15-441 Computer Networking

Transport Layer

Functionality Split

- Network provides best-effort delivery
- End-systems implement many functions
  - Reliability
  - In-order delivery
  - Demultiplexing
  - Message boundaries
  - Connection abstraction
  - Congestion control
  - …

Overview

- Transport introduction
- Error recovery
- TCP flow control
- TCP connection setup/data transfer
- TCP reliability

Transport Protocols

- UDP provides just integrity and demux
- TCP adds…
  - Connection-oriented
  - Reliable
  - Ordered
  - Point-to-point
  - Byte-stream
  - Full duplex
  - Flow and congestion controlled

UDP: User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

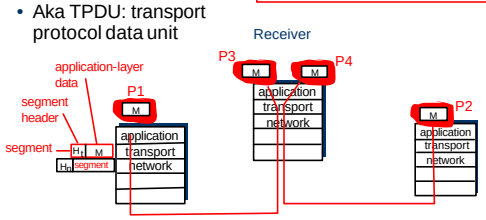
Why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small segment header
- No congestion control: UDP can blast away as fast as desired

Multiplexing & Demultiplexing

- Recall: *segment* - unit of data exchanged between transport layer entities
  - Aka TPDU: transport protocol data unit

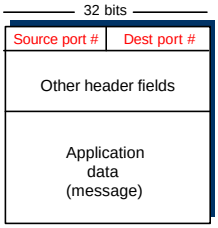
Demultiplexing: delivering received segments to correct app layer processes



Multiplexing & Demultiplexing

Multiplexing:
Gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

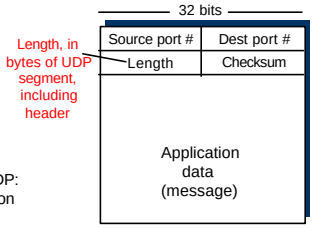
- Based on sender, receiver port numbers, IP addresses
  - Source, dest port #s in each segment
  - Recall: well-known port numbers for specific applications



TCP/UDP segment format

UDP, cont.

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recover!



UDP segment format

UDP Checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:
- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1's complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

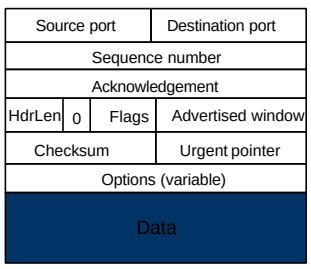
Receiver:
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonethless?*

High-Level TCP Characteristics

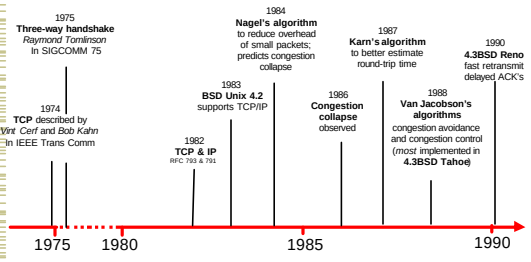
- Protocol implemented entirely at the ends
  - Fate sharing
- Protocol has evolved over time and will continue to do so
  - Nearly impossible to change the header
  - Uses options to add information to the header
  - Change processing at endpoints
  - Backward compatibility is what makes it TCP

TCP Header

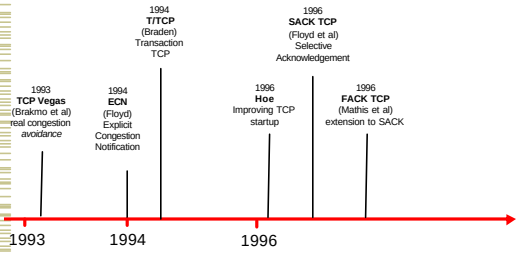
Flags: SYN
FIN
RESET
PUSH
URG
ACK



Evolution of TCP



TCP Through the 1990s



Overview

- Transport introduction
- Error recovery
- TCP flow control
- TCP connection setup/data transfer
- TCP reliability

