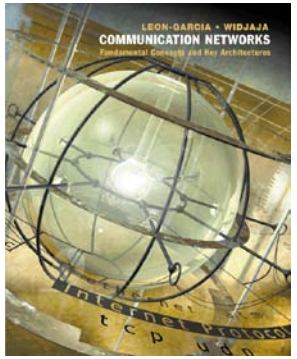
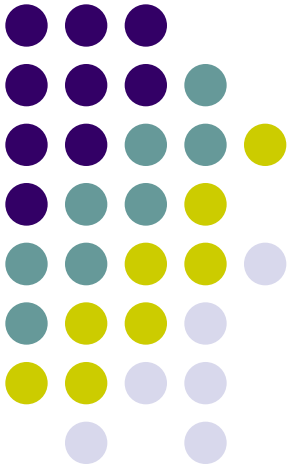
Chapter 8
Communication Networks and Services





Transport Layer Protocols: UDP and TCP

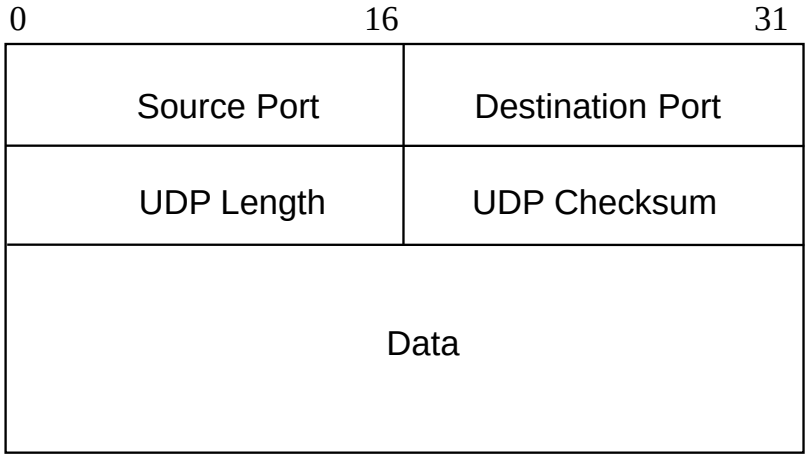
Outline

- UDP Protocol
- TCP – Quick Overview
- TCP Header
- TCP Connection Management
- TCP Congestion Control

UDP – User Datagram Protocol

- Best effort (**unreliable**) datagram service
- Multiplexing enables sharing of IP datagram service
- Simple transmitter & receiver
  - **Connectionless**: no handshaking & no connection state
  - Low header overhead
  - **No flow control, no error control, no congestion control**
  - UDP datagrams can be lost or out-of-order
- Applications
  - multimedia (e.g., VoIP, video, RTP)
  - network services (e.g. DNS, RIP, SNMP)

UDP Datagram



0-255
- Well-known ports

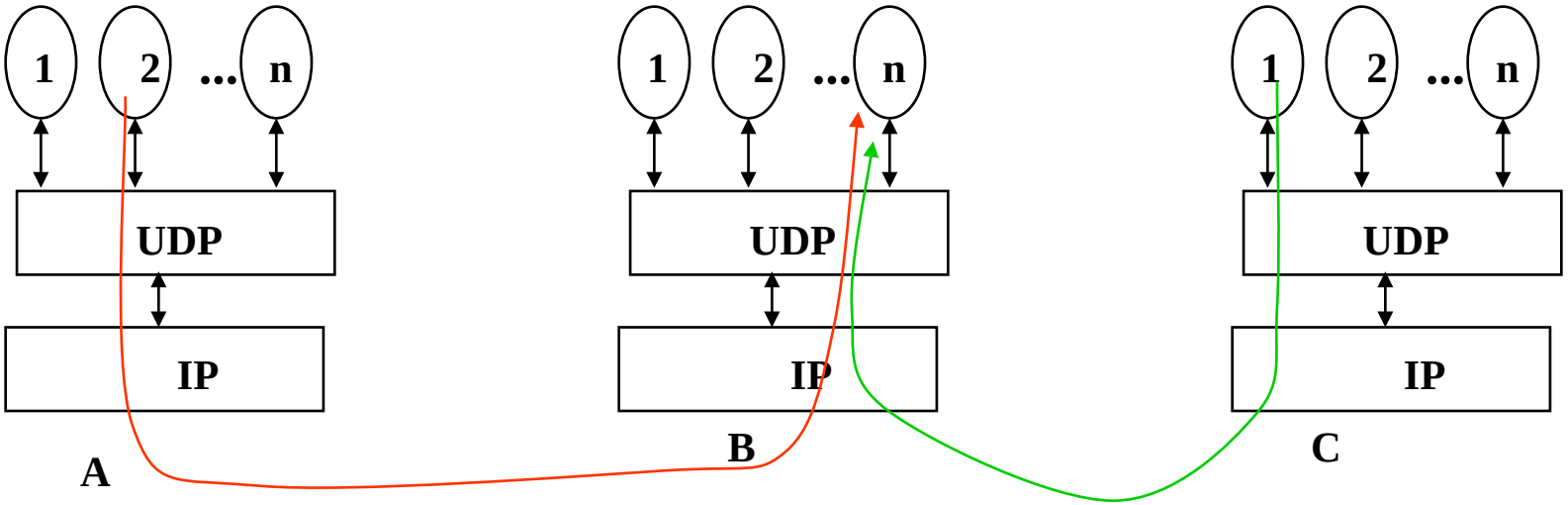
256-1023
- Less well-known ports

1024-65536
- Ephemeral client ports

- Source and destination ports:
  - Identify applications
  - Client ports are ephemeral
  - Server ports are well-known
  - Max number is 65,535
- UDP length
  - Total number of bytes in datagram (including header)
  - 8 bytes ≤ length ≤ 65,535
- UDP Checksum
  - Optionally detects errors in UDP datagram

UDP Multiplexing

- All UDP datagrams arriving to IP address B and destination port number *n* are delivered to the same process
- Source port number is not used in multiplexing



Why Use UDP?

- Finer control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, parameters, sequence #s, etc.
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only eight bytes long

Popular Applications that use UDP

- Multimedia streaming
  - Retransmitting lost/corrupted packets is not worthwhile
  - By the time the packet is retransmitted, it's too late
  - E.g., telephone calls, video conferencing, gaming
  - Note: stored video vs. live video
- Simple query protocols like Domain Name System
  - Overhead of connection establishment is overkill
  - Easier to have application retransmit if needed

Outline

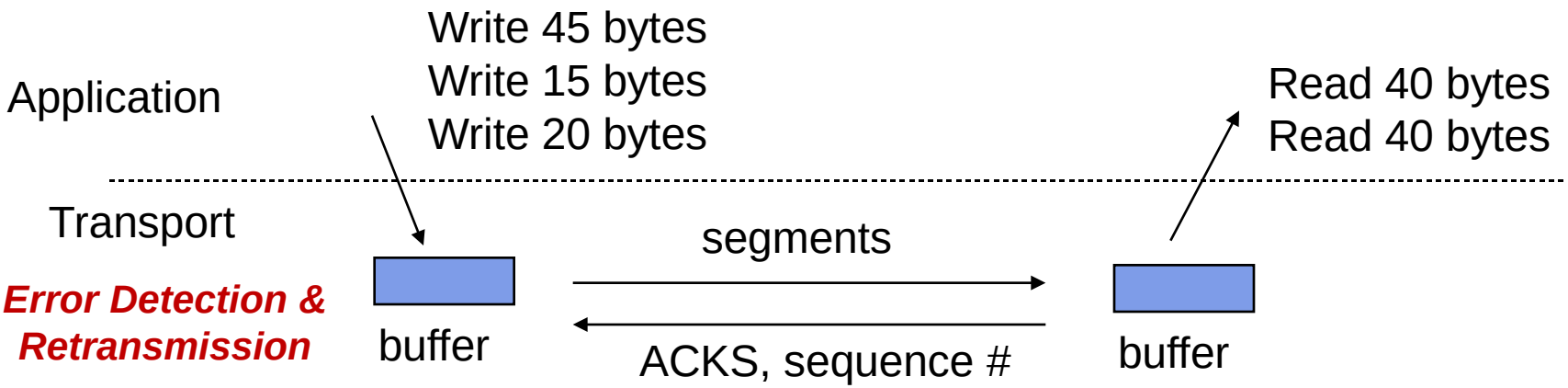
- TCP – Quick Overview
- TCP Header
- TCP Connection Management
- TCP Congestion Control

TCP – Quick Overview

- TCP: Transmission Control Protocol
- **Reliable** byte-stream service
- More complex transmitter & receiver
  - **Connection-oriented** (logical connection): full-duplex unicast connection between client & server processes
    - **Connection setup, connection state, connection release**
  - Higher delay than UDP
  - **Error control, flow control, and congestion control**
  - Higher header overhead
- Most applications use TCP
  - HTTP, SMTP, FTP, TELNET, POP3, …

Reliable Byte-Stream Service

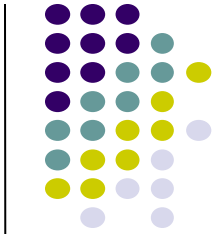
- Stream Data Transfer
  - Transfers a contiguous stream of bytes across the network, with no indication of boundaries
  - TCP groups bytes into **segments**
  - Transmits segments as convenient
  - Example: Application may send a 1000-byte message, TCP may transfer it into two chunks of 500-byte each or three chunks etc.
- Reliability
  - Error control mechanism to deal with IP transfer impairments



Challenges of Reliable Data Transfer

- **Over a reliable channel**
  - All of the data arrives in order, just as it was sent
  - Simple: sender sends data, and receiver receives data
  - Problem is ….
- **Over a channel with bit errors**
  - All of the data arrives in order, but some bits corrupted
  - Receiver detects errors and says “please repeat that”
  - Sender retransmits the data that were corrupted
- **Over a lossy channel with bit errors**
  - Some data are missing, and some bits are corrupted
  - Receiver detects errors but cannot always detect loss
  - Sender must wait for acknowledgment (“ACK” or “OK”)
  - … and retransmit data after some time if no ACK arrives
- **What about out of order packets?**

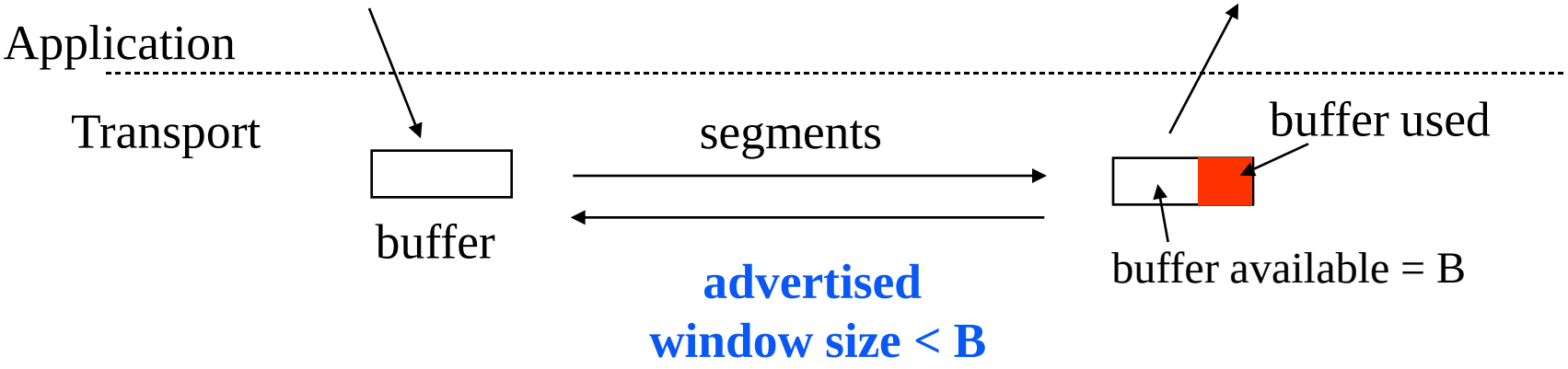
TCP Support for Reliable Delivery



- **Checksum**
  - Used to detect corrupted data at the receiver
  - … leading the receiver to drop the packet
- **Sequence numbers**
  - Used to detect missing data
  - ... and for putting the data back in order
- **Retransmission**
  - Sender retransmits lost or corrupted data
  - Timeout based on estimated round-trip time
  - Fast retransmit algorithm for rapid retransmission

