15-441 Computer Networking

Transport Protocols

Peter Steenkiste

Fall 2015

www.cs.cmu.edu/~prs/15-441-F15

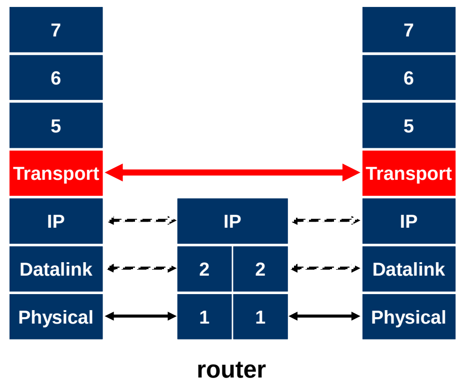
Outline

- Transport introduction
- Error recovery and flow control
- Congestion control
- Transport optimization and futures

2

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

32 bits

Source port #	Dest port #
Length	Checksum
Application data (message)	

UDP segment format

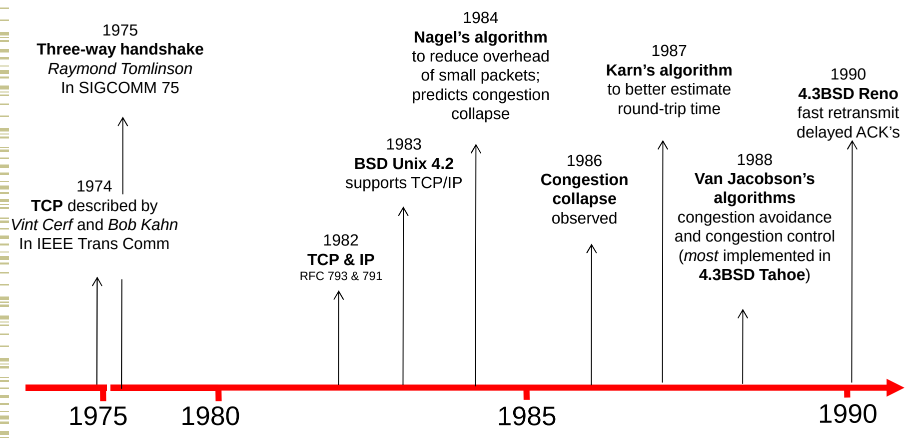
5

High-Level TCP Characteristics

- Protocol implemented entirely at the ends
  - 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
    - These do change sometimes
  - Backward compatibility is what makes it TCP
- Most changes related to:
  - Faster networks, efficiency
  - Congestion control