Transport vs. Link Layers

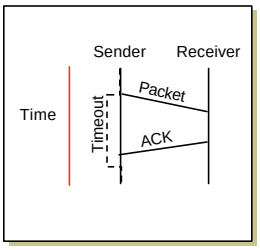
- Logical link vs. physical link
  - Must establish connection
- Variable RTT
  - May vary within a connection
- Reordering
  - How long can packets live ≪ max segment lifetime
- Can't expect endpoints to exactly match link
  - Buffer space availability
- Transmission rate
  - Don't directly know transmission rate

Error Recovery

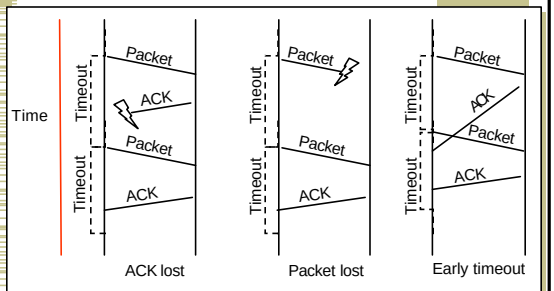
- Two forms of error recovery
  - Forward Error Correction (FEC)
  - Automatic Repeat Request (ARQ)
- FEC
  - Use error correcting codes to repair losses
- ARQ
  - Receiver sends acknowledgement (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period

Stop and Wait

- Simplest ARQ protocol
- Send a packet, stop and wait until acknowledgement arrives



Recovering from Error

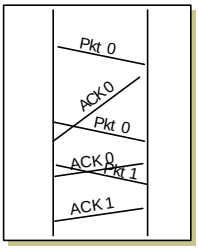


Problems with Stop and Wait

- How to recognize a duplicate
- Performance
  - Can only send one packet per round trip

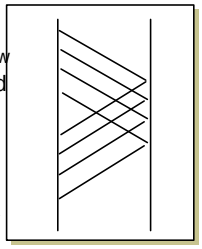
How to Recognize Resends?

- Use sequence numbers
  - both packets and acks
- Sequence # in packet is finite -- how big should it be?
  - For stop and wait?
- One bit – won't send seq #1 until received ACK for seq #0



How to Keep the Pipe Full?

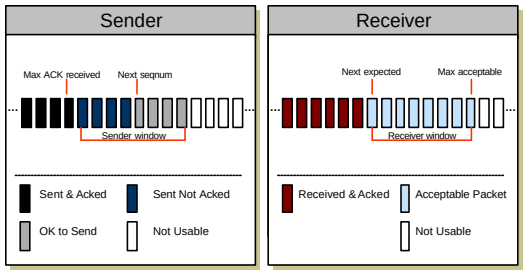
- Send multiple packets without waiting for first to be acked
  - Number of pkts in flight = window
- How large a window is needed
  - Round trip delay * bandwidth = capacity of pipe
- Reliable, unordered delivery
  - Several parallel stop & waits
  - Send new packet after each ack
  - Sender keeps list of unack'ed packets; resends after timeout
  - Receiver same as stop&wait



Sliding Window

- Reliable, ordered delivery
- Receiver has to hold onto a packet until all prior packets have arrived
- Sender must prevent buffer overflow at receiver
- Circular buffer at sender and receiver
  - Packets in transit ? buffer size
  - Advance when sender and receiver agree packets at beginning have been received

Sender/Receiver State



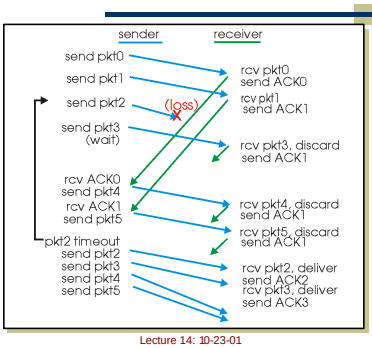
Window Sliding – Common Case

- On reception of new ACK (i.e. ACK for something that was not acked earlier)
  - Increase sequence of max ACK received
  - Send next packet
- On reception of new in-order data packet (next expected)
  - Hand packet to application
  - Send cumulative ACK – acknowledges reception of all packets up to sequence number
  - Increase sequence of max acceptable packet