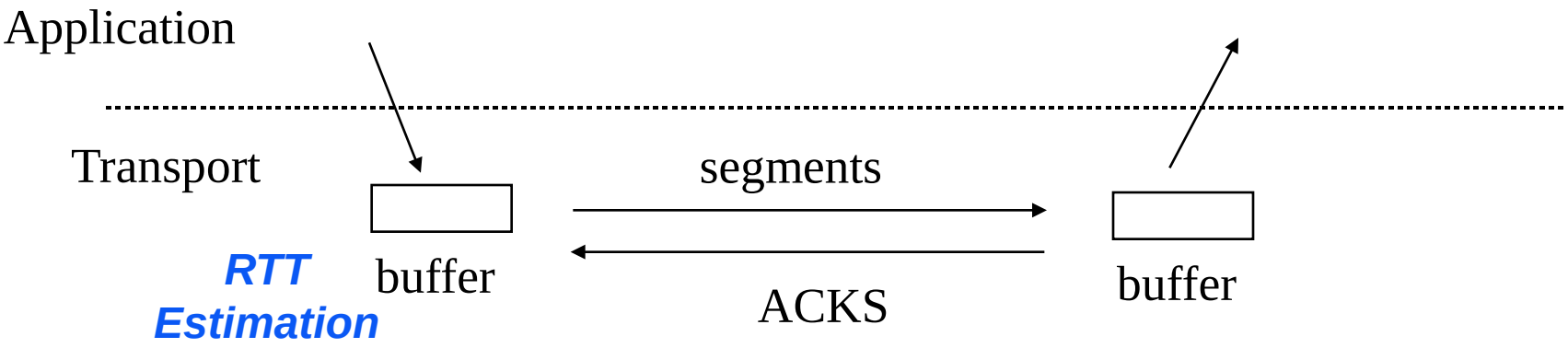
Flow Control between Hosts

- **Buffer limitations & speed mismatch** can result in loss of data that arrives at destination
- Solution?
  - **Receiver controls rate** at which sender transmits to prevent buffer overflow



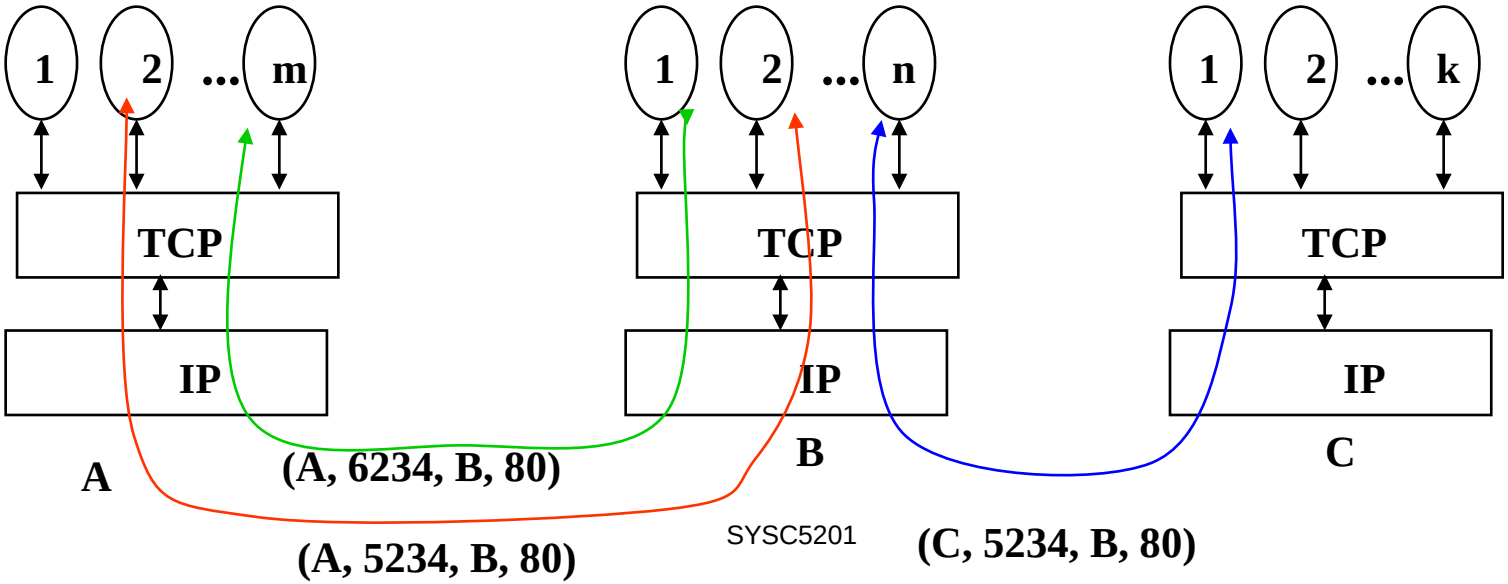
Congestion Control over the Network

- Available bandwidth to destination varies with activity of other users
- How to cope with it?
  - Transmitter dynamically adjusts transmission rate according to **network congestion as indicated by RTT (round trip time) & ACKs**
  - Elastic utilization of network bandwidth



TCP Multiplexing

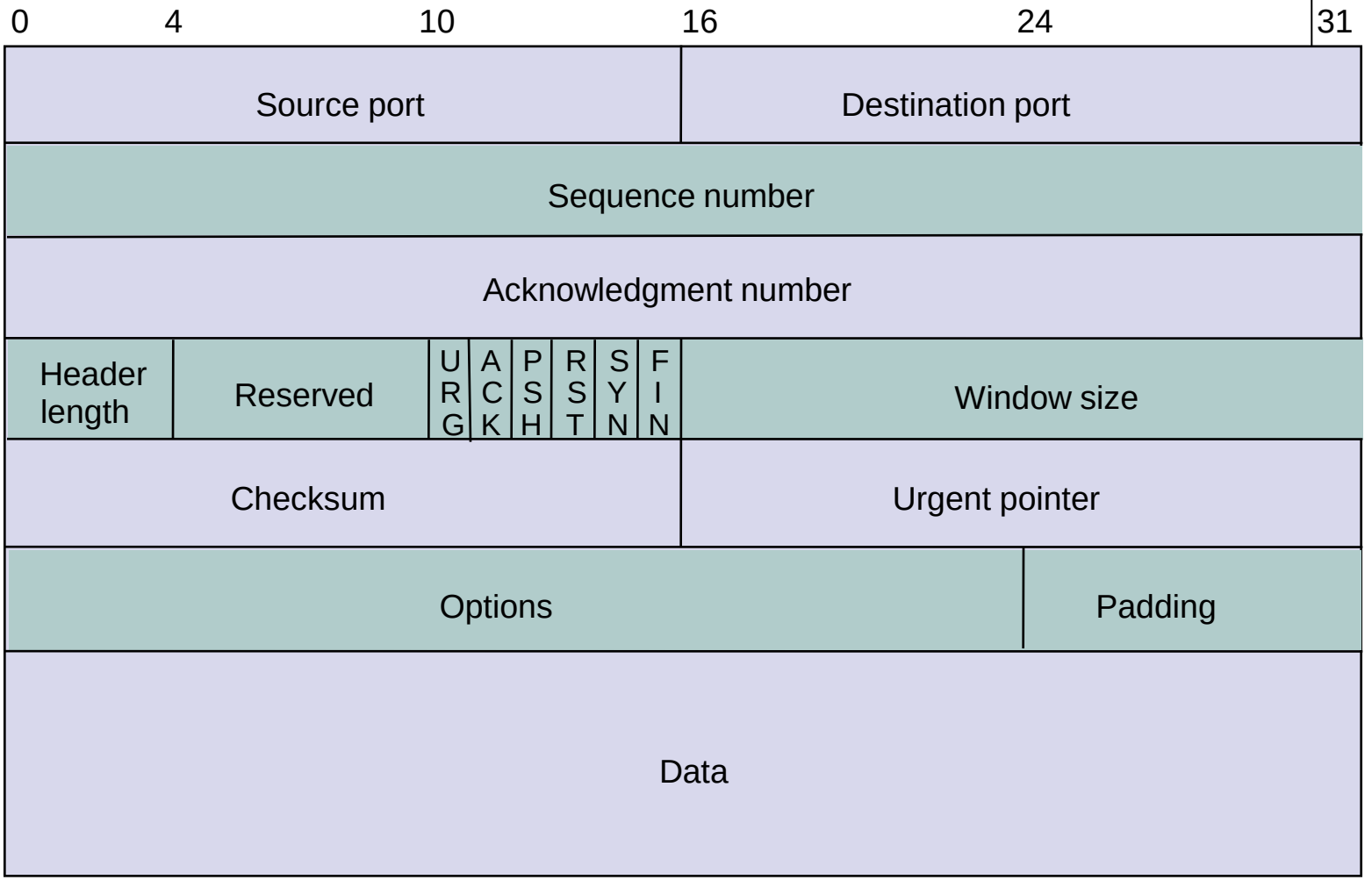
- A *TCP connection* is specified by a *4-tuple*
  - (source IP address, source port, destination IP address, destination port)
- TCP allows multiplexing of multiple connections between end systems to support multiple applications simultaneously
- Arriving segment directed according to connection 4-tuple



Outline

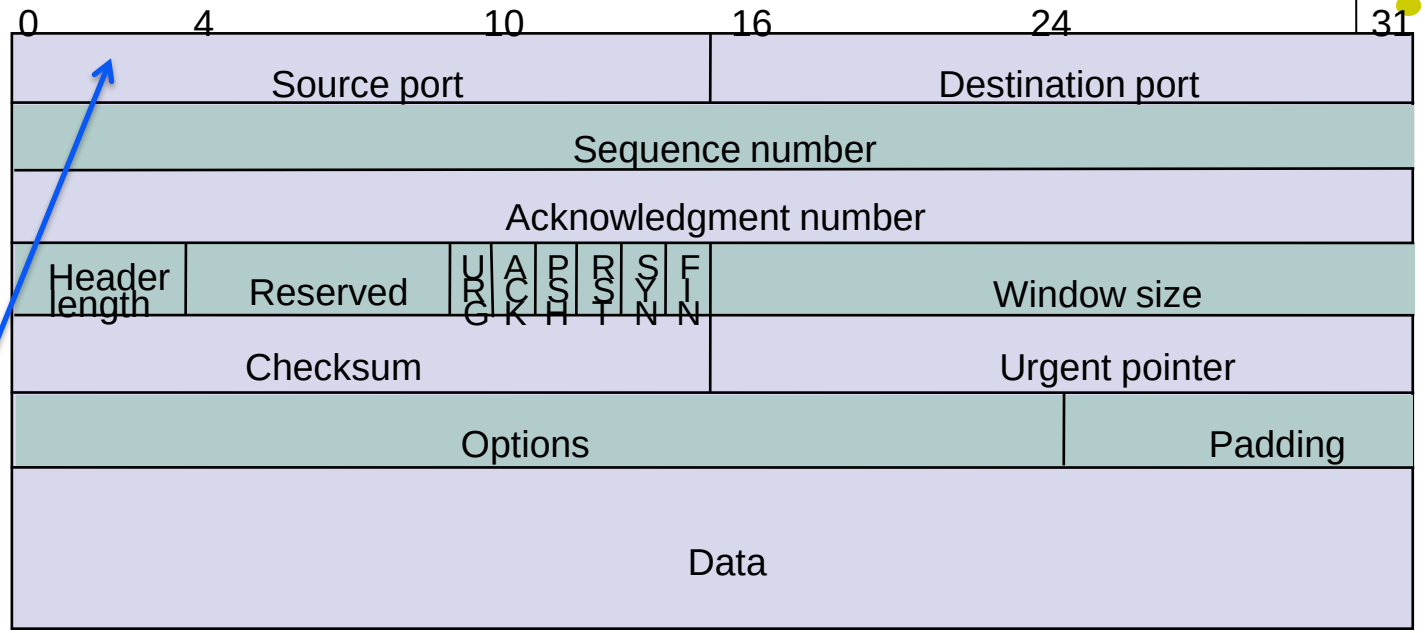
- TCP – Quick Overview
- TCP Header
- TCP Connection Management
- TCP Congestion Control

