The Transport Layer: Transport Services & Reliable Data Transfer

Smith College, CSC 249
February 26, 2008

slides mostly from J.F Kurose and K.W. Ross, copyright 1996-2007

Chapter 3: Transport Layer

Our goals:

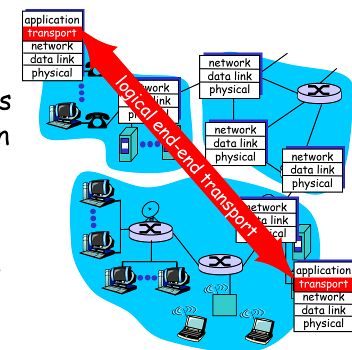
- To understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- To learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Transport services and protocols

- To provide *logical communication* between application processes running on different hosts
- Transport protocols run in end systems
  - sending side: breaks messages into segments, passes to network layer
  - receiving side: reassembles segments into messages, passes to application layer



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

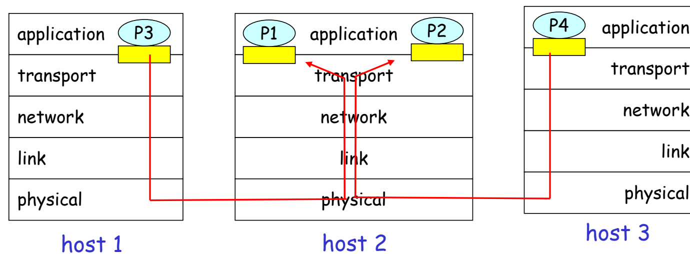
Multiplexing/demultiplexing

- Multiplexer
  - Selects input from one of many input lines and directs the information to a single output line
  - Many sockets gathered into one network connection
- Demultiplexer
  - Direct a single input to one of many possible output lines
  - Single network connection distributed to many sockets (processes)

Multiplexing/demultiplexing

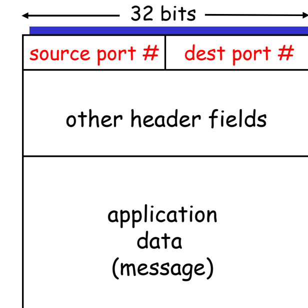
Demultiplexing at rcv host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- The transport service at each host uses the port numbers & perhaps the IP addresses (TCP) to direct a segment to the appropriate socket
- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number



TCP/UDP segment format

Comparing demultiplexing

- A UDP socket is identified by a 2-tuple
  - The destination IP & port numbers
  - The receiving host directs the UDP segment to the socket with the specified port number
  - Note that IP datagrams with different source IP addresses and/or source port numbers will be directed to the same UDP socket
- A TCP socket is identified by 4-tuple:
  - source IP address, source port number
  - destination IP address, destination port number
  - The receiving host uses all four values to direct the segment to the appropriate socket → The 'virtual pipe' connected between the sending and receiving hosts

UDP Socket & Segment

- Create sockets with or w/o port numbers:

```
DatagramSocket mySocket1 = new
    DatagramSocket();
DatagramSocket mySocket2 = new
    DatagramSocket(99222);
```

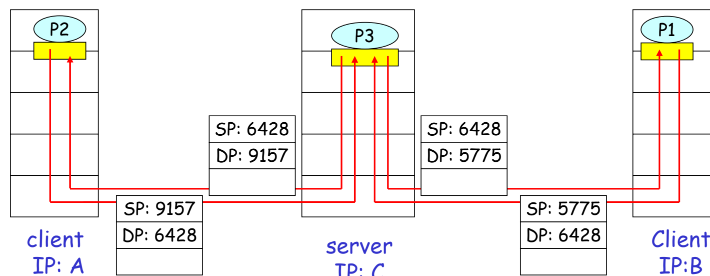
- UDP socket identified by two-tuple:

  (dest IP address, dest port number)

- UDP segment includes data, source port and destination port (+ length & checksum)

