15-441 15-641 Computer Networking

Lecture 12: Transport Protocols

Peter Steenkiste

Fall 2016

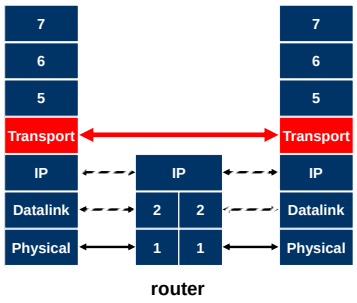
www.cs.cmu.edu/~prs/15-441-F16

Outline

- Transport introduction
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- Congestion control
- Transport optimization and futures

Transport Protocols

- Lowest level end-to-end protocol.
  - Header generated by sender is interpreted only by the destination
  - Routers view transport header as part of the payload



Functionality Split

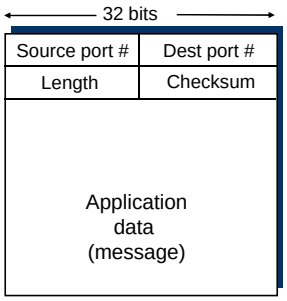
- Network provides best-effort delivery only
- End-systems must implement many functions
  - Demultiplexing
  - Error detection
  - Error recovery
  - In-order delivery
  - Message boundaries
  - Connection abstraction
  - Congestion control
  - …

UDP

TCP

UDP: User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- Demultiplexing based on ports
- Optional checksum
  - One's complement add (weak)
- That's it!
- So why do we need UDP?
  - No connections: no delay, state
    - Remember DNS?
  - No congestion control: can lead to unpredictable delays
    - Problem for multimedia, games, ..
  - Good starting point for other transport protocols
    - Implemented at application level

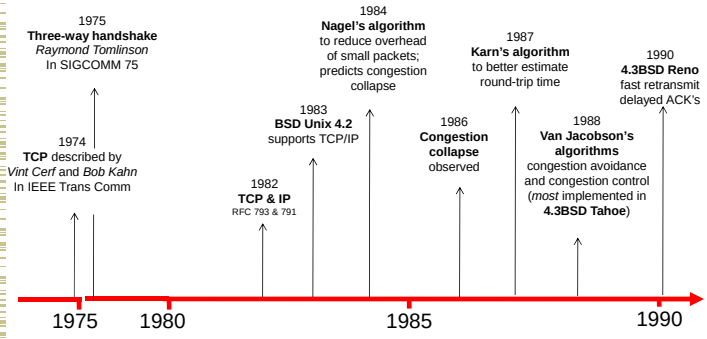


UDP segment format

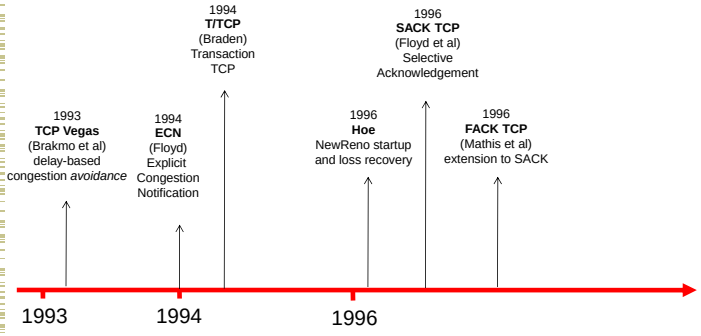
High-Level TCP Characteristics

- Protocol implemented entirely on endpoints
  - Fate sharing
- Protocol has evolved over time
  - Nearly impossible to change the header
  - Change processing at endpoints
  - Use options to add information to the header
  - Backward compatibility is what makes it TCP
- Most changes related to:
  - Faster networks, efficiency
  - Congestion control