TCP Segment Format



- Each TCP segment has header of **20** or more bytes + 0 or more bytes of data

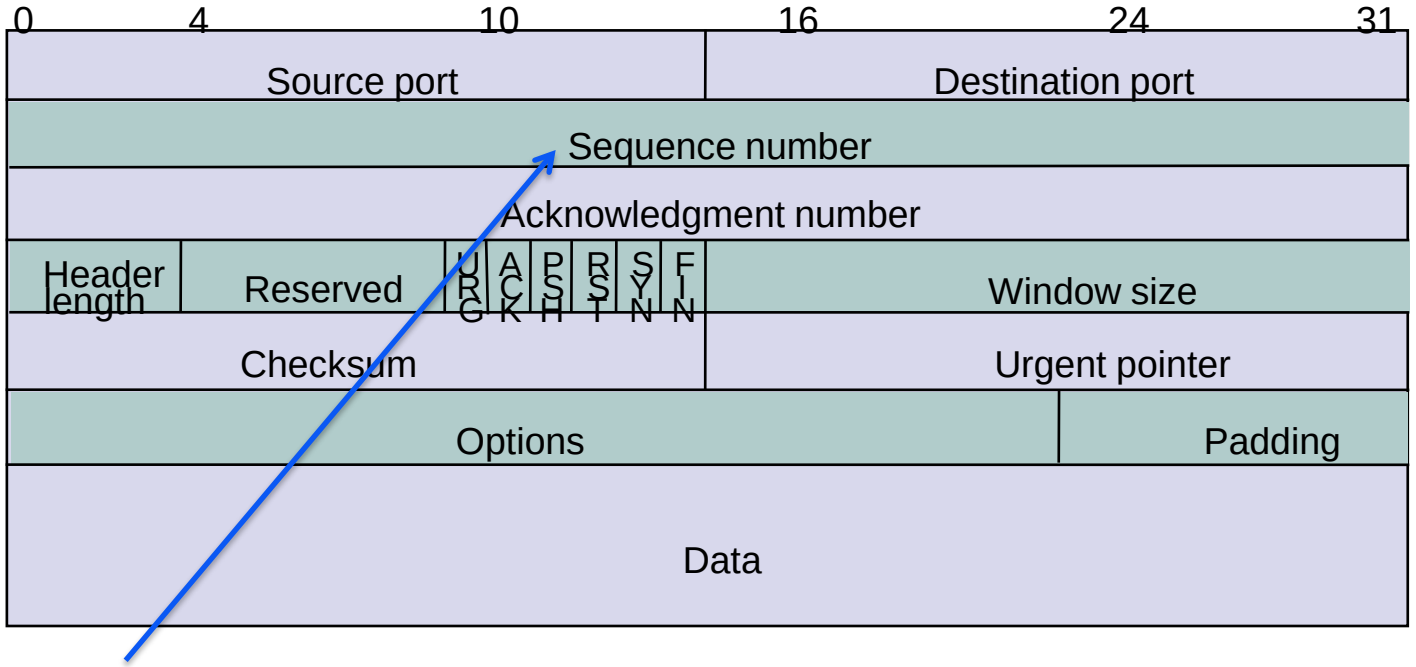
TCP Header



Port Numbers

- A socket identifies a connection endpoints or applications (processes)
  - IP address + port
- A connection specified by a *socket pair*
- Well-known ports: FTP 20, DNS 53, HTTP 80,

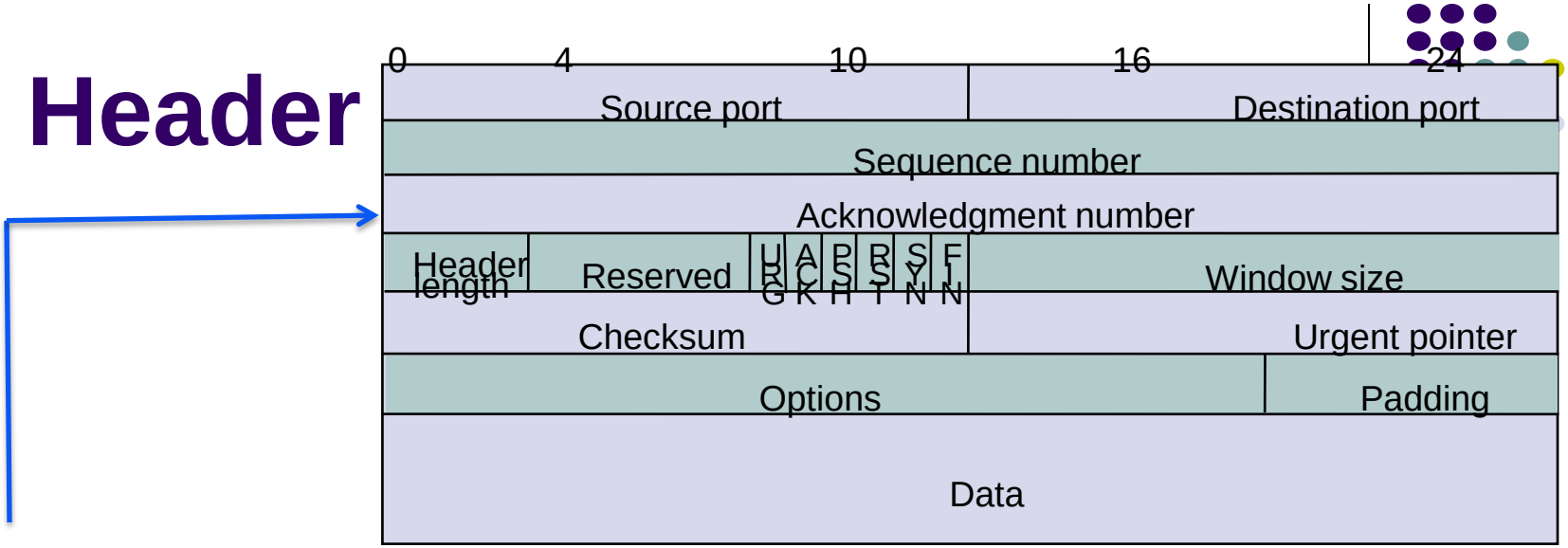
TCP Header



Sequence Number (SN): byte count, 32 bits ($0 \leq SN \leq 2^{32}-1$)

- Position of first data byte in segment (offset for the byte stream).
  - If SN=100 and there are 5 data bytes in the segment, then the next segment will have a SN=105.
- Initial sequence number selected during connection setup
  - If SYN=1(during connection establishment) the SN indicates the initial SN (ISN) of the senders byte stream. The sequence number for the first data byte in this stream will be ISN + 1.

TCP Header



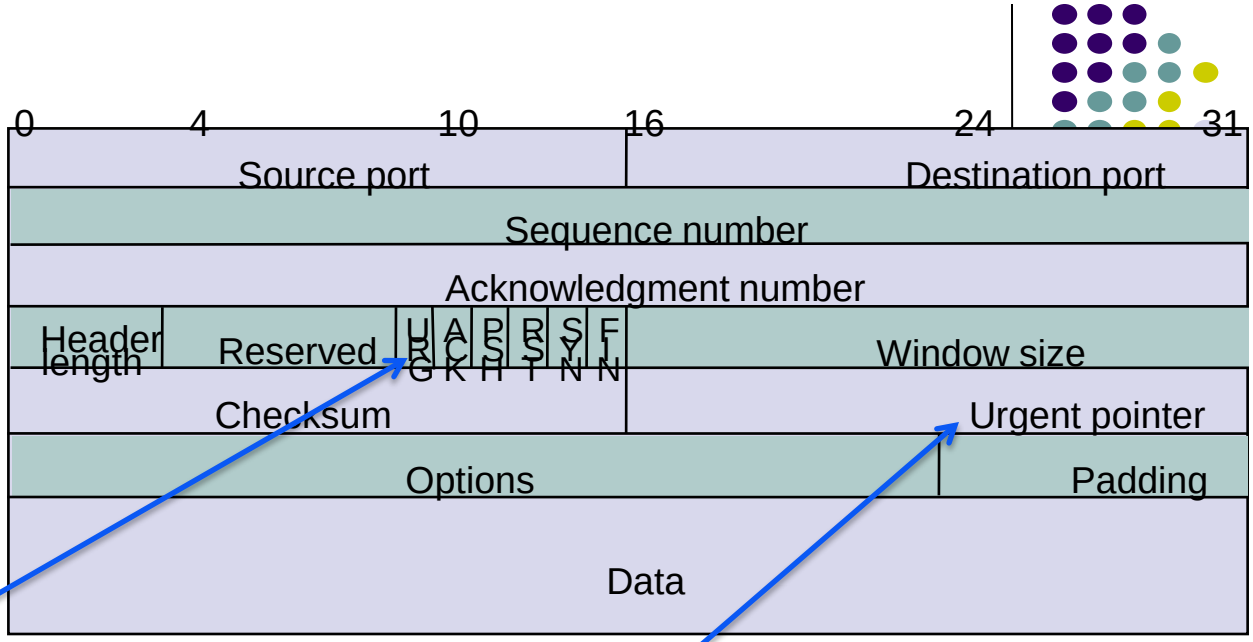
Acknowledgement Number (similar to ARQ)

- SN of **next byte** expected by receiver
- Acknowledges that all prior bytes in stream have been received correctly
- Valid if ACK flag is set

Header length (4 bits)

- Length of header in multiples of 32-bit words (4 bytes)
- Minimum 20 bytes, maximum 60 bytes

TCP Header



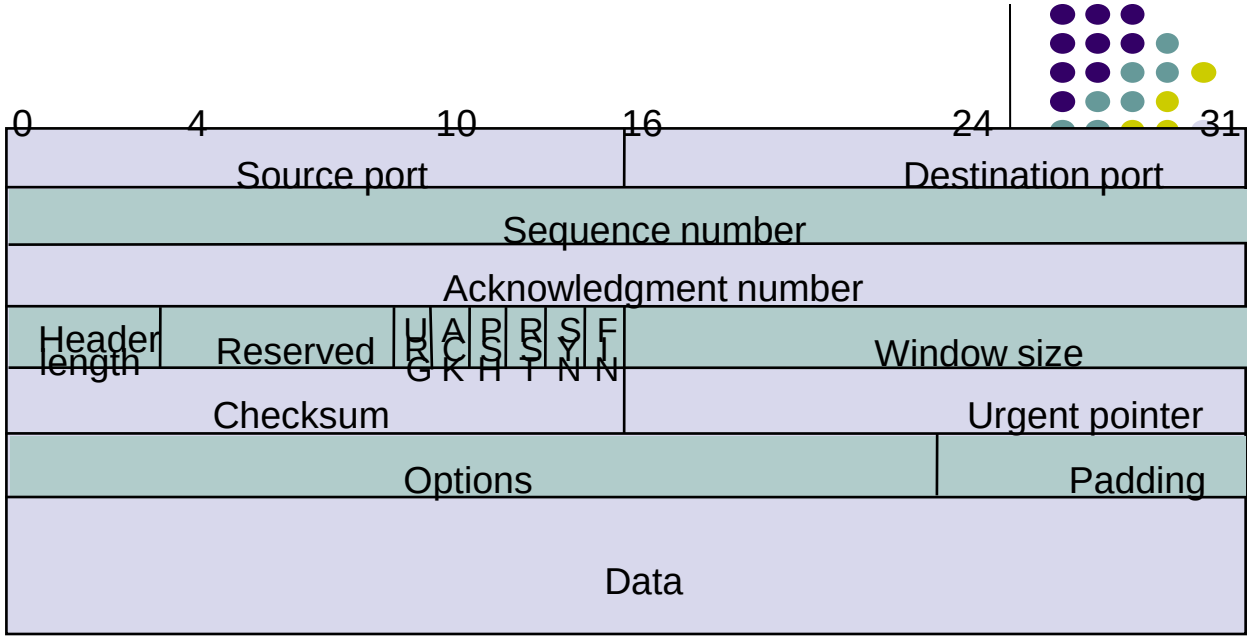
Reserved

- 6 bits
- Future use

Control (6 bits)

- URG: urgent pointer flag (data needs immediately delivery)
  - Urgent message end = SN + **urgent pointer**
- **ACK**: ACK number is valid
- PSH: override TCP buffering, pass to the application immediately
- **RST**: **reset** connection
  - Connection is aborted (e.g., abnormal op) and application layer notified
- **SYN:** request a connection
- **FIN:** sender finishes sending, but still needs to get a FIN from receiver

TCP Header



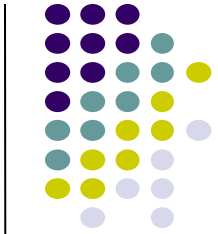
Window Size (16 bits to advertise window size)

- Used for **flow and congestion control**
- Sender will accept bytes with SN from ACK to **ACK + window**
- Maximum window size is 65535 bytes

TCP Checksum

- Internet checksum method
- TCP pseudoheader + TCP segment
  - Pseudoheader: simplified header created by src and dest., not transmitted.