UDP Connectionless demux

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

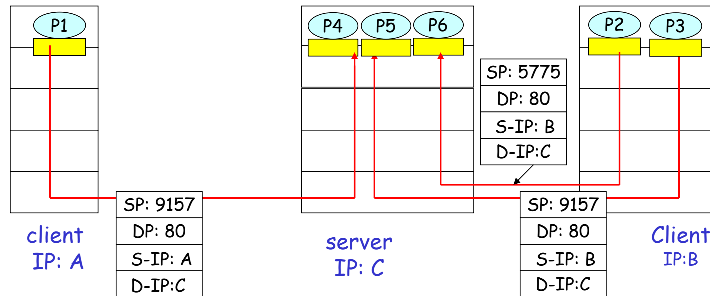


SP provides "return address"

TCP Connection-oriented demux

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- In what situation will all TCP segments with different source IP addr and port #s nonetheless go to the same port?
  - If they are TCP connection establishment requests
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

TCP Connection-oriented demux



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

Why is there a UDP?

- It is simple: no connection state at sender, receiver
- It is fast(er):
  - no connection establishment (which can add delay)
  - small segment header
  - no congestion control: UDP can blast away as fast as desired

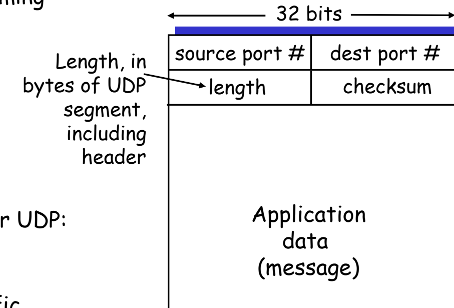
But

- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order

UDP: Segment Format

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

Length, in bytes of UDP segment, including header



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: 1's complement of the sum of (16-bit) segment contents
- sender puts checksum value into UDP checksum field

Receiver:
- compute checksum of received segment – including the sender's checksum 16-bit word in the sum
- If receiver's sum is all '1's then there were no errors (probably)
  - If a bit is 0 then the packet has errors

Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

UDP checksum example

Chapter Problem
- Why does UDP take the 1s complement of the sum?
  - Why not simply use the sum?
- With the 1s complement scheme, how does the receiver detect errors?
- Is it possible for a 1-bit error to go undetected?
- Is it possible for a 2-bit error to go undetected?

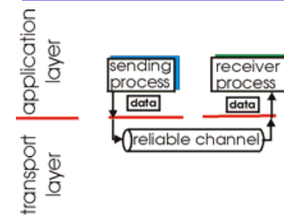
UDP checksum example

- To detect errors, the receiver adds all the original data words plus the checksum. If the sum contains a zero, the receiver knows there has been an error.
- All one-bit errors will be detected, but two-bit errors can be undetected (e.g., if the last digit of the first word is converted to a 0 and the last digit of the second word is converted to a 1).

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Reliable data transfer: getting started



(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable Data Transfer

- Transport service abstraction:
  - Must provide mux and demux service
- What can go wrong?
  - Bit errors
    - original data as well as acknowledgment messages
  - Lossy channel (with bit errors)
    - Stop-and-wait v. pipelining
  - Out-of-order packets

Principles of Reliable Data Transfer

- Transport service abstraction:
  - Must provide mux and demux service
- Reliable Data Transfer properties
  - Provide a reliable channel …
  - Checksum
  - Detect errors & packet loss, and respond
    - ACKnowledgment packets
    - Retransmission
    - Sequence numbers

Reliable data transfer

- Incrementally develop sender and receiver sides of a generic reliable data transfer protocol (rdt)...

Reliable data transfer: rdt 1.0

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Sender:
  - Receive the data from an application
  - Make the packet and send it on the unreliable channel
- Receiver:
  - Receive the packet
  - Extract and deliver the data to the application

Reliable data transfer: rdt 2.0

- Allow for possible bit errors
  - How can we recover from bit errors?
- Three additional protocol capabilities
  - Error detection: checksum
  - Receiver feedback: ACK and NAK packets
  - Retransmission: the sender will retransmit a packet that had errors by the time it was received

rdt2.0 has a fatal flaw!

What happens if ACK/NAK is corrupted?

