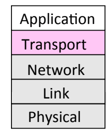
Operating Systems and Networks

Network Lecture 8: Transport Layer

Adrian Perrig
Network Security Group
ETH Zürich

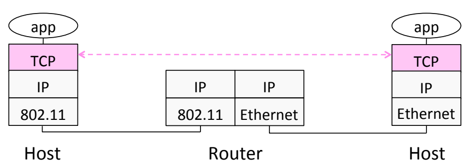
Where we are in the Course

- Starting the Transport Layer!
  - Builds on the network layer to deliver data across networks for applications with the desired reliability or quality



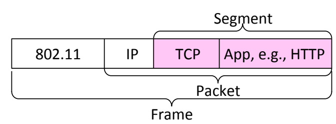
Recall

- Transport layer provides end-to-end connectivity across the network



Recall (2)

- Segments carry application data across the network
- Segments are carried within packets within frames



Transport Layer Services

- Provide different kinds of data delivery across the network to applications

	Unreliable	**Reliable**
Messages	Datagrams (UDP)	
Bytestream		Streams (TCP)

Comparison of Internet Transports

- TCP is full-featured, UDP is a glorified packet

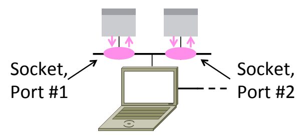
TCP (Streams)	**UDP (Datagrams)**
Connections	Datagrams
Bytes are delivered once, reliably, and in order	Messages may be lost, reordered, duplicated
Arbitrary length content	Limited message size
Flow control matches sender to receiver	Can send regardless of receiver state
Congestion control matches sender to network	Can send regardless of network state

Socket API

- Simple abstraction to use the network
  - The "network" API (really Transport service) used to write all Internet apps
  - Part of all major OSes and languages; originally Berkeley (Unix) ~1983
- Supports both Internet transport services (Streams and Datagrams)

Socket API (2)

- Sockets let apps attach to the local network at different ports



Socket API (3)

- Same API used for Streams and Datagrams

	Primitive	Meaning
	SOCKET	Create a new communication endpoint
	BIND	Associate a local address (port) with a socket
Only needed for Streams	LISTEN	Announce willingness to accept connections
	ACCEPT	Passively establish an incoming connection
	CONNECT	Actively attempt to establish a connection
To/From forms for Datagrams	SEND(TO)	Send some data over the socket
	RECEIVE(FROM)	Receive some data over the socket
	CLOSE	Release the socket

Ports

- Application process is identified by the tuple IP address, protocol, and port
  - Ports are 16-bit integers representing local "mailboxes" that a process leases
- Servers often bind to "well-known ports"
  - <1024, require administrative privileges
- Clients often assigned "ephemeral" ports
  - Chosen by OS, used temporarily

Some Well-Known Ports

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Topics

This time:

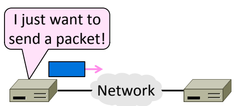
- Service models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Connections (TCP)
- Sliding Window (TCP)
- Flow control (TCP)
- Retransmission timers (TCP)

Later:

- Congestion control (TCP)

User Datagram Protocol (UDP) (§6.4)

- Sending messages with UDP
  - A shim layer on packets

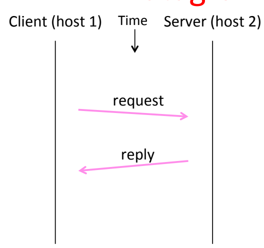


User Datagram Protocol (UDP)

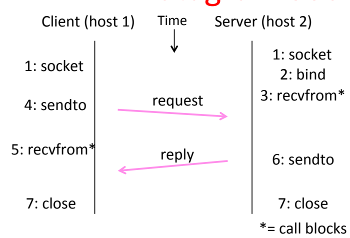
- Used by apps that don't want reliability or bytestreams
  - Voice-over-IP (unreliable)
  - DNS, RPC (message-oriented)
  - DHCP (bootstrapping)

(If application wants reliability and messages then it has work to do!)