TCP Header



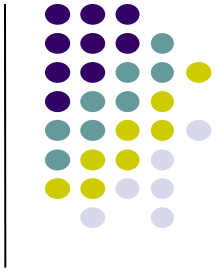
Options

- Variable length
- NOP (No Operation) option is used to pad TCP header to multiple of 32 bits
- Time stamp is used for:
  - Round trip measurements
  - Distinguish wrap around SNs for high speed routers

Options

- Maximum Segment Size (MSS) option specifies largest segment a receiver wants to receive
  - Specified during connection setup.
- Window Scale option increases TCP window from 16 to 32 bits

Example of Control Flags



- Two traces
  - CAIDA (C_04): “dirty” traffic due to **port scanning or DoS** attacks
  - NLANR (N_12): clean traffic
- Characteristics for TCP control packets

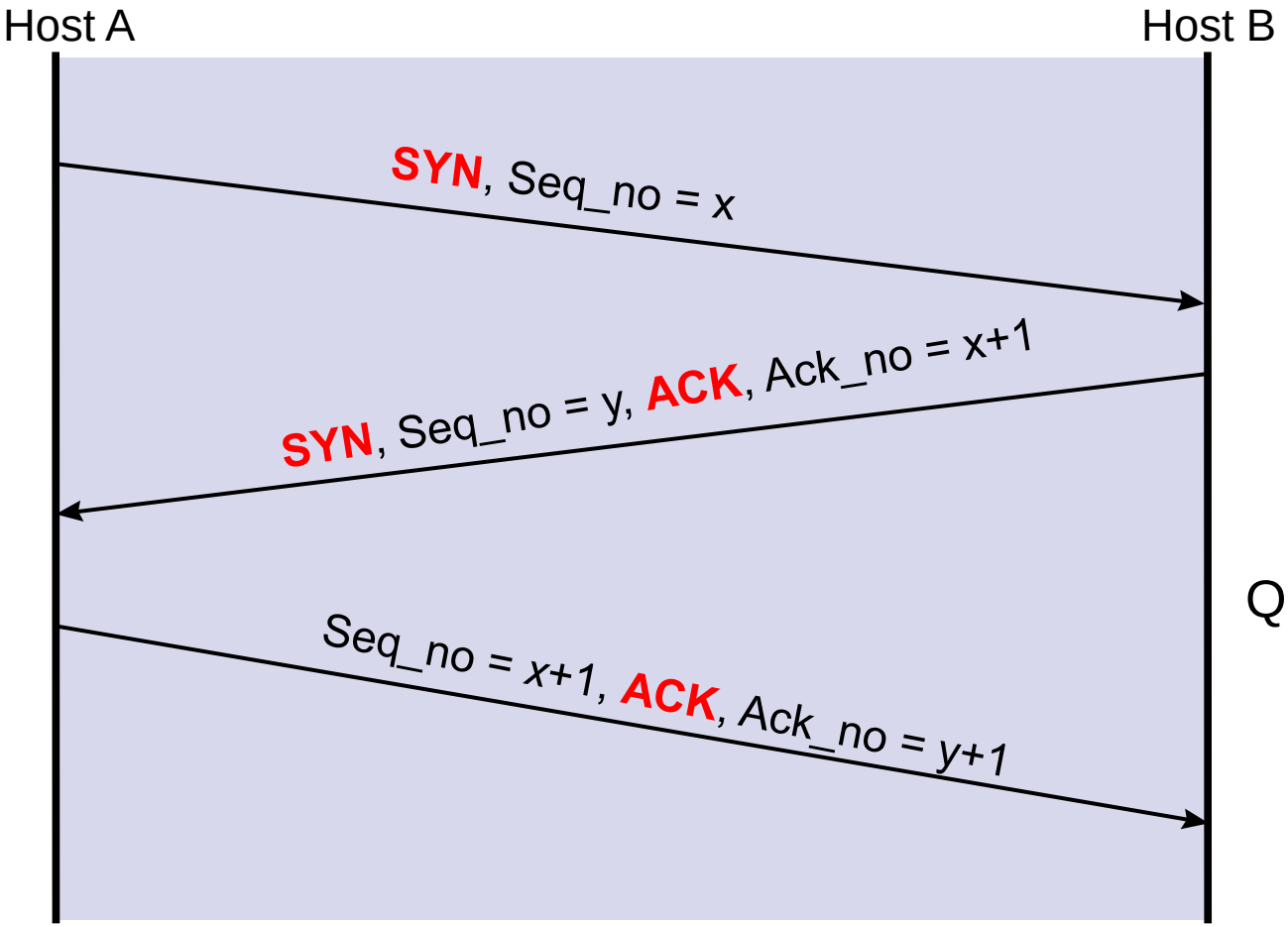
	N_12	As a % of total	C_04	As a % of total
Total	196.9M		202.5M	
SYN	732,075	**0.37%**	15,608,680	**7.71%**
FIN	586,000	**0.30%**	6,084,826	**3.00%**
RST	52,628	**0.03%**	3,914,433	**1.93%**

Outline

- TCP – Quick Overview
- TCP Header
- TCP Connection Management
- TCP Congestion Control

TCP Connection Establishment

- "**Three-way Handshake**"
- ISN's protect against segments from prior connections

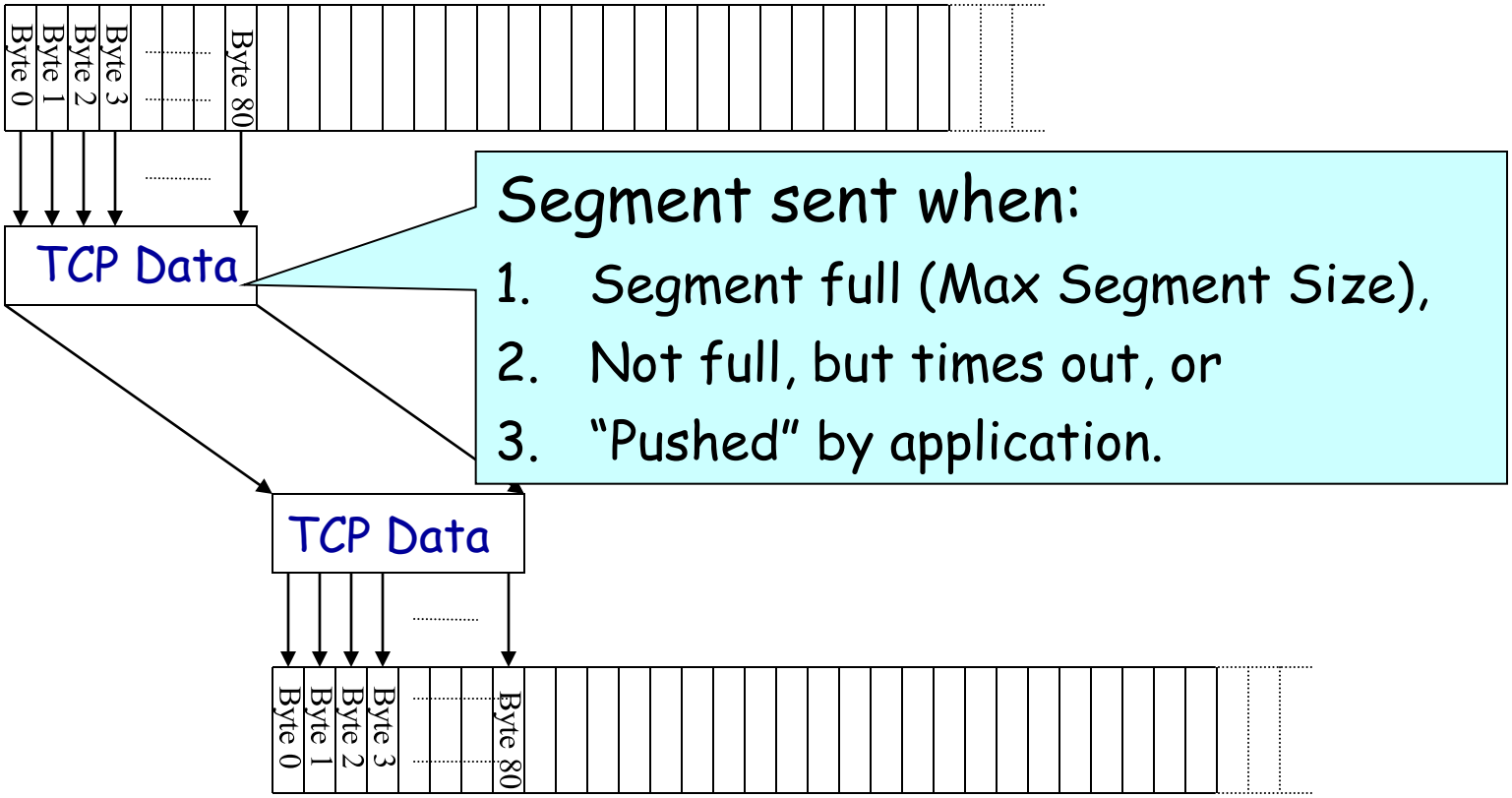


Q: What is the ACK no from receiver? From the sender? Seq. no?

Emulated Using TCP "Segments"

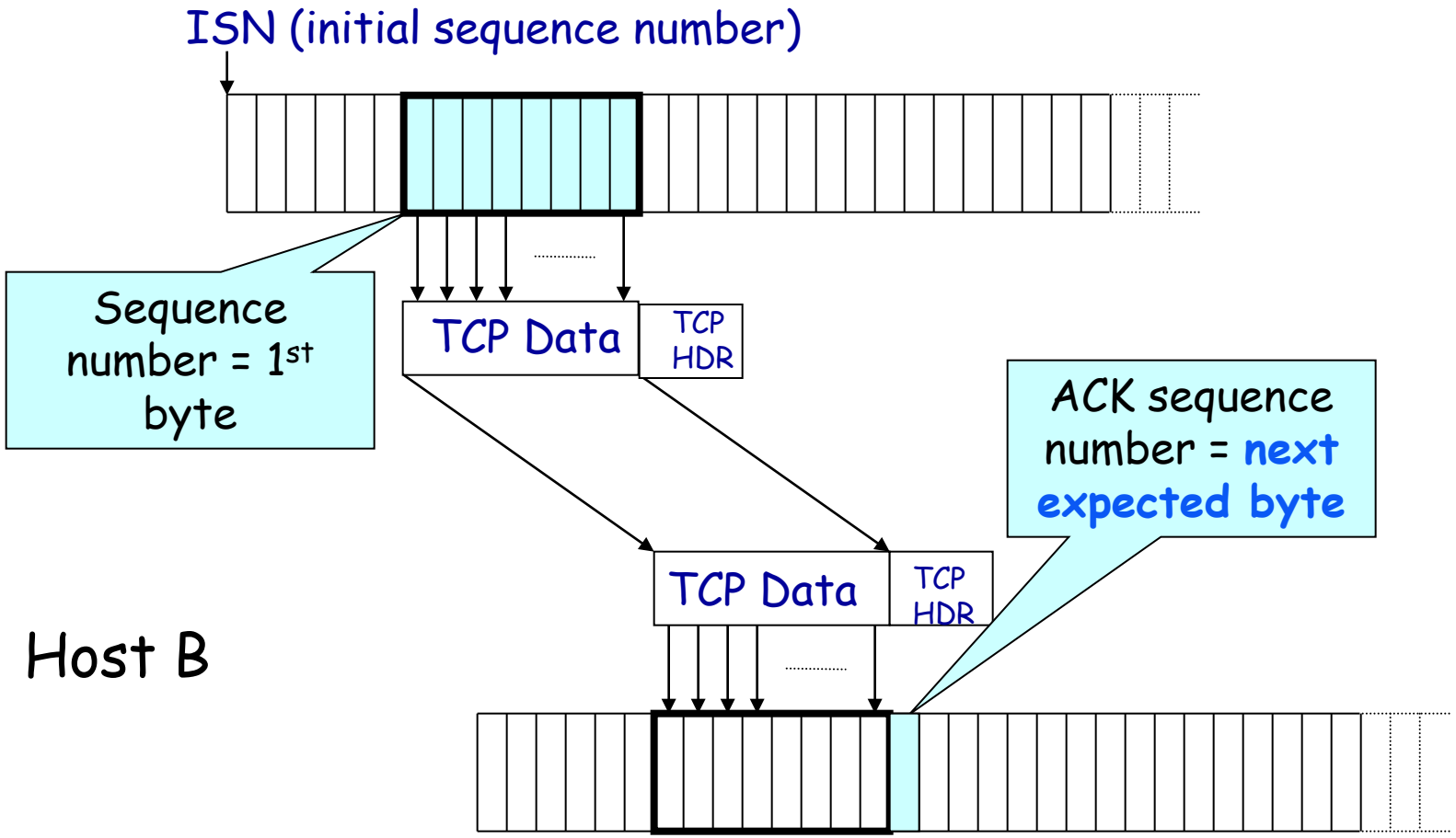
TCP "stream of bytes"