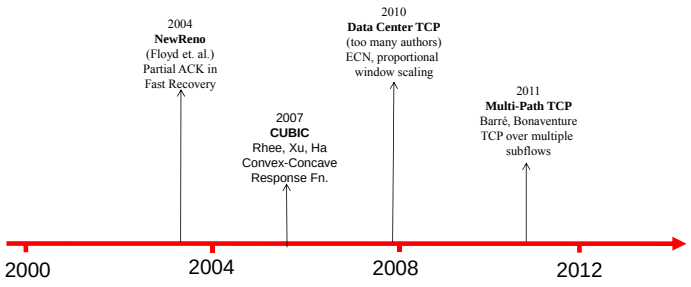
Evolution of TCP



TCP Through the 1990s



TCP Through the 2000s



TCP and its Header

- The cadillac of transport protocols
- Demultiplexing
- Connections
  - Sequence numbers
- Reliable
  - Acks, checksum
- Flow control
  - Window
- Congestion control
  - Nothing?
- Bookkeeping ++

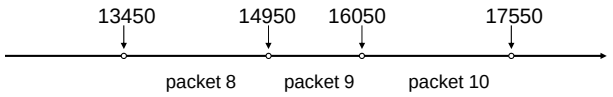
Source port	Destination port
Sequence number	
Acknowledgement	
HdrLen | 0 | Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

Outline

- Transport introduction
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- Congestion control
- Transport optimization and futures

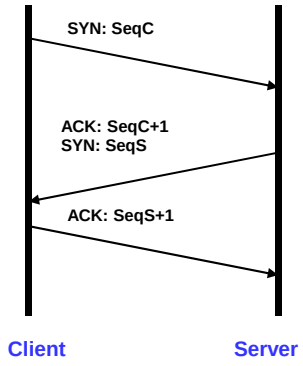
Sequence Number Space

- 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 into packets.
  - Packet size is limited to the Maximum Segment Size
- Each packet has a sequence number.
  - Indicates where it fits in the byte stream



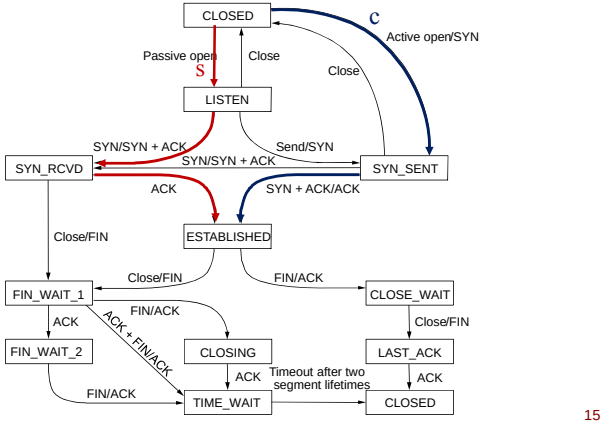
Establishing Connection: Three-Way handshake

- Each side notifies other of starting sequence number it will use for sending
  - Why not simply chose 0?
    - Must avoid overlap with earlier incarnation
    - Security issues
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- Can combine second SYN with first ACK



13

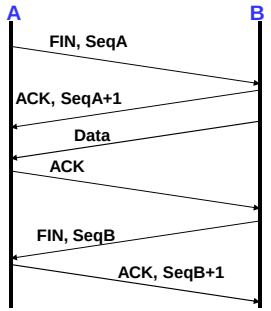
TCP State Diagram: Connection Setup



15

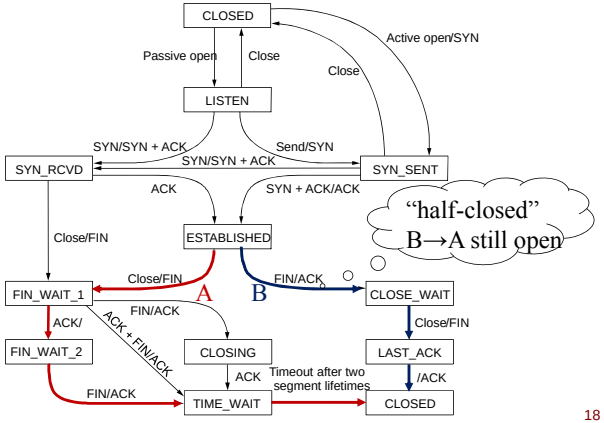
Tearing Down Connection

- Either side can initiate tear down
  - Send FIN signal
  - "I'm not going to send any more data"
- Other side can continue sending data
  - Half open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1