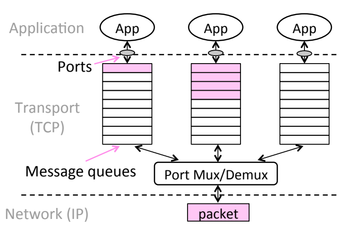
Datagram Sockets



Datagram Sockets (2)



UDP Buffering



UDP Header

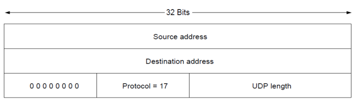
- Uses ports to identify sending and receiving application processes
- Datagram length up to 64K
- Checksum (16 bits) for reliability

32 Bits

Source port	Destination port
UDP length	UDP checksum

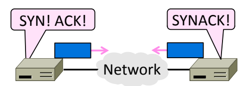
UDP Header (2)

- Optional checksum covers UDP segment and IP pseudoheader
  - Checks key IP fields (addresses)
  - Value of zero means "no checksum"



Connection Establishment (6.5.5, 6.5.7, 6.2.2)

- How to set up connections
  - We'll see how TCP does it

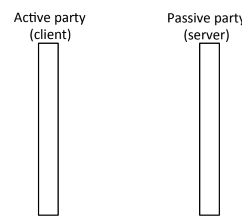


Connection Establishment

- Both sender and receiver must be ready before we start the transfer of data
  - Need to agree on a set of parameters
  - e.g., the Maximum Segment Size (MSS)
- This is signaling
  - It sets up state at the endpoints
  - Like "dialing" for a telephone call

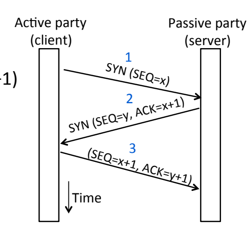
Three-Way Handshake

- Used in TCP; opens connection for data in both directions
- Each side probes the other with a fresh Initial Sequence Number (ISN)
  - Sends on a SYNchronize segment
  - Echo on an ACKnowledge segment
- Chosen to be robust even against delayed duplicates



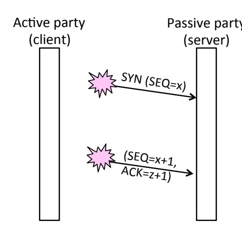
Three-Way Handshake (2)

- Three steps:
  - Client sends SYN(x)
  - Server replies with SYN(y)ACK(x+1)
  - Client replies with ACK(y+1)
  - SYNs are retransmitted if lost
- Sequence and ack numbers carried on further segments



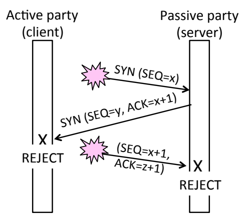
Three-Way Handshake (3)

- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  - Improbable, but anyhow …



Three-Way Handshake (4)

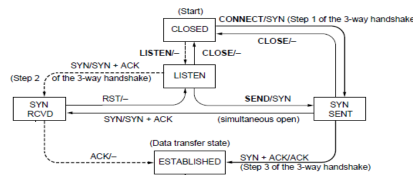
- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  - Improbable, but anyhow …
- Connection will be cleanly rejected on both sides ☺



TCP Connection State Machine

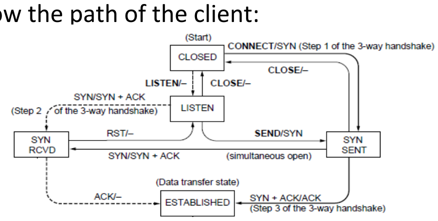
- Captures the states (rectangles) and transitions (arrows)
  - A/B means event A triggers the transition, with action B

Both parties run instances of this state machine



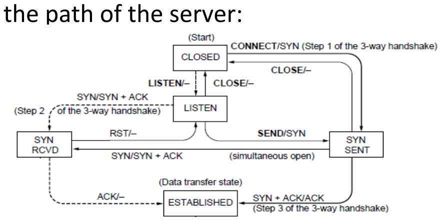
TCP Connections (2)