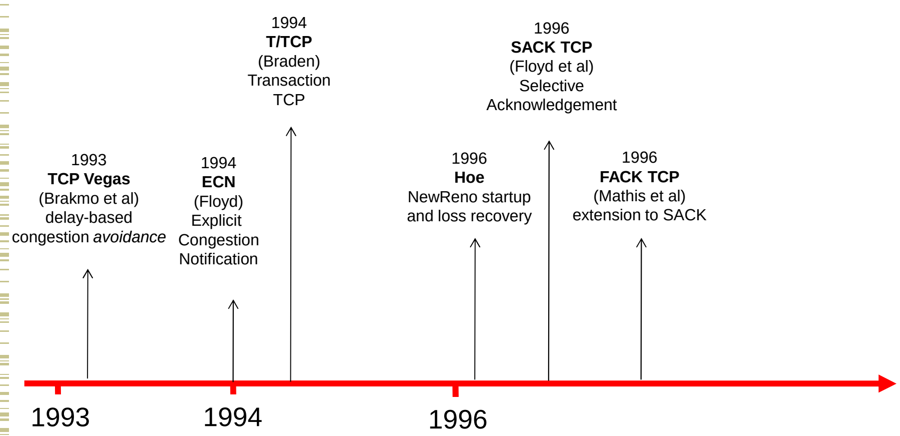
6

Evolution of TCP



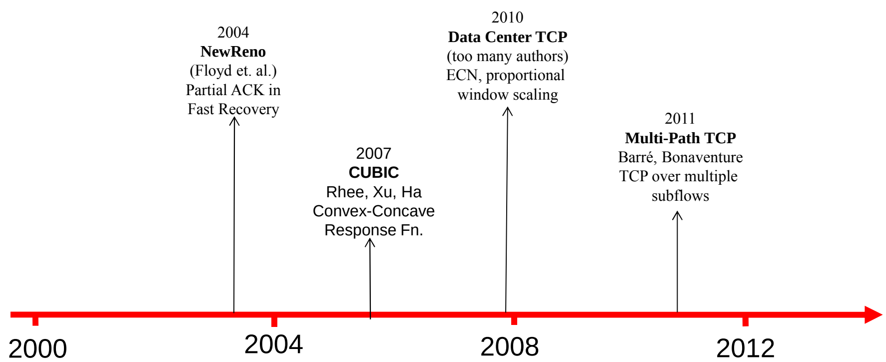
7

TCP Through the 1990s



8

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 ++

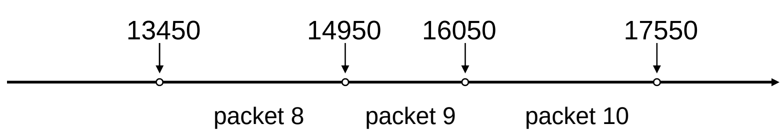
Source port		Destination port	
Sequence number			
Acknowledgement			
HdrLen	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			
Data			

Outline

- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

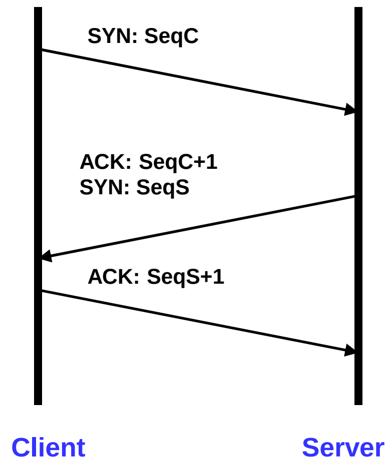
Sequence Number Space

- 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

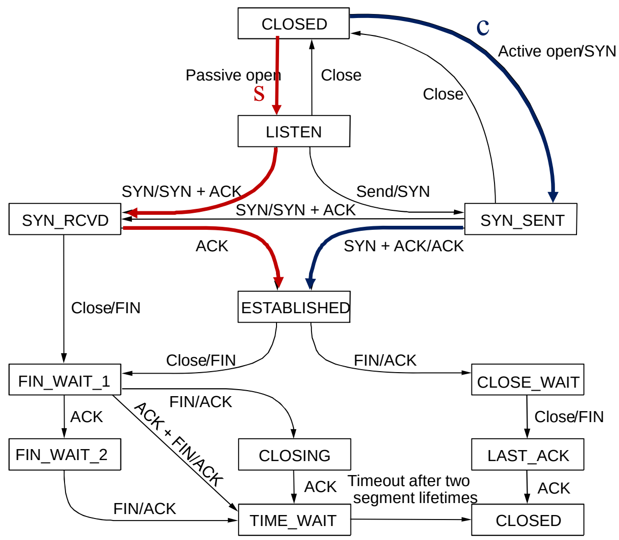


TCP Connection Setup Example

```
09:23:33.042318 IP 128.2.222.198.3123 > 192.216.219.96.80:
S 4019802004:4019802004(0) win 65535
<mss 1260,nop,nop,sackOK> (DF)

09:23:33.118329 IP 192.216.219.96.80 > 128.2.222.198.3123:
S 3428951569:3428951569(0) ack 4019802005 win 5840
<mss 1460,nop,nop,sackOK> (DF)

09:23:33.118405 IP 128.2.222.198.3123 > 192.216.219.96.80:
. ack 3428951570 win 65535 (DF)
```

- Client SYN
  - SeqC: Seq. #4019802004, window 65535, max. seg. 1260
- Server SYN-ACK+SYN
  - Receive: #4019802005 (= SeqC+1)
  - SeqS: Seq. #3428951569, window 5840, max. seg. 1460
- Client SYN-ACK
  - Receive: #3428951570 (= SeqS+1)