Host A



Host B

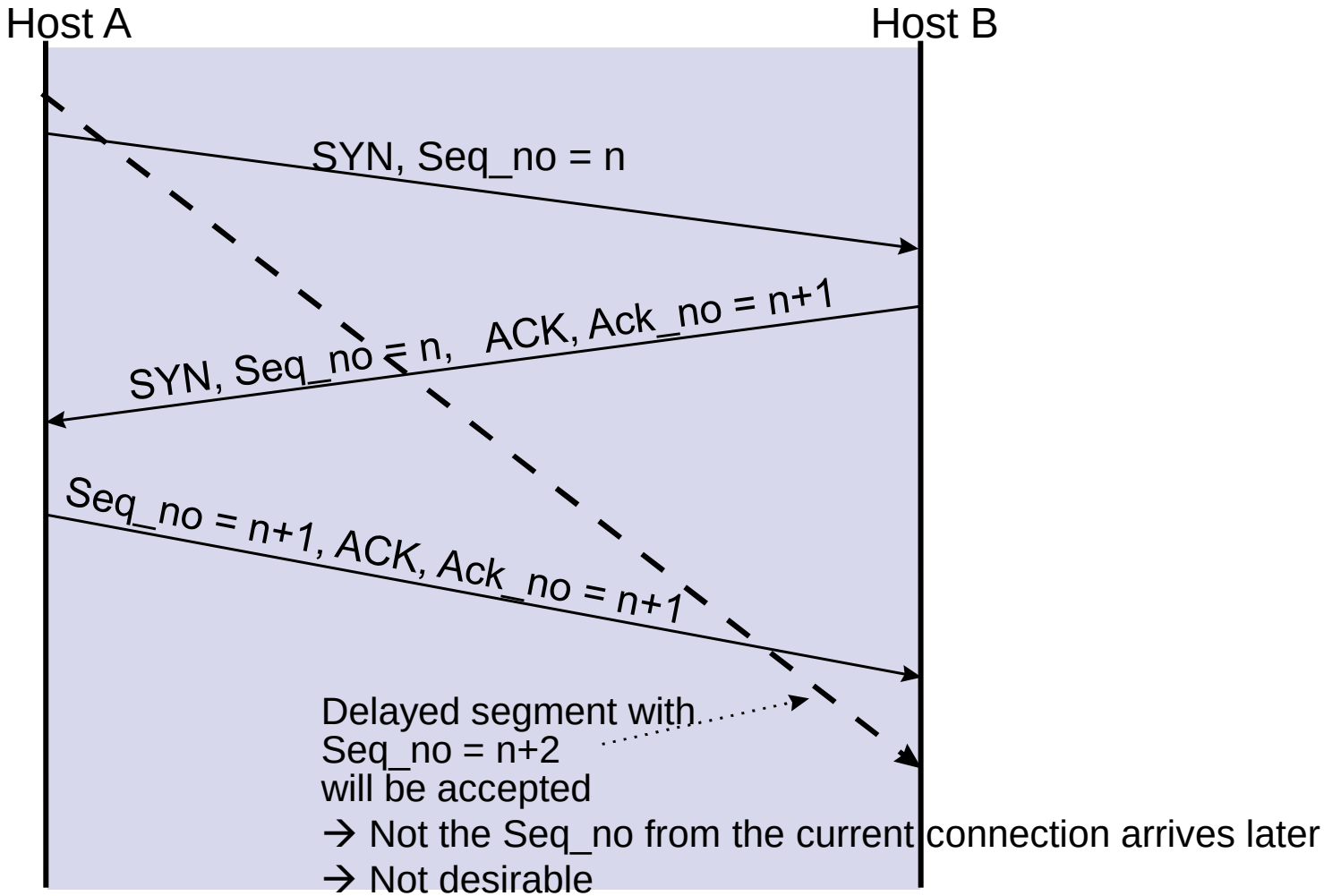
Sequence Numbers



Initial Sequence Number (ISN)

- The sequence number for the **very first byte**
  - Why ISN? Why not 0 for ISN?
- Select ISNs to protect against segments from prior connections (that may circulate in the network and arrive at a much later time)
- Select ISN to **avoid overlap** with sequence numbers of prior connections
- Use local clock to select ISN (ISN is increased by 1 every four microseconds)
  - Using a 32-bit clock, it wraps around every 4.55 hours.
- High bandwidth connections pose a problem
  - Use timestamps to distinguish wrap around SNs

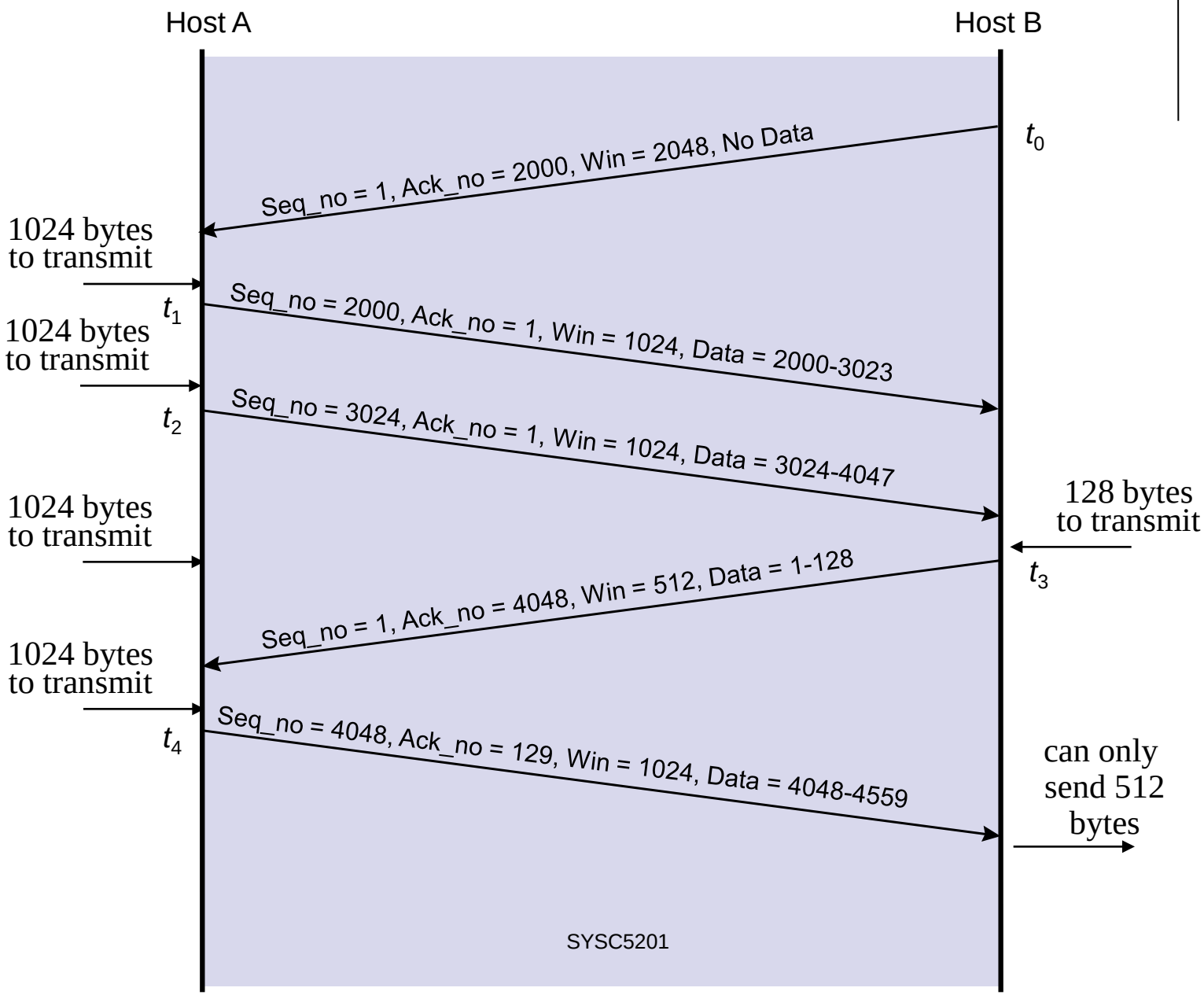
If host always uses the same ISN (p.609)



Maximum Segment Size

- Maximum Segment Size (MSS)
  - largest block of data that TCP sends to other end
- Each end can announce its MSS during connection establishment
- Default is 576 bytes including 20 bytes for IP header and 20 bytes for TCP header
- Ethernet implies MSS of 1460 bytes

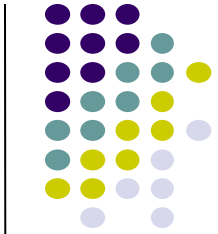
TCP Window Flow Control



Nagle's Algorithm

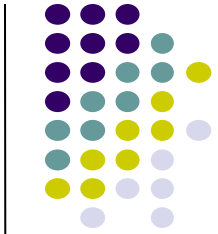
- Situation: user types 1 character at a time
- Example: remote login echoes the input
  - Transmitter sends TCP segment per character (41Bytes)
  - Receiver sends ACK (40Bytes)
  - Receiver echoes received character (41Bytes)
  - Transmitter ACKs echo (40 Bytes)
  - 162 bytes transmitted to transfer 1 character!
- Solution:
  - TCP sends data & waits for ACK
  - New characters buffered (instead of 1 char at a time)
  - Send new characters when ACK arrives
  - Algorithm adjusts to RTT
    - Short RTT send frequently at low efficiency
    - Long RTT send less frequently at greater efficiency

Silly Window Syndrome



- Situation:
  - Transmitter sends large amount of data
  - Receiver buffer depleted slowly, so buffer fills
  - Every time a few bytes read from buffer, a new advertisement to transmitter is generated
  - Sender immediately sends data & fills buffer
  - Many small, inefficient segments are transmitted
- Solution:
  - Receiver does not advertise window until window is at least ½ of receiver buffer or maximum segment size
  - Transmitter refrains from sending small segments