16

TCP State Diagram: Connection Teardown



18

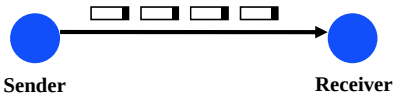
Outline

- Transport introduction
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- Congestion control
- Transport optimization and futures

19

A Naïve Protocol

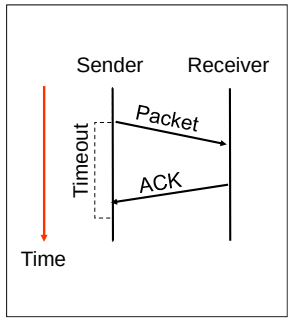
- Sender simply sends to the receiver whenever it has packets.
- Potential problem: sender can outrun the receiver.
  - Receiver too slow, runs out of buffer space, ..
- Even worse: packets can get lost!



20

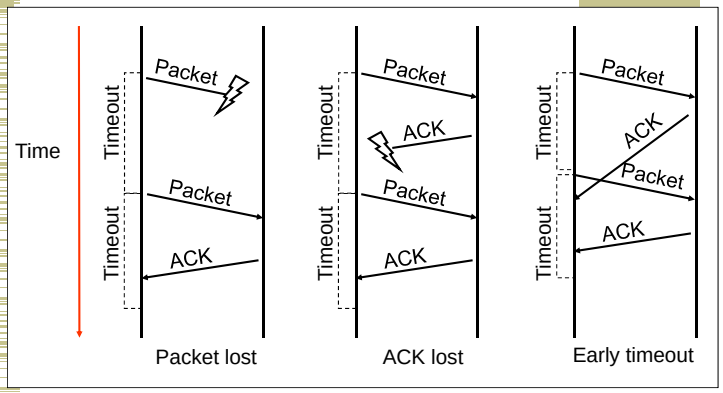
Stop and Wait

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



21

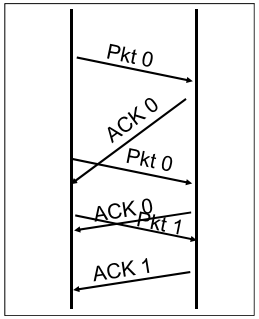
Recovering from Error



22

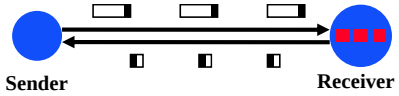
How to Recognize Retransmissions?

- 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

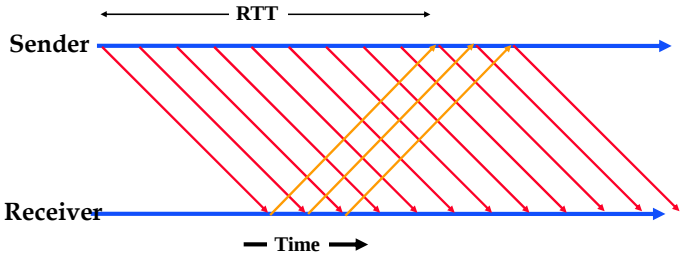


Window Flow Control

- Stop and wait flow control offers flow and error control, but it results in poor throughput for long-delay paths
- Solution: receiver provides sender with a window that it can fill with packets.
  - The window is backed up by buffer space on receiver
  - Receiver acknowledges the a packet every time a packet is consumed and a buffer is freed
  - Need larger sequence numbers: $W_s = 2^m - 1$ window for m bits