- Follow the path of the client:



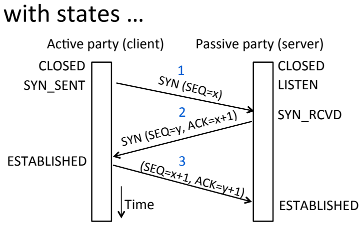
TCP Connections (3)

- And the path of the server:



TCP Connections (4)

- Again, with states …

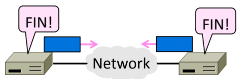


TCP Connections (5)

- Finite state machines are a useful tool to specify and check the handling of all cases that may occur
- TCP allows for simultaneous open
  - i.e., both sides open at once instead of the client-server pattern
  - Try at home to confirm it works ☺

Connection Release (6.5.6-6.5.7, 6.2.3)

- How to release connections
  - We'll see how TCP does it

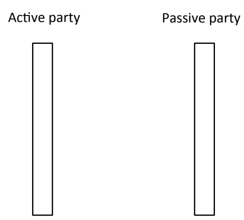


Connection Release

- Orderly release by both parties when done
  - Delivers all pending data and "hangs up"
  - Cleans up state in sender and receiver
- Key problem is to provide reliability while releasing
  - TCP uses a "symmetric" close in which both sides shutdown independently

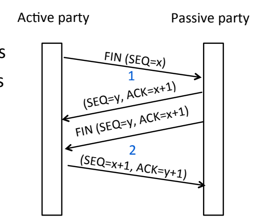
TCP Connection Release

- Two steps:
  - Active sends FIN(x), passive ACKs
  - Passive sends FIN(y), active ACKs
  - FINs are retransmitted if lost
- Each FIN/ACK closes one direction of data transfer



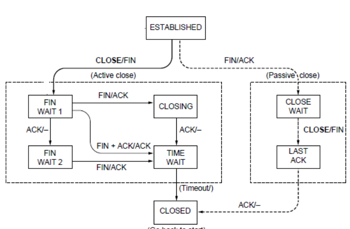
TCP Connection Release (2)

- Two steps:
  - Active sends FIN(x), passive ACKs
  - Passive sends FIN(y), active ACKs
  - FINs are retransmitted if lost
- Each FIN/ACK closes one direction of data transfer



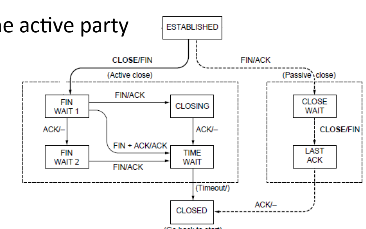
TCP Connection State Machine

Both parties run instances of this state machine



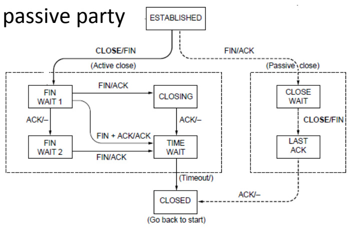
TCP Release

- Follow the active party



TCP Release (2)

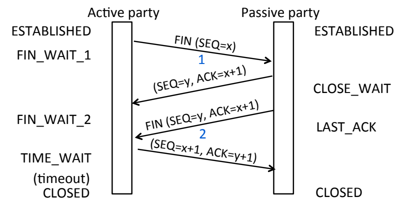
- Follow the passive party



Computer Networks 37

TCP Release (3)

- Again, with states …



Computer Networks 38

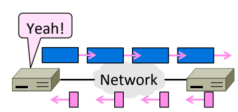
TIME_WAIT State

- We wait a long time (two times the maximum segment lifetime of 60 seconds) after sending all segments and before completing the close
- Why?
  – ACK might have been lost, in which case FIN will be resent for an orderly close
  – Could otherwise interfere with a subsequent connection

Computer Networks 39

Sliding Windows (§3.4, §6.5.8)