Loss Recovery

- On reception of out-of-order packet
  - Send nothing (wait for source to timeout)
  - Cumulative ACK (helps source identify loss)
- Timeout (Go-Back-N recovery)
  - Set timer upon transmission of packet
  - Retransmit all unacknowledged packets
- Performance during loss recovery
  - No longer have an entire window in transit
  - Can have much more clever loss recovery

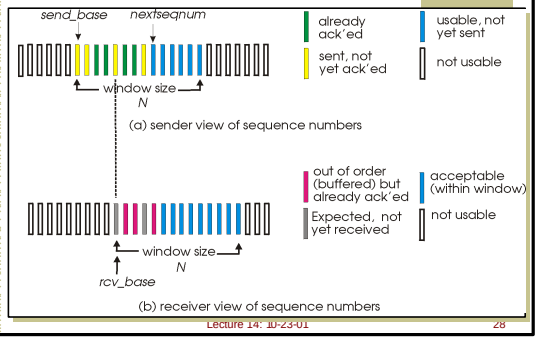
Go-Back-N in Action



Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each unACKed packet
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed packets

Selective Repeat: Sender, Receiver Windows



Sequence Numbers

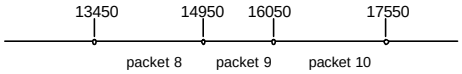
- How large do sequence numbers need to be?
  - Must be able to detect wrap-around
  - Depends on sender/receiver window size
- E.g.
  - Max seq = 7, send win=recv win=7
  - If pkts 0..6 are sent succesfully and all acks lost
    - Receiver expects 7,0..5, sender retransmits old 0..6!!!
- Max sequence must be ? send window + recv window

Overview

- Transport introduction
- Error recovery
- TCP flow control
- TCP connection setup/data transfer
- TCP reliability

Sequence Number Space

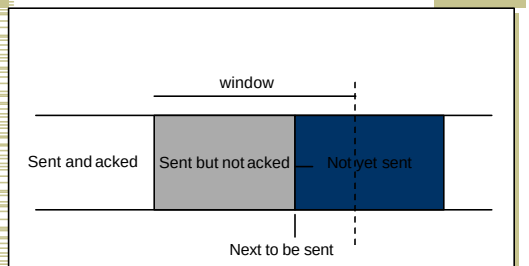
- Each byte in byte stream is numbered.
  - 32 bit value
  - Wraps around
  - Initial values selected at start up time
- TCP breaks up the byte stream in packets.
  - Packet size is limited to the Maximum Segment Size
- Each packet has a sequence number.
  - Indicates where it fits in the byte stream



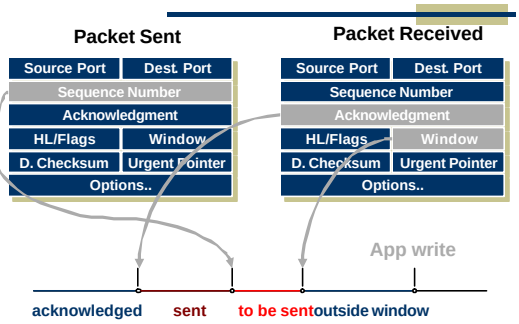
TCP Flow Control

- TCP is a sliding window protocol
  - For window size *n*, can send up to *n* bytes without receiving an acknowledgement
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - Congestion control now limits this

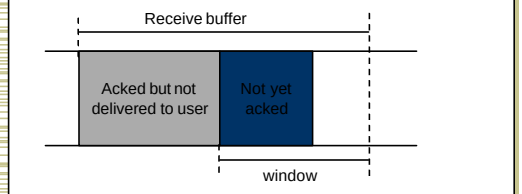
Window Flow Control: Send Side



Window Flow Control: Send Side



Window Flow Control: Receive Side



TCP Persist

- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?
- TCP Persist state
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK even if it can't store the packet