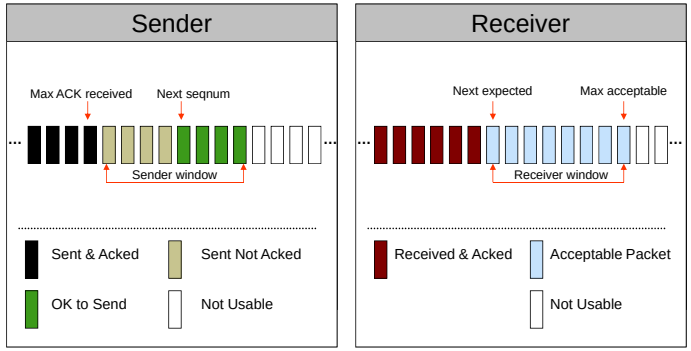


Bandwidth-Delay Product



$$\text{Max Throughput} = \frac{\text{Window Size}}{\text{Roundtrip Time}}$$

Sliding Window Sender/Receiver State



Window Sliding – Common Case

- On reception of new ACK
  - Increase max ACK received and send next packet
- On reception of new in-order data packet
  - Hand packet to application
  - Send an ACK that acknowledges the paper
  - Increase sequence of max acceptable packet
- But what do we do if packets are lost or reordered?
  - Results in a gap in the sequence of received packets
  - Raises two questions
    - What feedback does receiver give to the sender?
    - How and when does the sender retransmit packets

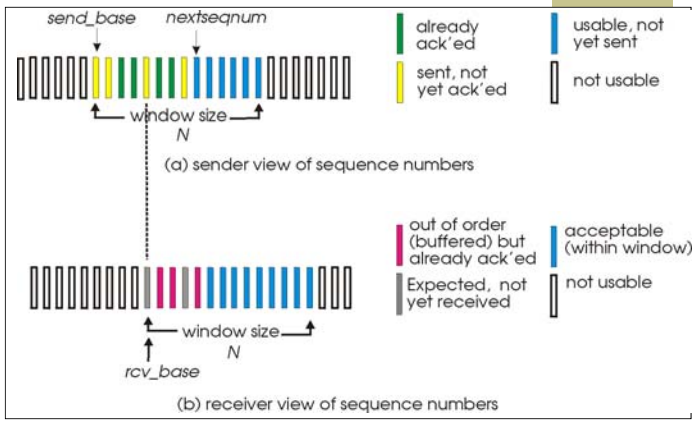
ACKing Strategies

- Per-packet ACKs acknowledge exactly one packet
  - Simple solution, but bookkeeping on sender is a bit messy
    - Must keep per packet state – not too bad
  - Inefficient: need ACK packet for every data packet
- Cumulative acks acknowledge all packets up to a specific sequence number
  - Maybe not as intuitive, but simple to implement
  - Stalls the pipe until lost packet is retransmitted and ACKed
- Negative ACKs allow a receiver to ask for a (presumed to be) lost packet
  - Avoids the delay of a timeout but is not sufficient!

Selective Repeat Retransmissions

- Simple retransmission strategy for when receiver acknowledges correctly received packets *individually*
  - If packets out of order, receiver cannot hand data to application so window does not move forward
- Sender only resends packets for which ACK not received
  - Sender timer for individual unACKed packet
- Sender window calculation
  - Buffer space used at receiver consists of N consecutive seq #'s
  - Starts with an earliest unacknowledged packet
    - Some packets in the window may have been acknowledged but packet could not be given to application

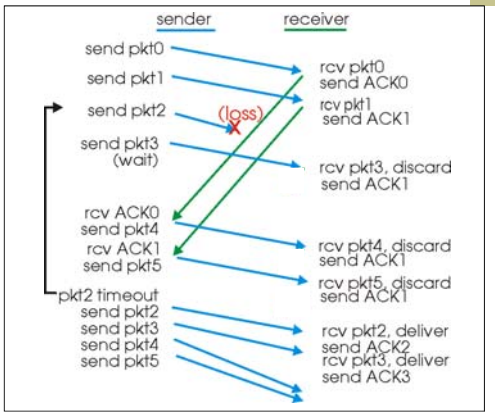
Selective Repeat: Sender, Receiver Windows