Sequence Number Wraparound



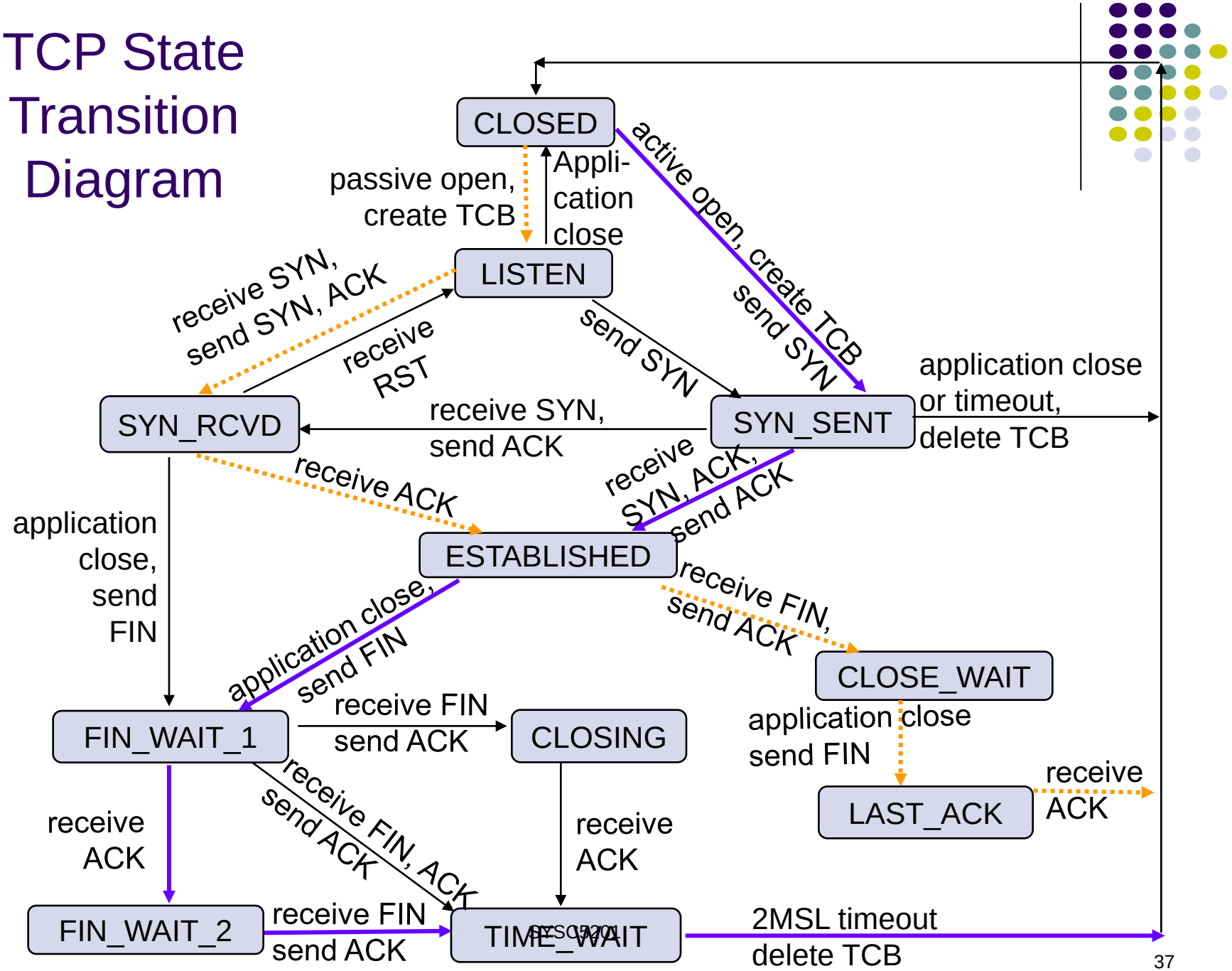
- $2^{32} = 4.29 \times 10^9$ bytes = 34.3×10^9 bits
  - At 1 Gbps, sequence number wraparound in 34.3 seconds.
- Timestamp option: Insert 32 bit timestamp in header of each segment
  - Timestamp + sequence no → 64-bit seq. no
  - Timestamp clock must:
    - tick forward at least once every 2^{31} bits
    - Not complete cycle in less than one MSL
    - Example: clock tick every 1 ms @ 8 Tbps wraps around in 25 days

Delay-BW Product & Advertised Window Size

- Suppose RTT=100 ms, R=2.4 Gbps
  - # bits in pipe → 30 Mbytes
- If single TCP process occupies pipe, then required advertised window size is
  - RTT x Bit rate = 30 Mbytes
  - Normal maximum window size is 65535 bytes
- Solution: Window Scale Option
  - Window size up to 65535 x 2^{14} = 1 Gbyte allowed
  - Requested in SYN segment

*

TCP State Transition Diagram



What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or
  - Server rejects the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and wait for the SYN-ACK
  - … and retransmits the SYN-ACK if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - Some TCPs use a default of 3 or 6 seconds

SYN Loss and Web Downloads

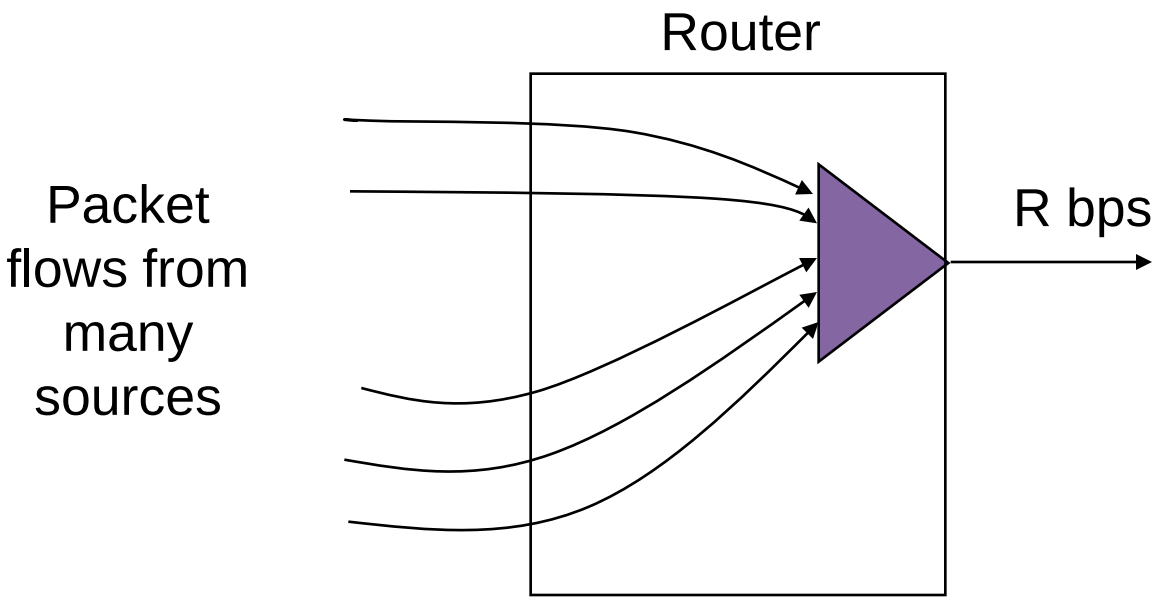
- Example: User clicks on a web link
  - Browser creates a socket and does a “connect”
  - The “connect” triggers the OS to transmit a SYN
- If the SYN is lost…
  - The 3-6 seconds of delay may be very long
  - The user may get impatient … and click the hyperlink again, or click “refresh”
- User triggers an “abort” of the “connect”
  - Browser creates a new socket and does a “connect”
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes fast

Outline

- TCP – Quick Overview
- TCP Header
- TCP Connection Management
- TCP Congestion Control

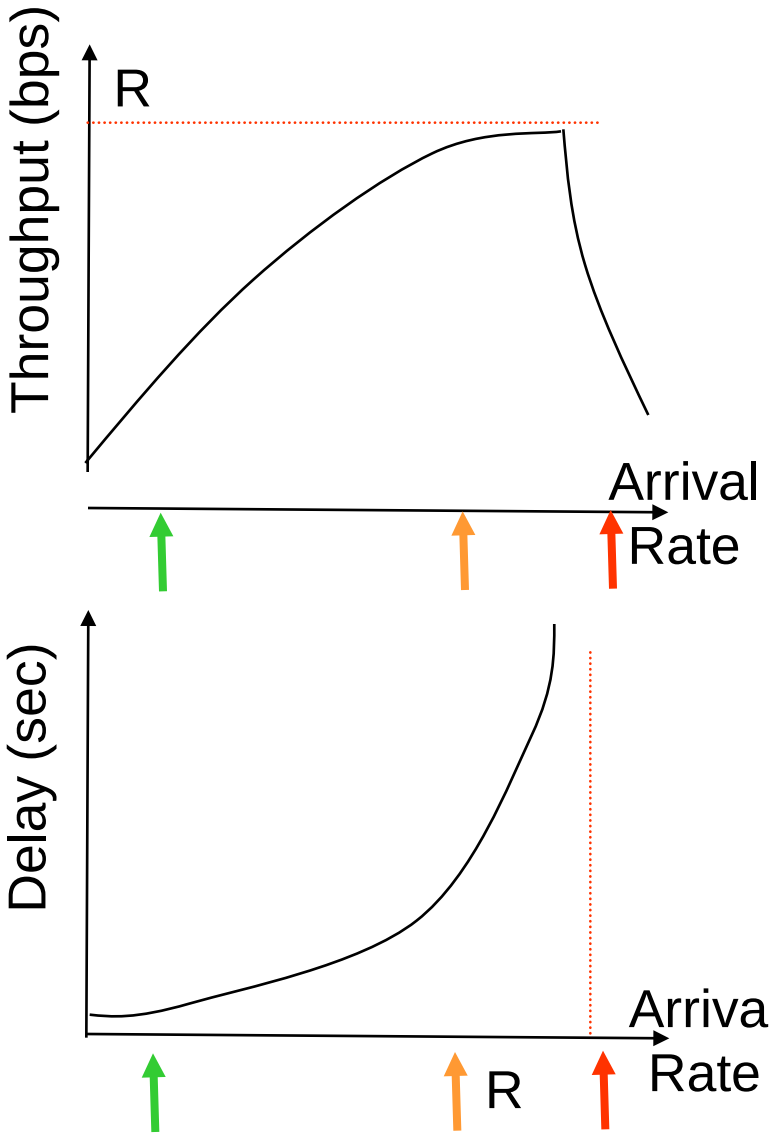
TCP Congestion Control

- *Advertised window* size is used to ensure that receiver's buffer will not overflow
- However, buffers at intermediate routers between source and destination may overflow



- Congestion occurs when total arrival rate from all packet flows exceeds R over a sustained period of time
- Buffers at multiplexer will fill and packets will be lost

Phases of Congestion Behavior



1. **Light traffic**
   - Arrival Rate << R
   - Low delay
   - Can accommodate more
2. **Knee (congestion onset)**
   - Arrival rate approaches R
   - Delay increases rapidly
   - Throughput begins to saturate
3. **Congestion collapse**
   - Arrival rate > R
   - Large delays, packet loss
   - Useful application throughput drops

Window Congestion Control *

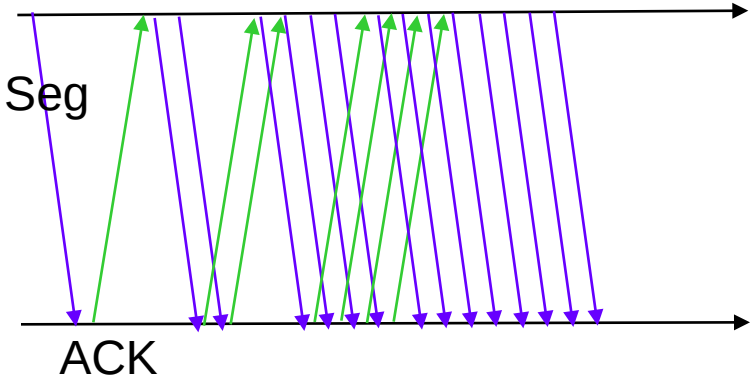
- Desired operating point: just before knee
  - Sources must control their sending rates so that aggregate arrival rate is just before knee
- TCP sender maintains a ***congestion window* cwnd** to control congestion at intermediate routers
- Effective window is minimum of congestion window and advertised window
- Problem: source does not know what its “fair” share of available bandwidth should be
- Solution: adapt dynamically to available BW
  - Sources probe the network by increasing cwnd
  - When congestion detected, sources reduce rate
  - Ideally, sources sending rate stabilizes near ideal point