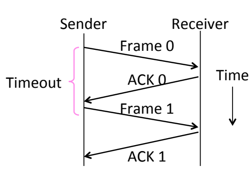
- The sliding window algorithm
  – Pipelining and reliability
  – Building on Stop-and-Wait



Computer Networks 40

Recall

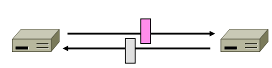
- ARQ with one message at a time is Stop-and-Wait (normal case below)



Computer Networks 41

Limitation of Stop-and-Wait

- It allows only a single message to be outstanding from the sender:
  – Fine for LAN (only one frame fit)
  – Not efficient for network paths with BD >> 1 packet



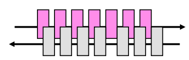
Computer Networks 42

Limitation of Stop-and-Wait (2)

- Example: R=1 Mbps, D = 50 ms
  - RTT (Round Trip Time) = 2D = 100 ms
  - How many packets/sec?
  - What if R=10 Mbps?

Sliding Window

- Generalization of stop-and-wait
  - Allows W packets to be outstanding
  - Can send W packets per RTT (=2D)



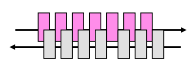
  - Pipelining improves performance
  - Need W=2BD to fill network path

Sliding Window (2)

- What W will use the network capacity?
- Ex: R=1 Mbps, D = 50 ms
- Ex: What if R=10 Mbps?

Sliding Window (3)

- Ex: R=1 Mbps, D = 50 ms
  - 2BD = 10^6 b/sec x 100. 10^{-3} sec = 100 kbit
  - W = 2BD = 10 packets of 1250 bytes



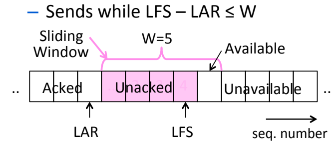
- Ex: What if R=10 Mbps?
  - 2BD = 1000 kbit
  - W = 2BD = 100 packets of 1250 bytes

Sliding Window Protocol

- Many variations, depending on how buffers, acknowledgements, and retransmissions are handled
- Go-Back-N
  - Simplest version, can be inefficient
- Selective Repeat
  - More complex, better performance

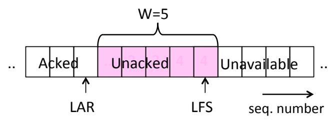
Sliding Window – Sender

- Sender buffers up to W segments until they are acknowledged
  - LFS=LAST FRAME SENT, LAR=LAST ACK REC'D
  - Sends while LFS – LAR ≤ W



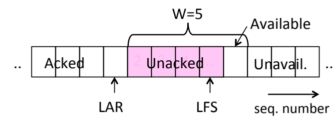
Sliding Window – Sender (2)

- Transport accepts another segment of data from the Application ...
  - Transport sends it (as LFS–LAR → 5)



Sliding Window – Sender (3)

- Next higher ACK arrives from peer…
  - Window advances, buffer is freed
  - LFS–LAR → 4 (can send one more)



Sliding Window – Go-Back-N

- Receiver keeps only a single packet buffer for the next segment
  - State variable, LAS = LAST ACK SENT
- On receive:
  - If seq. number is LAS+1, accept and pass it to app, update LAS, send ACK
  - Otherwise discard (as out of order)

Sliding Window – Selective Repeat

- Receiver passes data to app in order, and buffers out-of-order segments to reduce retransmissions
- ACK conveys highest in-order segment, plus hints about out-of-order segments
- TCP uses a selective repeat design; we'll see the details later

Sliding Window – Selective Repeat (2)

- Buffers W segments, keeps state variable, LAS = LAST ACK SENT
- On receive:
  - Buffer segments [LAS+1, LAS+W]
  - Pass up to app in-order segments from LAS+1, and update LAS
  - Send ACK for LAS regardless