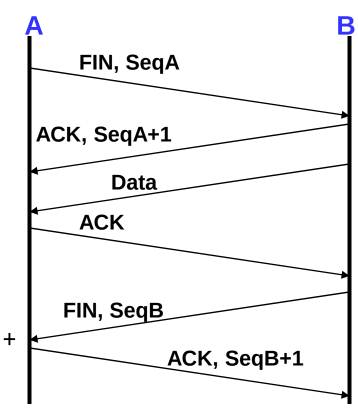
TCP State Diagram: Connection Setup



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

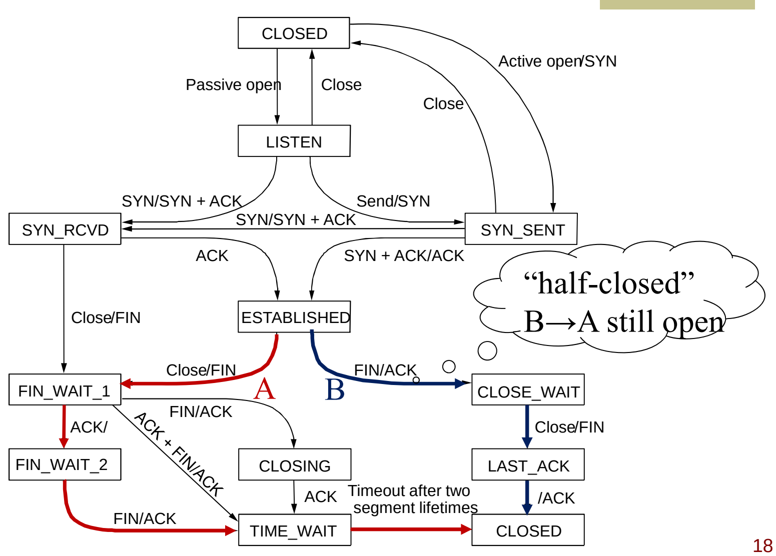
TCP Connection Teardown Example

```
09:54:17.585396 IP 128.2.222.198.4474 > 128.2.210.194.6616:
F 1489294581:1489294581(0) ack 1909787689 win 65434 (DF)

09:54:17.585732 IP 128.2.210.194.6616 > 128.2.222.198.4474:
F 1909787689:1909787689(0) ack 1489294582 win 5840 (DF)

09:54:17.585764 IP 128.2.222.198.4474 > 128.2.210.194.6616:
. ack 1909787690 win 65434 (DF)
```

- Session
  - Echo client on 128.2.222.198, server on 128.2.210.194
- Client FIN
  - SeqC: 1489294581
- Server ACK + FIN
  - Ack: 1489294582 (= SeqC+1)
  - SeqS: 1909787689
- Client ACK
  - Ack: 1909787690 (= SeqS+1)

17

TCP State Diagram: Connection Teardown



18

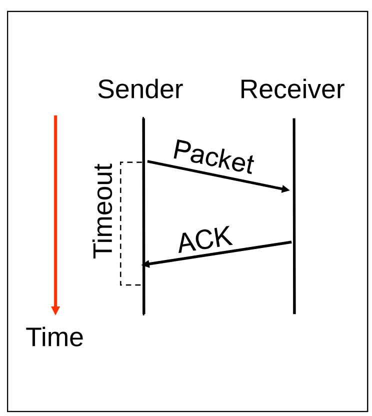
Outline

- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

19

Review: Stop and Wait

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



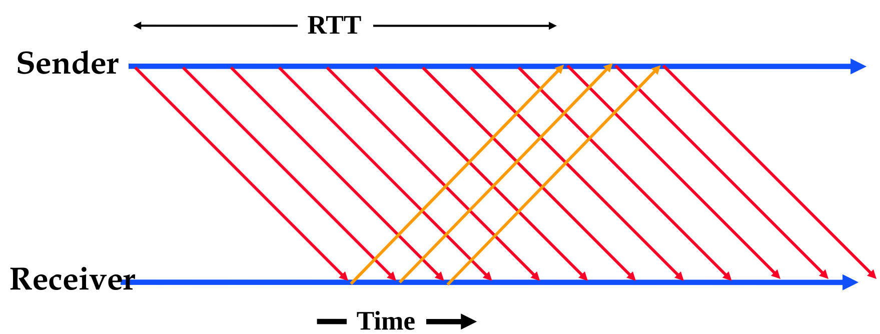
20

Problems with Stop and Wait

- Stop and wait offers provides flow and error control, but ..
- How do we overcome the limitation of one packet per roundtrip time: Sliding window.
  - Receiver advertises a “window” of buffer space
  - Sender can fill the window -> fills the “pipe”
- How do we distinguish new and duplicate packets: Sequence numbers
  - 1 bit enough for stop and wait
  - More bits for larger windows (see datalink lecture)