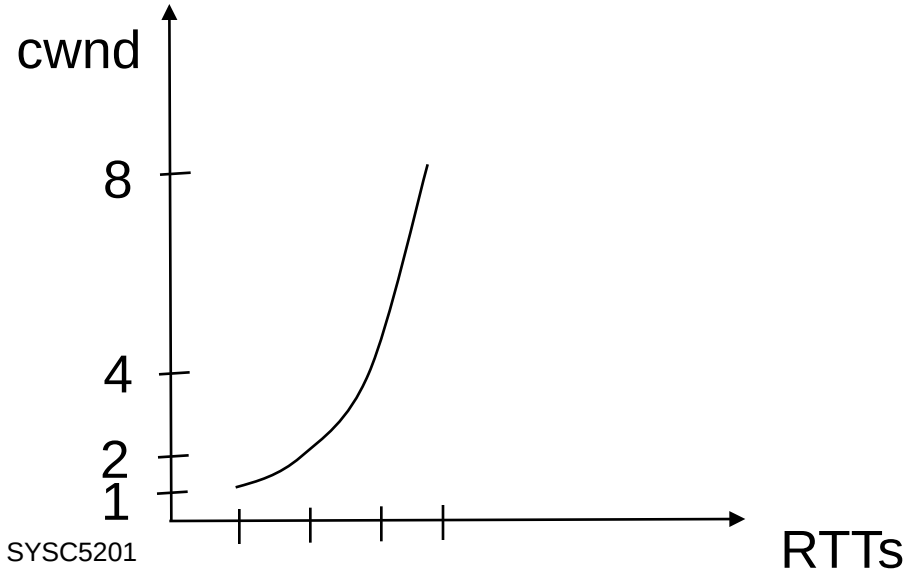
Congestion Window

- How does the TCP congestion algorithm change congestion window dynamically according to the most up-to-date state of the network?
- **At light traffic:** each segment is ACKed quickly
  - Increase cwnd aggresively
- **At knee**: segment ACKs arrive, but more slowly
  - Slow down increase in cwnd
- **At congestion**: segments encounter large delays (so retransmission timeouts occur); segments are dropped in router buffers (resulting in duplicate ACKs)
  - Reduce transmission rate, then probe again

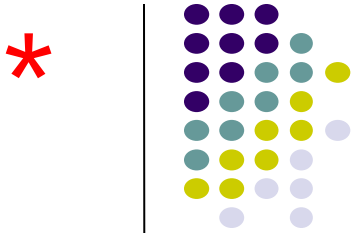
TCP Congestion Control: Slow Start

- **Slow start**: increase congestion window size by one segment upon receiving an **ACK** from receiver
  - initialized at ≤ 2 segments
  - used at (re)start of data transfer
  - congestion window increases **exponentially**

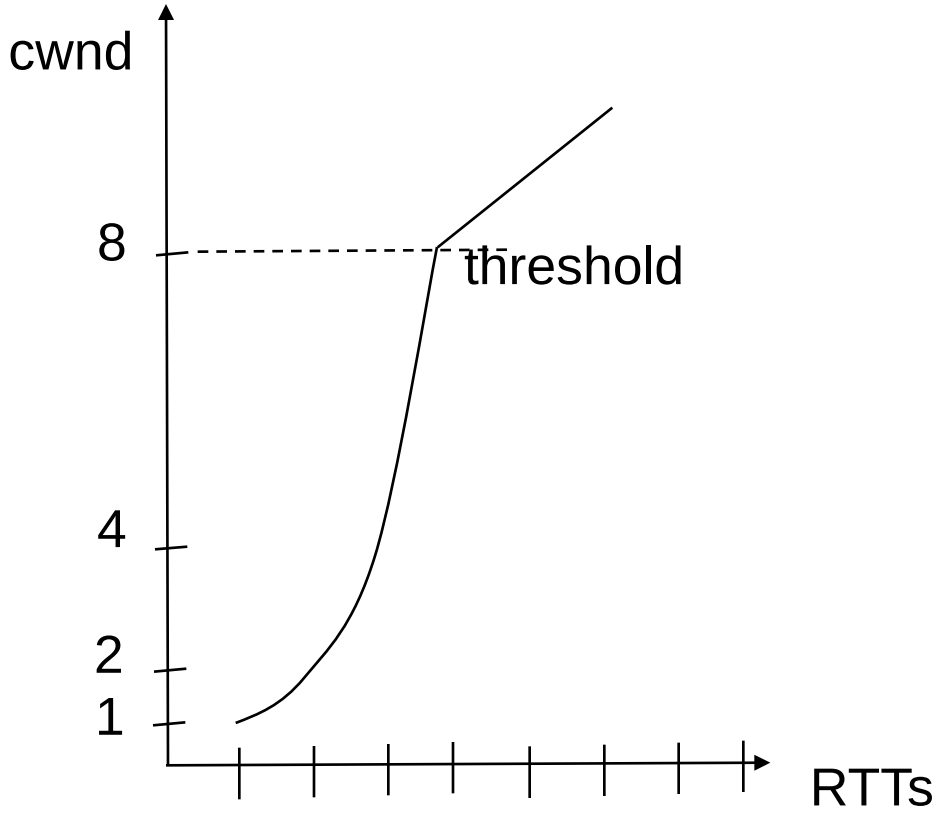




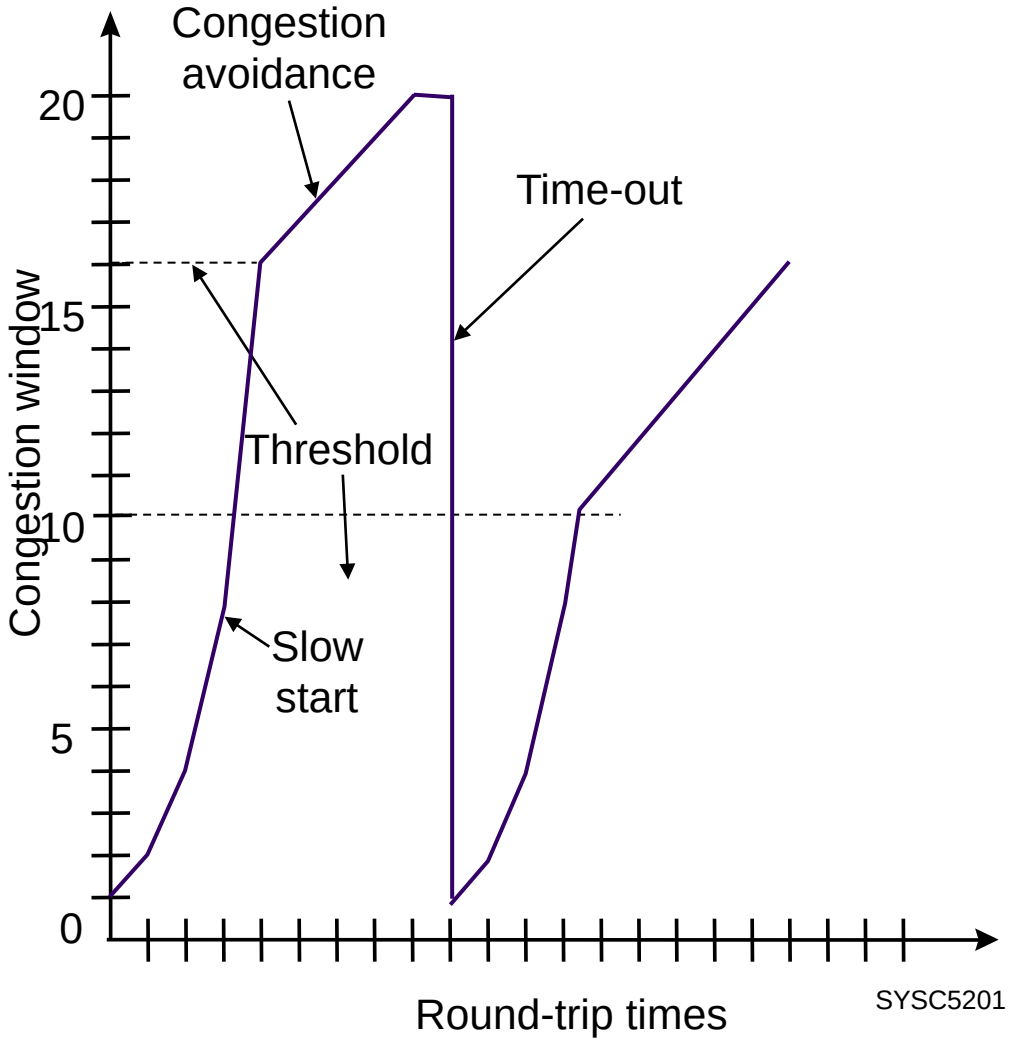
TCP Congestion Control: Congestion Avoidance



- Algorithm progressively sets a *congestion threshold*
  - When cwnd > threshold, slow down rate at which cwnd is increased
- Increase congestion window size by one segment per round-trip-time (RTT)
  - cwnd grows **linearly** with time