- sender does not know what happened at the receiver
- cannot just retransmit: possible duplicate

Handling duplicates:

- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

Sender sends one packet, then waits for receiver response

Reliable data transfer: rdt 2.0

- Shortcomings
  - What about corrupted ACKs or NAKs?
    - Receiver must be able to tell if the packet it is receiving is new data or a retransmission…
  - A 'stop-and-wait' protocol: sender waits for ACK before sending next packet
  - Solution … leading to rdt 2.1
    - Include sequence numbers in the data packets
    - This allows the receiver to know if the packet is a retransmission

rdt2.1: Include Sequence Numbers

Sender:

- Sequence numbers added to packets
- Must check if the received ACK/NAK is corrupted

Receiver:

- Must check if received packet is a duplicate
  - state indicates the next expected packet sequence number
- Note: receiver can *not* know if its last ACK/NAK was received correctly by the sender

rdt2.1: Discussion Question 1

- Suppose Host A sends two TCP segments back to back to Host B over a TCP connection. The first segment has sequence number 90; the second has sequence number 110
  - How much data is in the first segment?
  - Suppose that the first segment is lost but the second segment arrives at B. In the acknowledgment that Host B sends to Host A, what will be the acknowledgment number?
- a) 20 bytes
- b) ACK number = 90

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, but using ACKs only
- instead of NAK, receiver sends ACK for last packet received without errors
  - receiver must *explicitly* include the sequence number of the packet being ACKed
- duplicate ACK at sender results in same action as a NAK in rdt 2.1: *retransmit current packet*

rdt3.0: channels with errors *and* loss

New assumption: underlying channel can also lose packets (data or ACKs)

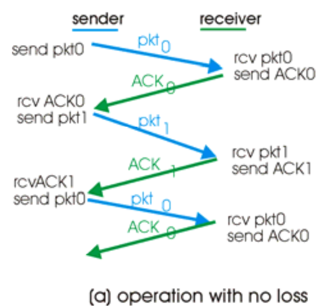
- checksum, sequence numbers, ACKs, retransmissions will be of help, but not enough

rdt3.0: channels with errors *and* loss

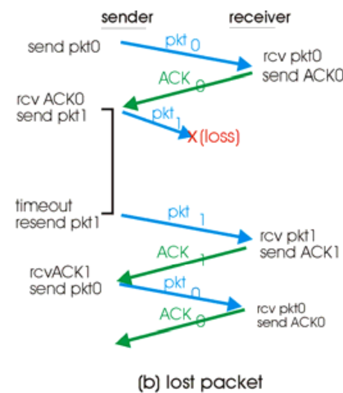
Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- Is there a problem if the packet (or ACK) is just delayed, not lost?
  - retransmission will be duplicate, but use of sequence numbers already handles this
  - receiver must specify sequence number of packet being ACKed
- New element: This requires a …?

rdt3.0 in action

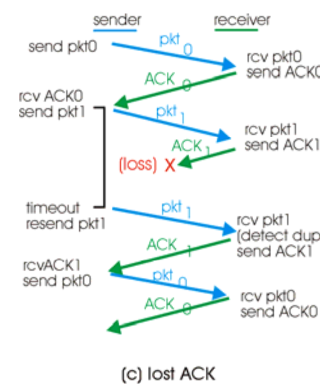


(a) operation with no loss

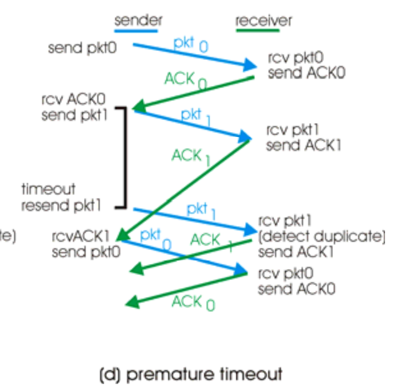


(b) lost packet

rdt3.0 in action



(c) lost ACK

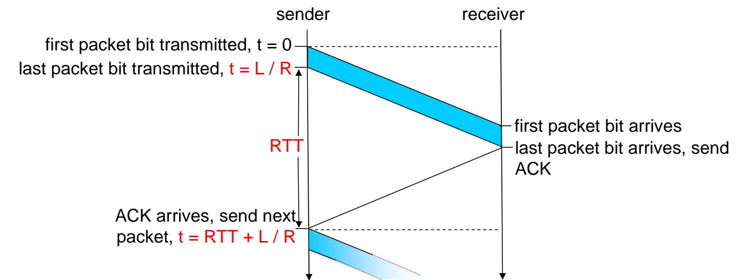


(d) premature timeout

Performance of rdt3.0: "Utilization"