21

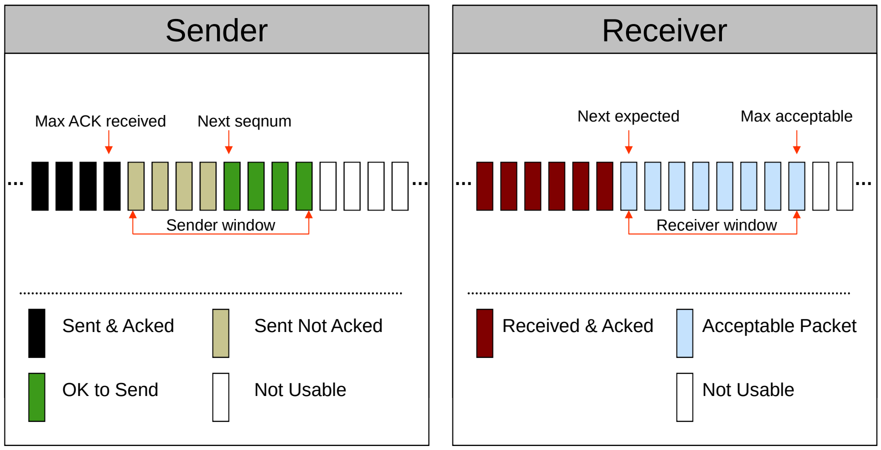
Bandwidth-Delay Product



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

22

Sliding Window Sender/Receiver State



23

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 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, and how?
    - How and when does the sender retransmit packets

24

ACKing Strategies

- 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 packet
  - 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 packet that is (presumed to be) lost
  - Avoids the delay associated with a timeout

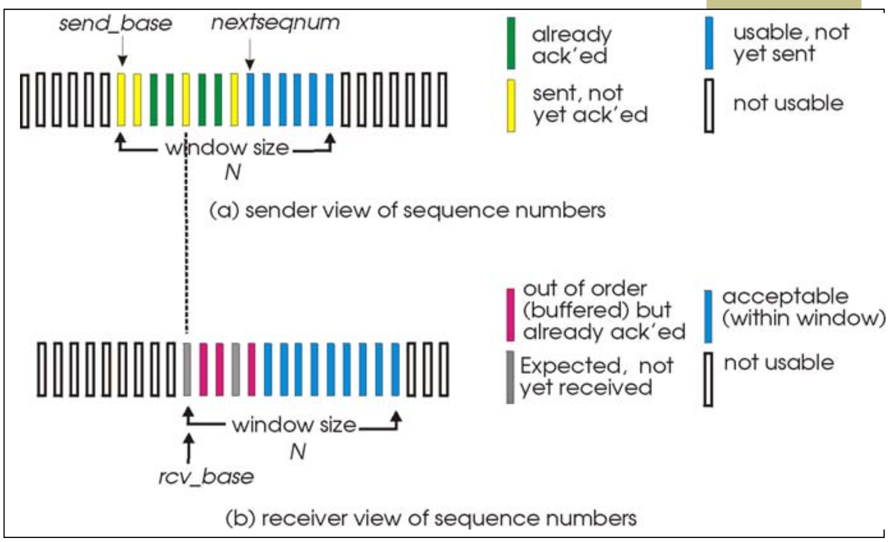
25

Selective Repeat

- Receiver *individually* acknowledges correctly received packets
  - 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
  - N consecutive seq #'s
  - Starts with an earliest unacknowledged packet
    - Some packets in the window may have been acknowledged