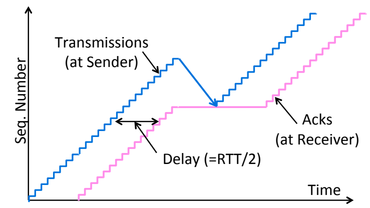
Sliding Window – Retransmissions

- Go-Back-N sender uses a single timer to detect losses
  - On timeout, resends buffered packets starting at LAR+1
- Selective Repeat sender uses a timer per unacked segment to detect losses
  - On timeout for segment, resend it
  - Hope to resend fewer segments

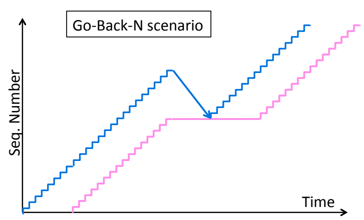
Sequence Numbers

- Need more than 0/1 for Stop-and-Wait …
  - But how many?
- For Selective Repeat, need W numbers for packets, plus W for acks of earlier packets
  - 2W seq. numbers
  - Fewer for Go-Back-N (W+1)
- Typically implement seq. number with an N-bit counter that wraps around at $2^N—1$
  - E.g., N=8: …, 253, 254, 255, 0, 1, 2, 3, …

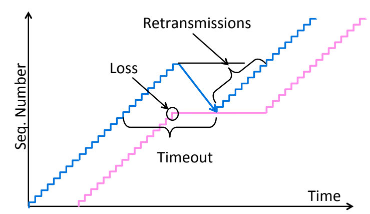
Sequence Time Plot



Sequence Time Plot (2)

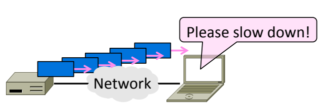


Sequence Time Plot (3)



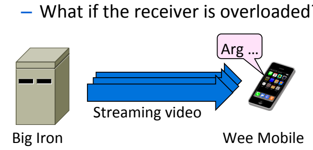
Flow Control (§6.5.8)

- Adding flow control to the sliding window algorithm
  - To slow the over-enthusiastic sender



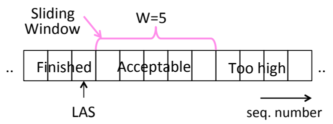
Problem

- Sliding window uses pipelining to keep the network busy
  - What if the receiver is overloaded?



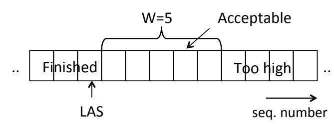
Sliding Window – Receiver

- Consider receiver with W buffers
  - LAS=LAST ACK SENT, app pulls in-order data from buffer with recv() call



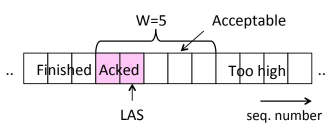
Sliding Window – Receiver (2)

- Suppose the next two segments arrive but app does not call recv()



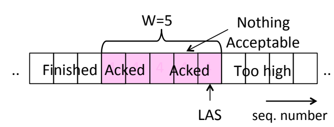
Sliding Window – Receiver (3)

- Suppose the next two segments arrive but app does not call recv()
  - LAS rises, but we can't slide window!



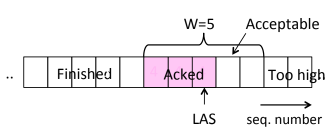
Sliding Window – Receiver (4)

- If further segments arrive (even in order) we can fill the buffer
  - Must drop segments until app recvs!



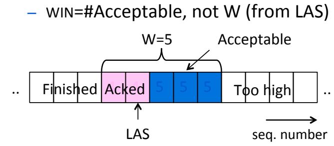
Sliding Window – Receiver (5)

- App recv() takes two segments
  - Window slides (phew)



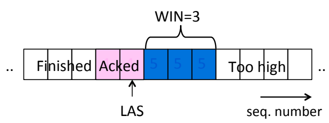
Flow Control

- Avoid loss at receiver by telling sender the available buffer space
  - WIN=#Acceptable, not W (from LAS)



Flow Control (2)