Performance Considerations

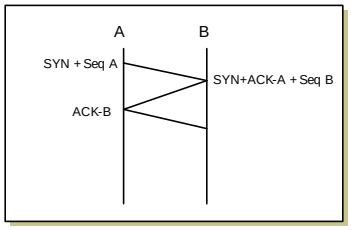
- The window size can be controlled by receiving application
  - Can change the socket buffer size from a default (e.g. 16Kbytes) to a maximum value (e.g. 64 Kbytes)
- The window size field in the TCP header limits the window that the receiver can advertise
  - 16 bits ⇙ 64 KBytes
  - 10 msec RTT ⇙ 51 Mbit/second
  - 100 msec RTT ⇙ 5 Mbit/second

Overview

- Transport introduction
- Error recovery
- TCP flow control
- TCP connection setup/data transfer
- TCP reliability

Connection Establishment

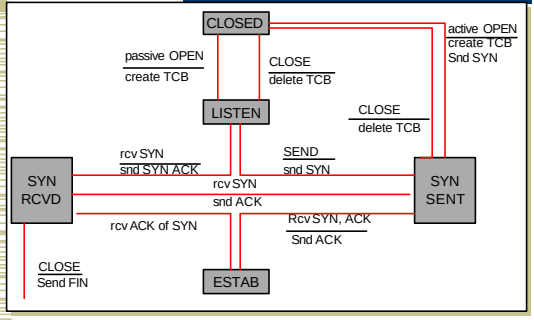
- A and B must agree on initial sequence number selection
  - Use 3-way handshake



Sequence Number Selection

- Why not simply chose 0?
- Must avoid overlap with earlier incarnation

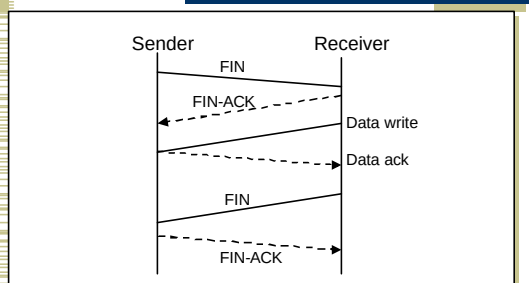
Connection Setup



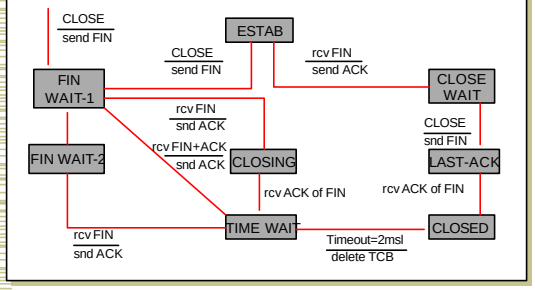
Connection Tear-down

- Normal termination
  - Allow unilateral close
- TCP must continue to receive data even after closing
- Cannot close connection immediately
  - What if a new connection restarts and uses same sequence number?

Tear-down Packet Exchange



Connection Tear-down



Detecting Half-open Connections

	TCP A		TCP B
1.	(CRASH)		(send 300, receive 100)
2.	CLOSED		ESTABLISHED
3.	SYN-SENT	<SEQ=400><CTL=SYN>	(??)
4.	(!!)	<SEQ=300><ACK=100><CTL=ACK>	ESTABLISHED
5.	SYN-SENT	<SEQ=100><CTL=RST>	(Abort!!)
6.	SYN-SENT		CLOSED
7.	SYN-SENT	<SEQ=400><CTL=SYN>	

Observed TCP Problems

- Too many small packets
  - Silly window syndrome
  - Nagel's algorithm
- Initial sequence number selection
- Amount of state maintained

Silly Window Syndrome

- Problem: (Clark, 1982)
  - If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets
- Solution
  - Receiver must not advertise small window increases
  - Increase window by min(MSS,RecvBuffer/2)

Nagel's Algorithm

- Small packet problem:
  - Don't want to send a 41 byte packet for each keystroke
  - How long to wait for more data?
- Solution:
  - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged

Why is Selecting ISN Important?