Go-Back-N Recovery

- Strategy when receiver sends cumulative ACKs
  - Send nothing for out of order packet – sender will timeout
  - Otherwise sends cumulative ACK
- Sender implements Go-Back-N recovery
  - Set timer upon transmission of packet
  - Retransmit all unacknowledged packets upon timeout
- Performance during loss recovery
  - Single loss can result in many packet retransmissions
  - Timeouts are expensive – add significant delay
  - Puts emphasis on simplicity of implementation
    - E.g., receiver can drop non-contiguous packets (resent anyway)

Go-Back-N in Action



Window size = 4

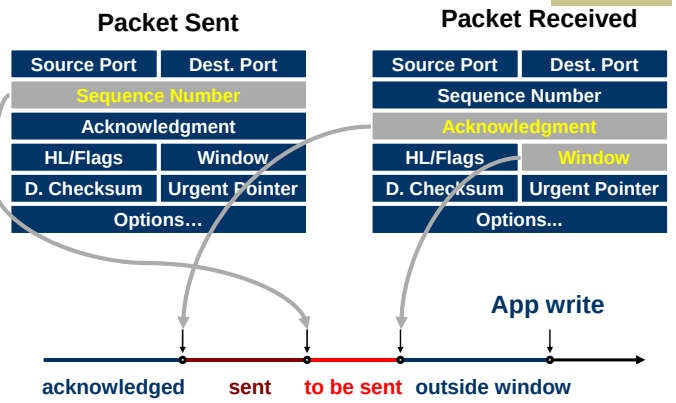
Outline

- Transport introduction
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- Congestion control
- Transport optimization and futures

TCP = Go-Back-N Variant

- TCP uses a sliding window with cumulative acks
  - Receiver can only return a single "ack" sequence number
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
- But: sender only retransmits one packet after timeout
  - Reason: only knows that that specific packet is lost
    - Network is congested → should not overload it with questionable retransmits
- Receiver stores out of order packets
  - Can be used after the sender "fills the gap"
- Error control is based on byte sequences, not packets.
  - Retransmissions can be different from lost packets – Why?

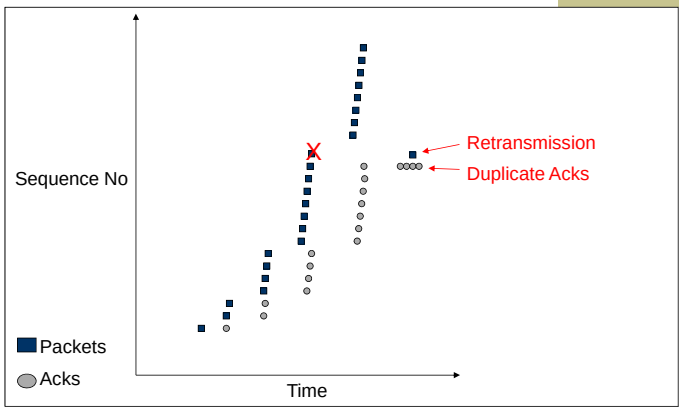
Window Flow Control: Send Side



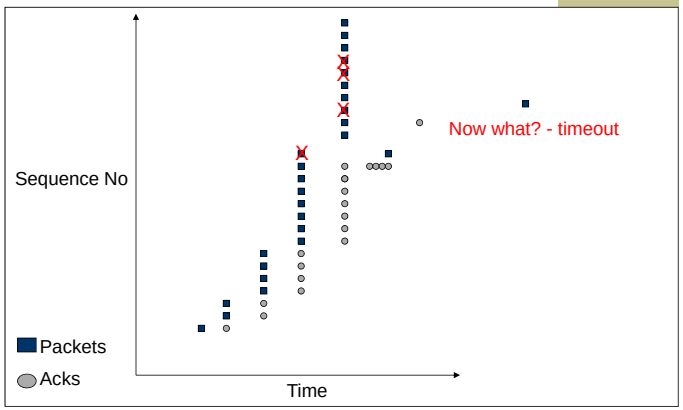
Duplicate ACKs (Fast Retransmit)

