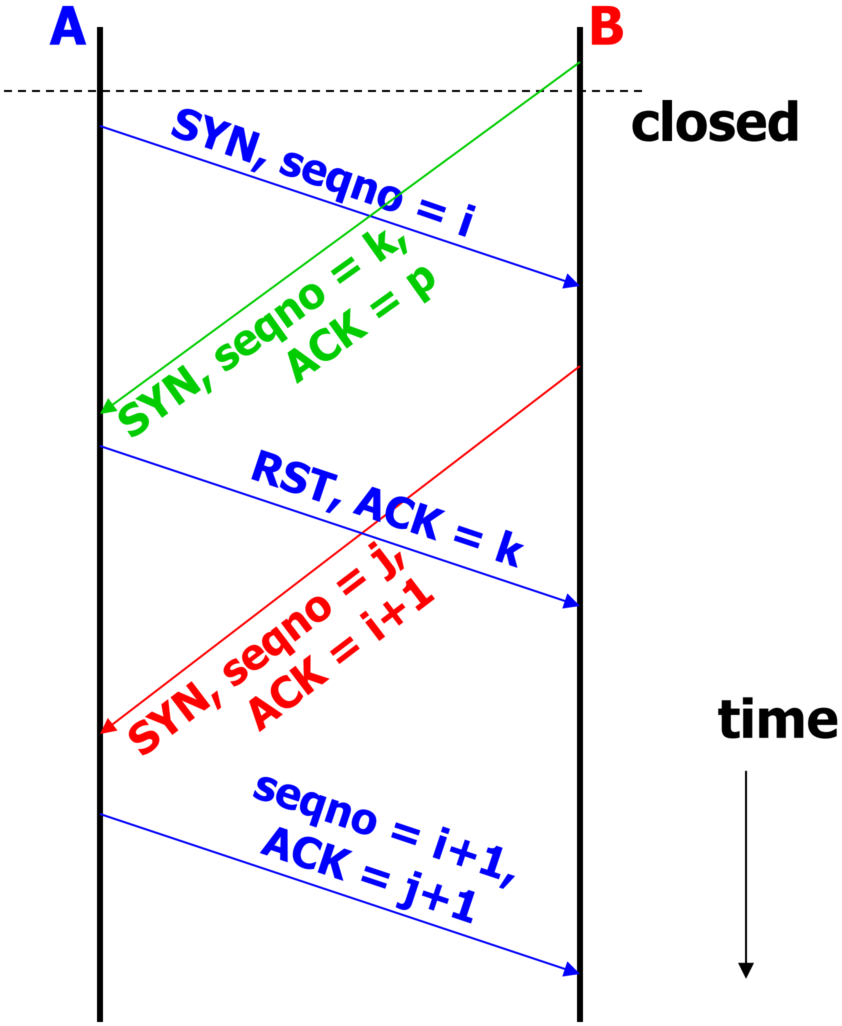
# Robustness of 3-Way Handshake: Delayed SYN

- A's **SYN(*i*)** delayed: arrives after A closes connection
  - B responds: **SYN(j)/ACK(*i*+1)**
- A doesn't recognize *i*+1; responds with **reset, RST** flag set in TCP header
- **A rejects** the connection, **no dropped data later on**



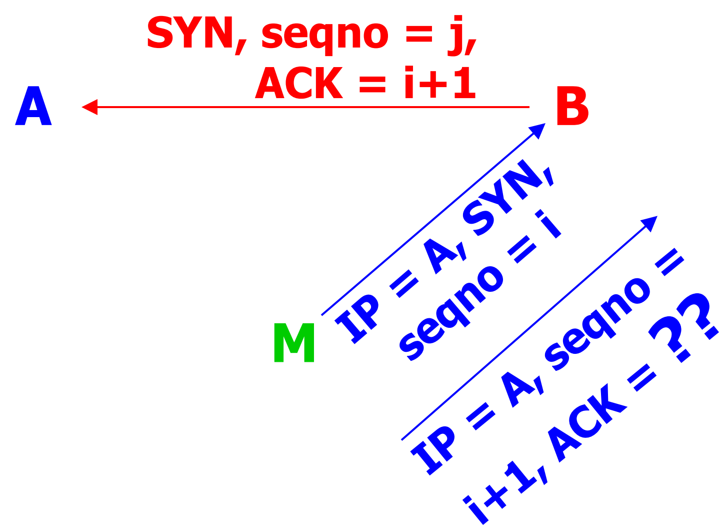
# Robustness of 3-Way Handshake: Delayed SYN/ACK

- 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**
- **Connection from A to B succeeds unimpeded**



# Robustness of 3-Way Handshake: Stopping Source Spoofing

- Suppose host B trusts host A, based on A's IP
  - *e.g.*, B allows any account creation request from A




- Adversary **M** may not control A, but may seek to impersonate A
  - M may not need to receive data from B; only send data (e.g., "create an account l33thaxor")

- **Can M establish a connecti B as A?**

**Unless they are on path between A and B, adversary cannot spoof A to B or vice-versa!**

**Why: random ISNs on SYNs**

# Outline

**Today:**

- Fundamentals, Data transmission
- Connection establishment

**Coming up, in Lecture 6:**

- Retransmit timeouts
- RTT estimator
- Slow Start and Self-clocking
- AIMD Congestion control