- Suppose machine X selects ISN based on predictable sequence
- Fred has .rhosts to allow login to X from Y
- Evil Ed attacks
  - Disables host Y – denial of service attack
  - Make a bunch of connections to host X
  - Determine ISN pattern a guess next ISN
  - Fake pkt1: [<src Y><dst X>, guessed ISN]
  - Fake pkt2: desired command

Time Wait Issues

- Web servers not clients close connection first
  - Established ✍ Fin-Waits ✍ Time-Wait ✍ Closed
  - Why would this be a problem?
- Time-Wait state lasts for 2 * MSL
  - MSL should be 120 seconds (is often 60s)
  - Servers often have order of magnitude more connections in Time-Wait

Overview

- Transport introduction
- Error recovery
- TCP flow control
- TCP connection setup/data transfer
- TCP reliability

Reliability Challenges

- Like reliability on links
  - Similar techniques (timeouts, acknowledgements, etc.)
- New challenges
  - Congestion related losses
  - Variable packet delays
    - What should the timeout be?
  - Reordering of packets
    - Ensure sequences numbers are not reused
    - How long do packets live?
      - MSL = 120 seconds based on IP behavior

TCP = Go-Back-N Variant

- Receiver can only return a single “ack” sequence number to the sender.
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
- But: sender only retransmits a single packet.
  - Reason???
- Error control is based on byte sequences, not packets.
  - Retransmitted packet can be different from the original lost packet – why?
  - Packets can overlap – why?
- Sliding window with cumulative acks
  - Ack field contains last in-order packet received
  - Duplicate acks sent when out-of-order packet received

Round-trip Time Estimation

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT ✍ unneeded retransmissions
  - High RTT ✍ poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- Spurious timeouts
  - “Conservation of packets” principle – more than a window worth of packets in flight

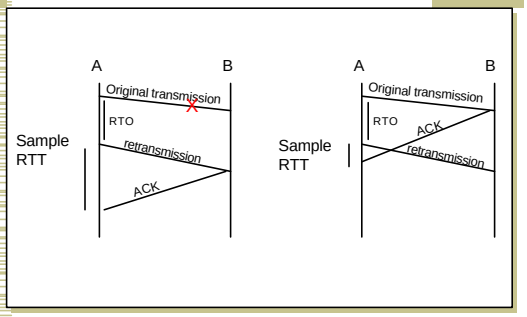
Initial Round-trip Estimator

- Round trip times exponentially averaged:
  - New RTT = ? (old RTT) + (1 - ?) (new sample)
  - Recommended value for ? : 0.8 - 0.9
    - 0.875 for most TCP's
- Retransmit timer set to ? RTT, where ? = 2
  - Every time timer expires, RTO exponentially backed-off
  - Like Ethernet
- Not good at preventing spurious timeouts

Jacobson's Retransmission Timeout

- Key observation:
  - At high loads round trip variance is high
- Solution:
  - Base RTO on RTT and standard deviation or RRTT
  - rttvar = ? * dev + (1- ?)rttvar
    - Dev = linear deviation
    - Inappropriately named – actually smoothed linear deviation

Retransmission Ambiguity



Karn's RTT Estimator

- Accounts for retransmission ambiguity
- If a segment has been retransmitted:
  - Don't count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
  - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

Timer Granularity

- Many TCP implementations set RTO in multiples of 200,500,1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient
- What happens for the first couple of packets?
  - Pick a very conservative value (seconds)

Delayed ACKS

- Problem:
  - In request/response programs, you send separate ACK and Data packets for each transaction
- Solution:
  - Don't ACK data immediately
  - Wait 200ms (must be less than 500ms – why?)
  - Must ACK every other packet
  - Must not delay duplicate ACKs

TCP ACK Generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
In-order segment arrival, No gaps, Everything else already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
In-order segment arrival, No gaps, One delayed ACK pending	Immediately send single cumulative ACK
Out-of-order segment arrival Higher-than-expect seq. # Gap detected	Send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that Partially or completely fills gap	Immediate ACK if segment starts at lower end of gap