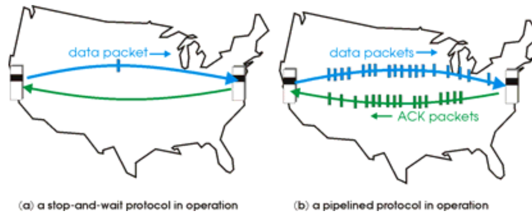
- rdt3.0 works, but performance is poor
- As an example (in text), a 1Gbps link may be constrained to a throughput of only 267kbps!
- Why?
  - This simple protocol is a "stop-and-wait" protocol, *i.e.,*…
  - Send packet, wait for confirmation before sending next packet
  - The "wait" includes processing at both ends, transmission and propagation delays

rdt3.0: stop-and-wait operation



Solution: Pipelining

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: (1) *go-Back-N,* (2) *selective repeat*

Pipelining Protocols

Go-back-N: big picture:
- Sender can have up to N unACKed packets in pipeline
- Rcvr only sends cumulative ACKs
  - Doesn't ACK packet if there's a gap
- Sender has timer for oldest unACKed packet
  - If timer expires, retransmit all unACKed packets

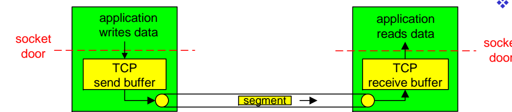
Selective Repeat: big pic
- Sender can have up to N unACKed packets in pipeline
- Rcvr ACKs individual packets
- Sender maintains timer for each unACKed packet
  - When timer expires, retransmit only unACKed packet

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP: Overview

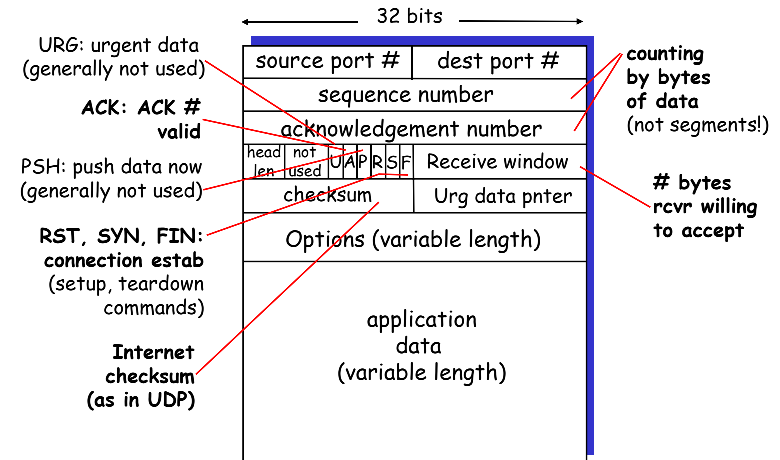
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam*
- pipelined:
  - TCP congestion and flow control set window size
- Segment size:
  - MSS: maximum segment size
- connection-oriented:
  - Handshaking (exchange of control msgs)
- flow controlled:
  - Sender will not overwhelm receiver
  - Send & receive buffers



TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service ?
  - Error detection
  - Retransmissions
  - Detect packet loss
  - Sequence and acknowledgment numbers
    - Random initial number
  - Cumulative acknowledgments
  - Timer – single countdown timer
  - Pipelining
  - Flow control
  - Congestion control
- Retransmissions are triggered by ?
  - timeout events
  - duplicate ACKs

TCP segment structure



TCP Sequence and ACK Numbers

- TCP uses byte streams
  - Recall our socket programming examples
  - Input and output BYTE streams attached to sockets
- Implications
  - Sequence numbers and ACK numbers refer to the number of BYTES not the number of segments
- From TCPClient.java
  - outToServer.writeBytes(sentence + '\n');