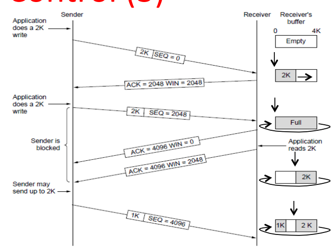
- Sender uses the lower of the sliding window and <u>flow control window</u> (WIN) as the effective window size



Computer Networks 67

Flow Control (3)

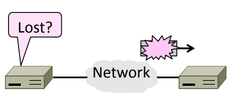
- TCP-style example
  - SEQ/ACK sliding window
  - Flow control with WIN
  - SEQ + length < ACK+WIN
  - 4KB buffer at receiver
  - Circular buffer of bytes



Computer Networks 68

Retransmission Timeouts (§6.5.9)

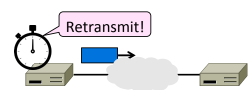
- How to set the timeout for sending a retransmission
  - Adapting to the network path



Computer Networks 69

Retransmissions

- With sliding window, the strategy for detecting loss is the <u>timeout</u>
  - Set timer when a segment is sent
  - Cancel timer when ack is received
  - If timer fires, <u>retransmit</u> data as lost



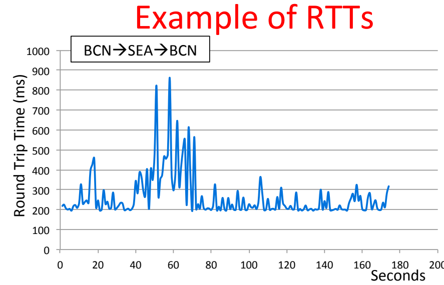
Computer Networks 70

Timeout Problem

- Timeout should be "just right"
  - Too long wastes network capacity
  - Too short leads to spurious resends
  - But what is "just right"?
- Easy to set on a LAN (Link)
  - Short, fixed, predictable RTT
- Hard on the Internet (Transport)
  - Wide range, variable RTT

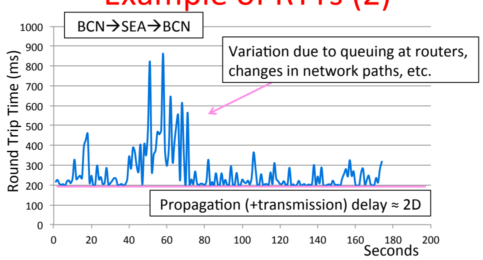
Computer Networks 71

Example of RTTs



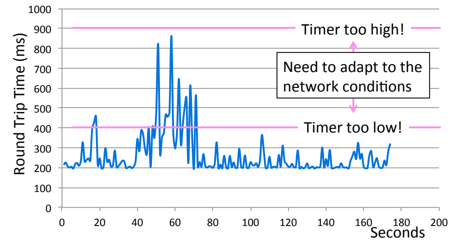
Computer Networks 72

Example of RTTs (2)



Computer Networks 73

Example of RTTs (3)



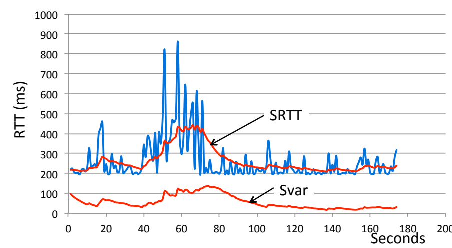
Computer Networks 74

Adaptive Timeout

- Keep smoothed estimates of the RTT (1) and variance in RTT (2)
  - Update estimates with a moving average
  1. $SRTT_{N+1} = 0.9*SRTT_N + 0.1*RTT_{N+1}$
  2. $Svar_{N+1} = 0.9*Svar_N + 0.1*|RTT_{N+1} - SRTT_{N+1}|$
- Set timeout to a multiple of estimates
  - To estimate the upper RTT in practice
  - TCP $Timeout_N = SRTT_N + 4*Svar_N$