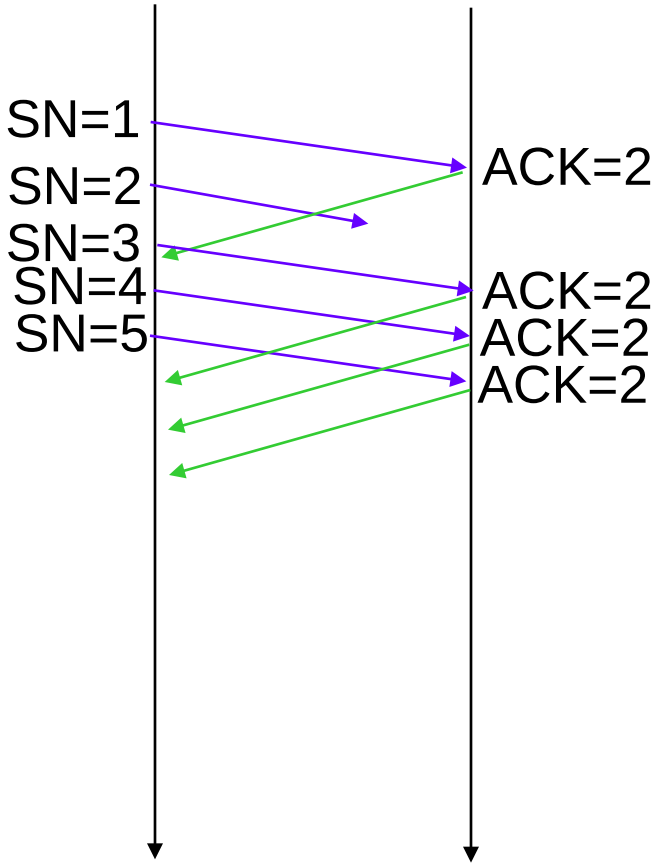
TCP Congestion Control: Congestion



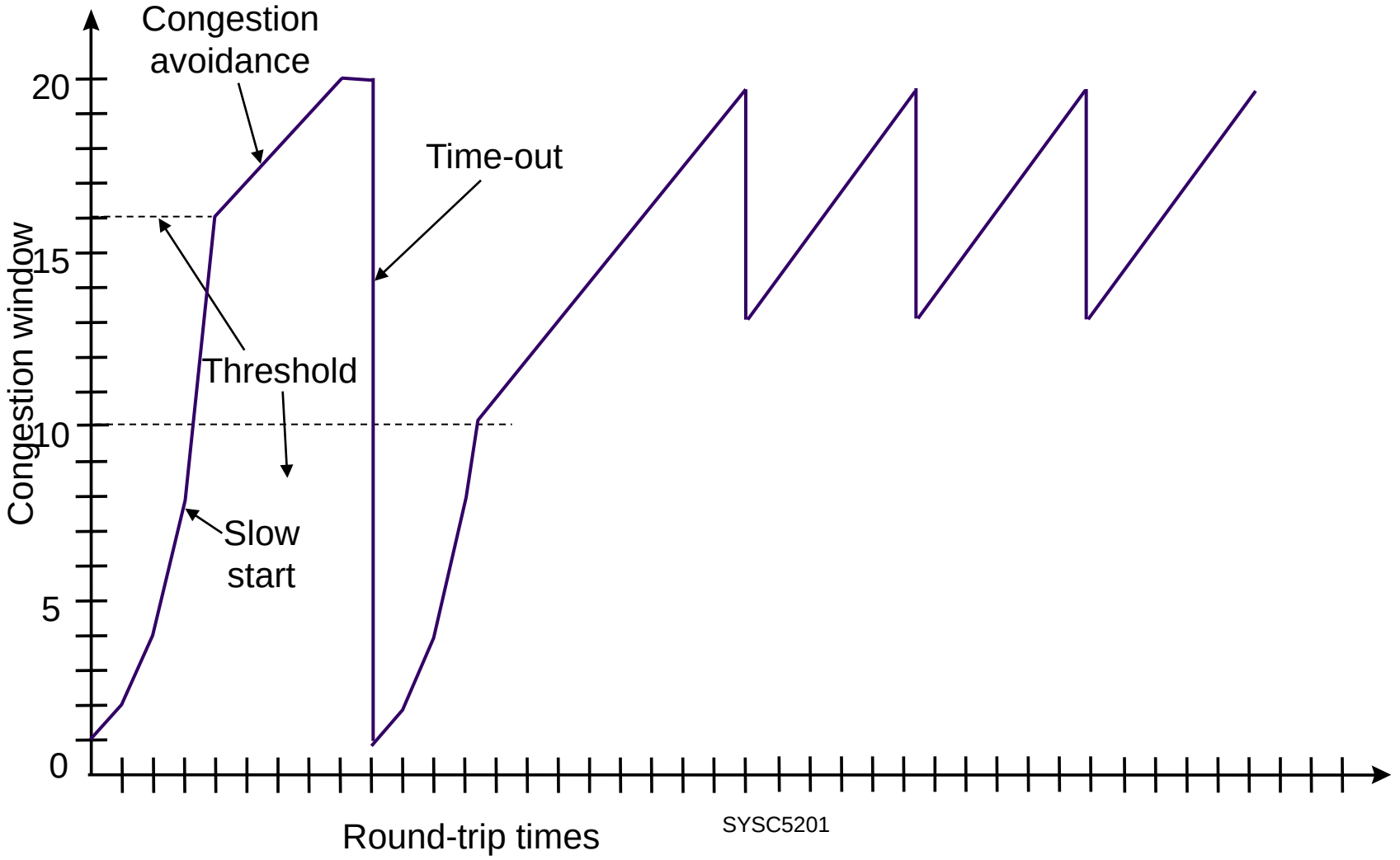
- Congestion is detected upon timeout or receipt of duplicate ACKs
- Assume current cwnd corresponds to available bandwidth
- Adjust congestion threshold = ½ x current cwnd
- Reset cwnd to 1 (or 2)
- Go back to slow-start
- Over several cycles expect to converge to congestion threshold equal to about ½ the available bandwidth

Fast Retransmit & Fast Recovery

- Congestion causes many segments to be dropped
- If only a single segment is dropped, then subsequent segments trigger duplicate ACKs before timeout
- Can avoid large decrease in cwnd as follows:
  - When three duplicate ACKs arrive, retransmit lost segment immediately
  - Reset congestion threshold to ½ cwnd
  - Reset cwnd to congestion threshold + 3 to account for the three segments that triggered duplicate ACKs
  - Remain in congestion avoidance phase
  - However if timeout expires, reset cwnd to 1
  - In absence of timeouts, cwnd will oscillate around optimal value

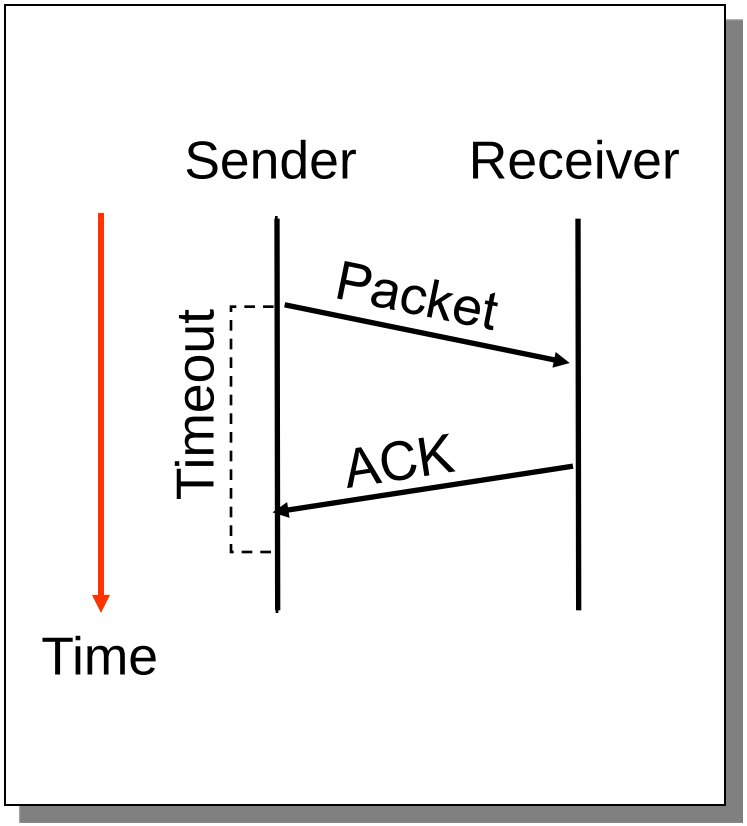


TCP Congestion Control: Fast Retransmit & Fast Recovery

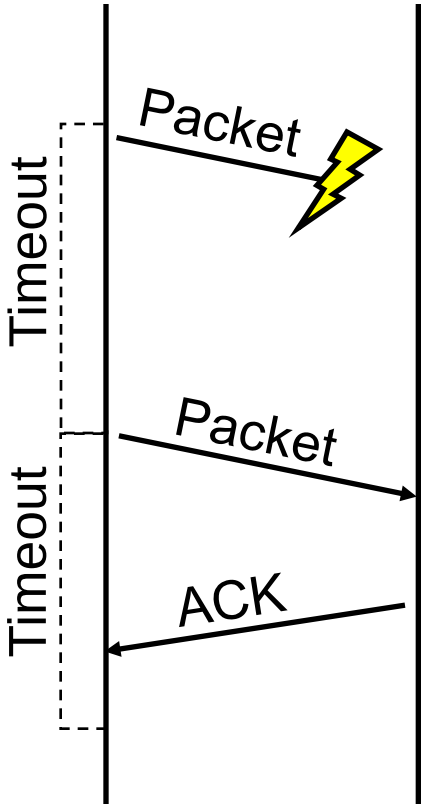


Automatic Repeat reQuest (ARQ)

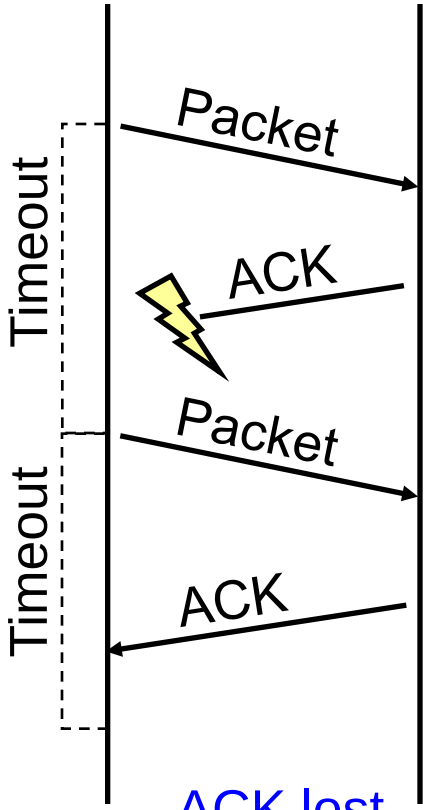
- Automatic Repeat Request
  - Receiver sends acknowledgment (ACK) when it receives a packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period
- Simplest ARQ protocol
  - Stop and wait
  - Send a packet, stop and wait until ACK arrives



Reasons for Retransmission

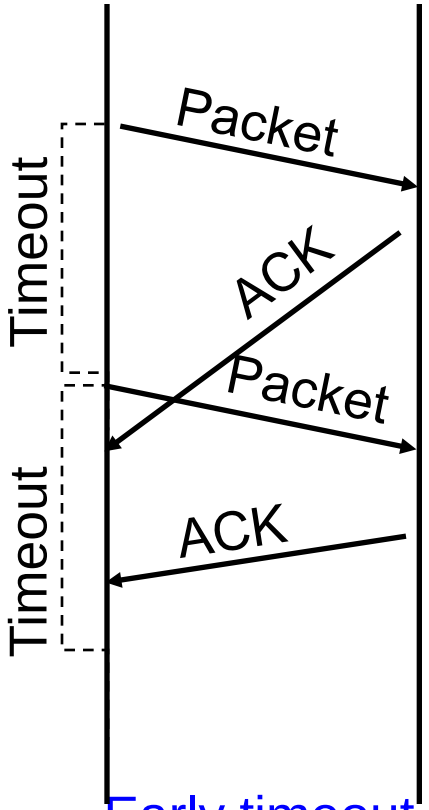


Packet lost



ACK lost

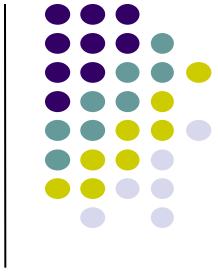
DUPLICATE PACKET



Early timeout

DUPLICATE PACKETS

Time Out: How Long Should Sender Wait?



- Sender sets a timeout to wait for an ACK
  - Too short: wasted retransmissions
  - Too long: excessive delays when packet lost
- TCP sets timeout as **a function of the RTT**
  - Expect ACK to arrive after an RTT

    … plus a fudge factor to account for queuing
- But, how does the sender know the RTT?
  - Can estimate the RTT by watching (measuring) ACKs
  - Smooth estimate: keep a running average of the RTT
    - EstimatedRTT(n) = α * EstimatedRTT(n-1) + (1 – α) * SampleRTT
  - Compute timeout

TCP Retransmission Timeout

- TCP retransmits a segment after timeout period
  - Timeout too short: excessive number of retransmissions
  - Timeout too long: recovery too slow & slow reaction to loss
  - Timeout depends on RTT: time from when segment is sent to when ACK is received
- Round trip time (RTT) in Internet is highly variable
  - Routes vary and can change in mid-connection
  - Traffic fluctuates, multiple traffic flows
- TCP uses adaptive estimation of RTT
  - Measure RTT each time ACK received: M_n

$$t_{RTT}(\text{new}) = \alpha\, t_{RTT}(\text{old}) + (1 - \alpha)\, M_n$$

  - $\alpha = 7/8$ typical

RTT Variability

- Estimate variance σ^2 of RTT variation
- Estimate for timeout:

$$t_{out} = t_{RTT} + k\, d_{RTT}$$

- If RTT highly variable, timeout increase accordingly
- If RTT nearly constant, timeout close to RTT estimate

- Approximate estimation of deviation

$$d_{RTT}(new) = \beta\, d_{RTT}(old) + (1-\beta)\,|\,M_n - t_{RTT}\,|$$

$$t_{out} = t_{RTT} + 4\, d_{RTT} \quad (I.e.\ k=4)$$

RTT and Timeout: an Example

- For packet (n), use timeout (n-1).

- Example: At time 0 the TCP round trip time is actually 30 msec. For the following packets, acknowledgements came back after 26, 32, 24 msec, respectively. Apply the dynamic timeout Jacobson's algorithm to calculate the best timeout estimate at the end. Use α =0.9 and β =0.9. (The notations used in the following are simplified.)

- **Assume at the start d(0)= 0 msec and RTT(0) = 30.**
- **Measured values: M(0)=30, M(1)=26, M(2)=32, M(3)=24.**

- **RTT(n) = α * RTT(n-1) + (1- α) *M (n)** — **d(n) = β * d (n-1) + (1- β) * | RTT(n) -M(n) |**
- **RTT(1) = 0.9 x 30 + 0.1 x 26=29.6** — **d(1)= 0.9 x 0 + 0.1 x |29.6-26|=0.36**
- **RTT(2) = 0.9 x 29.6 + 0.1 x 32=29.84** — **d(2)= 0.9 x .36 + 0.1 x |29.84-32|=0.54**
- **RTT(3) = 0.9 x 29.84 + 0.1 x 24=29.256** — **d(3)= 0.9 x 0.54 + 0.1 x |29.256-24|=1.01**

- **Timeout(n) = RTT(n) + 4 * d(n)**
  Timeout(3) = RTT(3) + 4 * d(3) = 29.256 + 4 x 1.01 = 33.3 msec