26

Selective Repeat: Sender, Receiver Windows



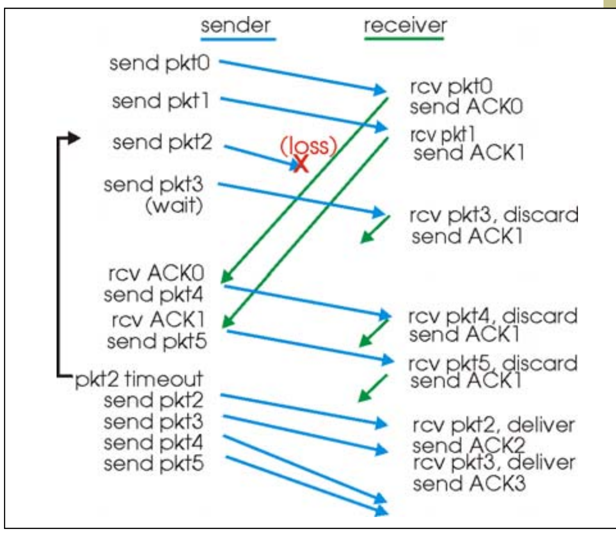
27

Go-Back-N Recovery

- Receiver sends cumulative ACKs
  - When out of order packet - send nothing (wait for sender to 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
  - Puts emphasis on simplicity of implementation
    - E.g., receiver can drop non-contiguous packets (resent anyway)

28

Go-Back-N in Action



Window size = 4

29

Outline

- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

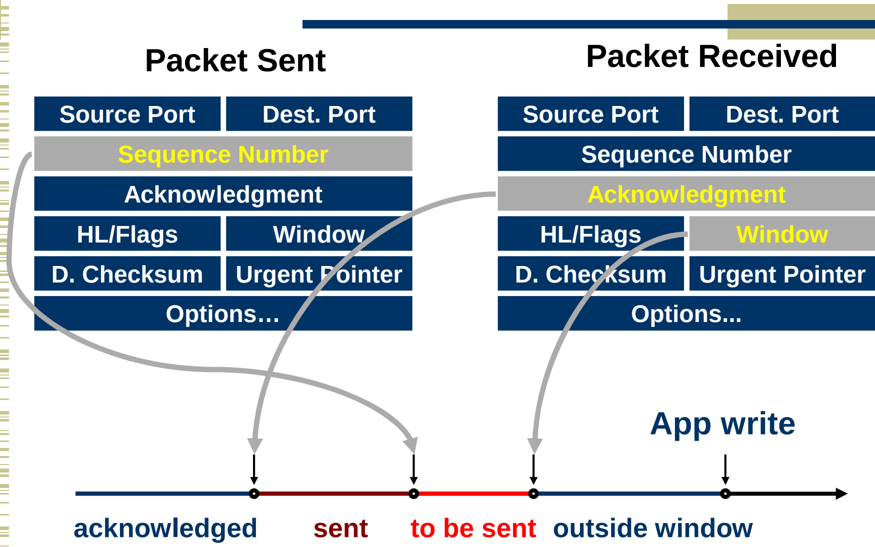
30

TCP = Go-Back-N Variant

- Sliding window with cumulative acks
  - 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 after timeout.
  - Reason???
    - Only one that that specific packet is lost
    - Network is congested → shouldn't overload it with questionable retransmits
- Receiver stores out of order packets
  - Can we used after the sender "fills the gap"
- Error control is based on byte sequences, not packets.
  - Retransmitted packet can be different from the original lost packet – Why?