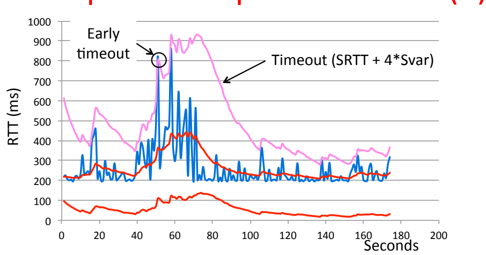
Computer Networks 75

Example of Adaptive Timeout



Computer Networks 76

Example of Adaptive Timeout (2)



Computer Networks 77

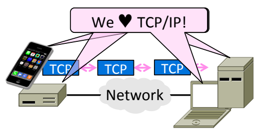
Adaptive Timeout (2)

- Simple to compute, does a good job of tracking actual RTT
  - Little "headroom" to lower
  - Yet very few early timeouts
- Turns out to be important for good performance and robustness

Computer Networks 78

Transmission Control Protocol (TCP) (§6.5)

- How TCP works!
  - The transport protocol used for most content on the Internet



TCP Features

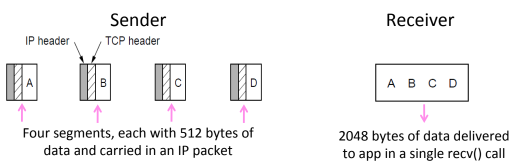
- A reliable bytestream service
- Based on connections
- Sliding window for reliability
  - With adaptive timeout
- Flow control for slow receivers

} This time

- Congestion control to allocate network bandwidth } Next time

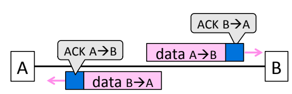
Reliable Bytestream

- Message boundaries not preserved from send() to recv()
  - But reliable and ordered (receive bytes in same order as sent)



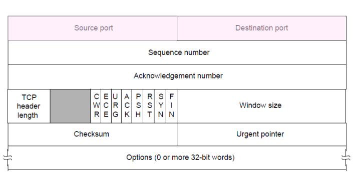
Reliable Bytestream (2)

- Bidirectional data transfer
  - Control information (e.g., ACK) piggybacks on data segments in reverse direction



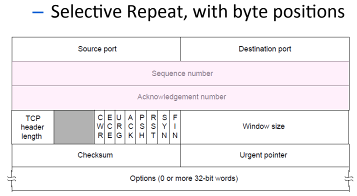
TCP Header (1)

- Ports identify apps (socket API)
  - 16-bit identifiers



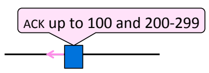
TCP Header (2)

- SEQ/ACK used for sliding window
  - Selective Repeat, with byte positions



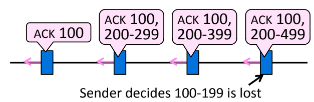
TCP Sliding Window – Receiver

- Cumulative ACK tells next expected byte sequence number ("LAS+1")
- Optionally, selective ACKs (SACK) give hints for receiver buffer state
  - List up to 3 ranges of received bytes



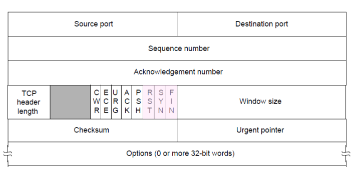
TCP Sliding Window – Sender

- Uses adaptive retransmission timeout to resend data from LAS+1
- Uses heuristics to infer loss quickly and resend to avoid timeouts
  - "Three duplicate ACKs" treated as loss



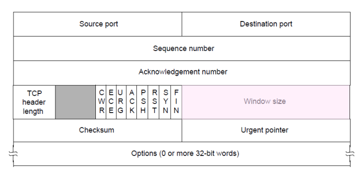
TCP Header (3)

- SYN/FIN/RST flags for connections
  - Flag indicates segment is a SYN etc.



TCP Header (4)

- Window size for flow control
  - Relative to ACK, and in bytes



Other TCP Details

- Many, many quirks you can learn about its operation
  - But they are the details
- Biggest remaining mystery is the workings of congestion control
  - We'll tackle this next time!