- Basic Go-Back-N incurs timeout for every loss
  - Can we do better? How about a NACK?
- Receiver sends "duplicate ack" for out of order packets
  - Repeated acks for the same sequence
  - Serves as a NACK – no room in header for real NACK!
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering – oops! Unnecessary retransmit
- Solution - assume re-ordering is infrequent :
  - Receipt of 3 or more duplicate acks is indication of loss
  - Sender does not wait for timeout to retransmit packet
  - When does this fail?

Duplicate ACKs (Fast Retransmit)



How about Multiple Losses?

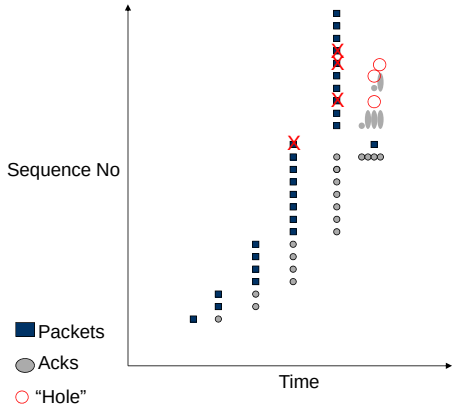


SACK

- Cumulative acks provide little information
  - How many packets were really lost?
  - Becomes a problem as windows get bigger
- Selective acknowledgement (SACK) essentially adds a bitmask of packets received
  - Implemented as a TCP option
  - Encoded as a set of received byte ranges (max of 4 ranges/often max of 3)
- When to retransmit?
  - Still need to deal with reordering → wait for out of order by 3pkts

39

Selective ACK (SACK)



40

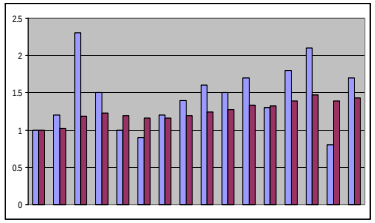
Round-trip Time Estimation

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT estimate: unneeded retransmissions
  - High RTT estimate: poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- So how do we estimate RTT?

41

Original TCP Round-trip Estimator

- Round trip times exponentially averaged:
  - New RTT = α (old RTT) + (1 - α) (new sample)
  - Recommended value for α: 0.8 - 0.9
    - 0.875 for most TCP's



- Retransmit timer set to (b * RTT), where b = 2
  - Every time timer expires, RTO exponentially backed-off
- Not good at preventing spurious timeouts
  - Why?

42

Jacobson's Retransmission Timeout

- Key observation:
  - At high loads, round trip variance is high
- Solution:
  - Base RTO on RTT and standard deviation
    - RTO = RTT + 4 * rttvar
  - new_rttvar = β * dev + (1- β) old_rttvar
    - Dev = linear deviation
    - Inappropriately named – actually smoothed linear deviation
- In practice: TOs use coarse clock, e.g., 100s of msec

Important Lessons

- Transport service
  - UDP → mostly just IP service + demultiplexing
  - TCP → congestion controlled, reliable, byte stream
- Types of ARQ protocols
  - Sliding window for high throughput
  - Go-back-n → can keep link utilized (except w/ losses)
  - Selective repeat → efficient loss recovery
- TCP uses go-back-n variant
  - Avoid unnecessary retransmission ..
  - … and gaps in the flow (fast retransmit/recovery, SACK)