31

Window Flow Control: Send Side



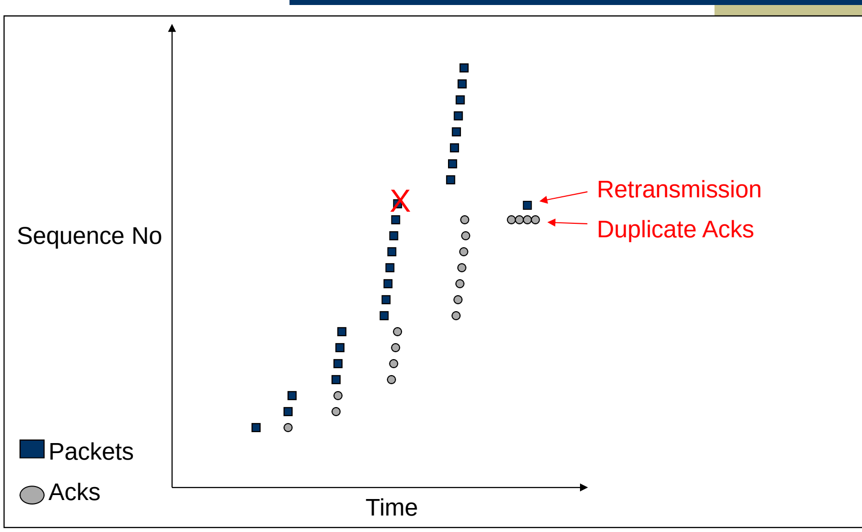
32

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?

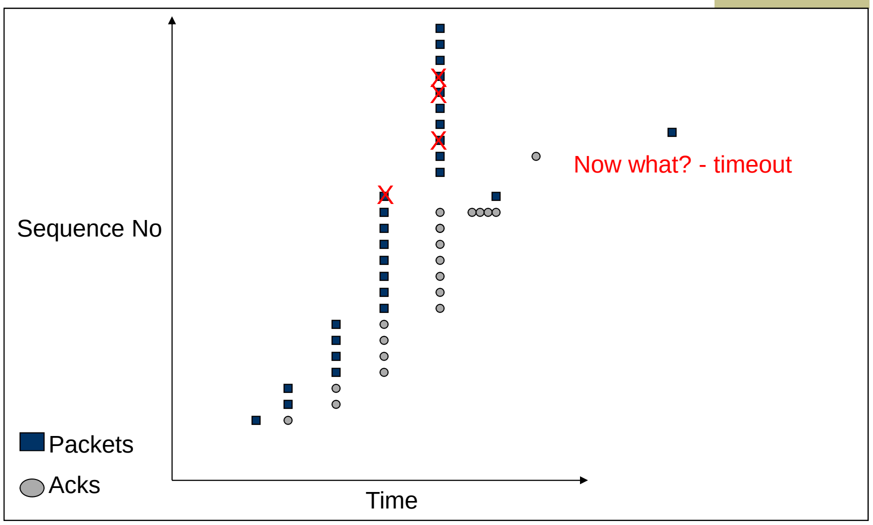
33

Duplicate ACKs (Fast Retransmit)



34

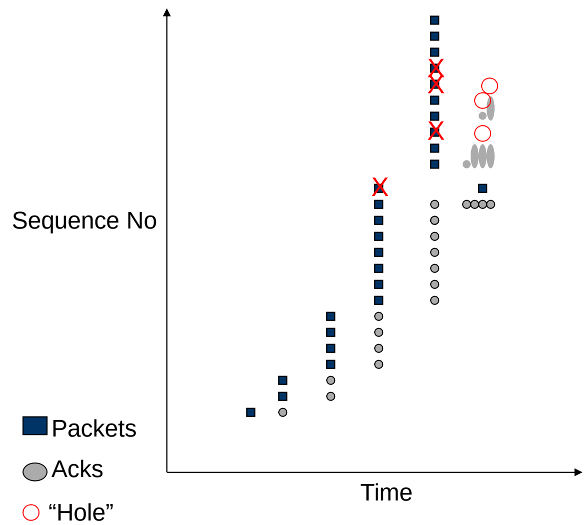
How about Multiple Losses?



SACK

- Basic problem is that cumulative acks provide little information
- 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

Selective ACK (SACK)



Outline

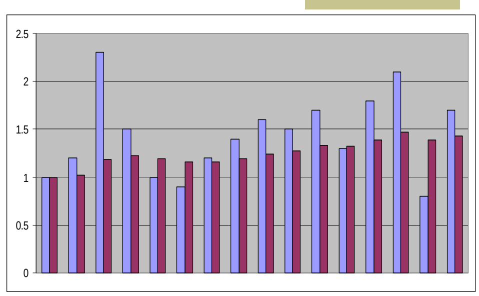
- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

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?

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?

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)