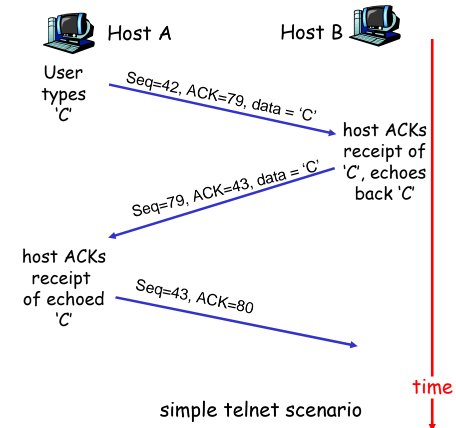
45

TCP Sequence and ACK Numbers

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how does receiver handle out-of-order segments?
- A: TCP spec does not specify - up to implementor



46

TCP Sequence and ACK Numbers

Seq. #'s Question:
- A few seconds after the user types the letter 'C' the user types the letter 'R.' After typing the letter 'R' how many segments are sent, and what is put in the sequence number and acknowledgment fields of the segments?

47

Setting the TCP Timer

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?
- `SampleRTT`: measured time from segment transmission until ACK receipt
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current `SampleRTT`
  - Weighted moving average

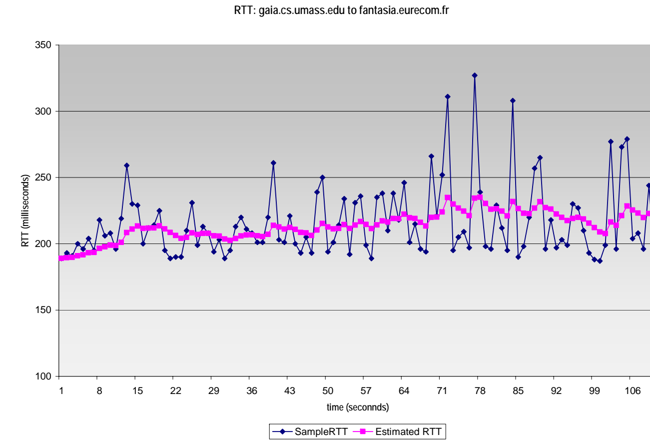
48

TCP Round Trip Time and Timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

Example RTT estimation:



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP sender events:

data received from application:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

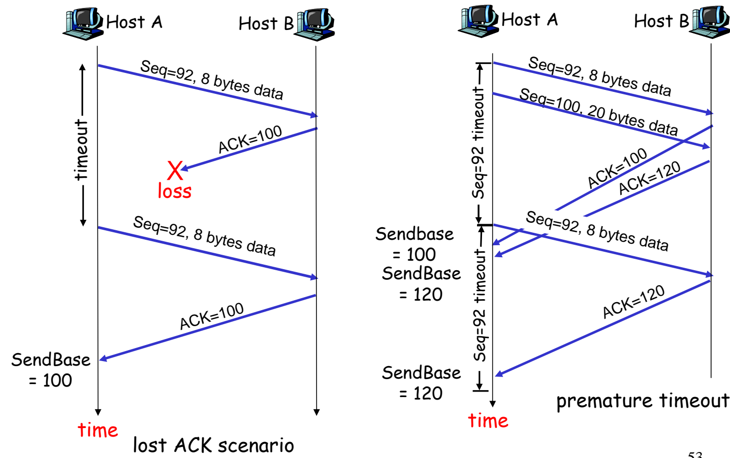
timeout:

- retransmit segment that caused timeout
- restart timer

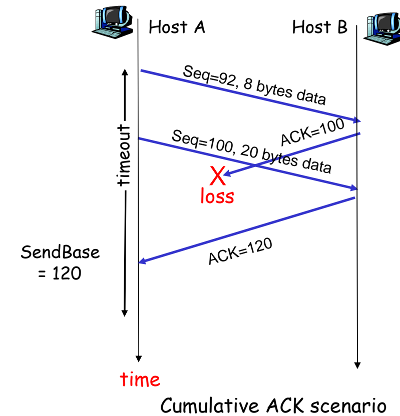
ACK received:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP: retransmission scenarios



TCP retransmission scenarios (more)



Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives **3 ACKs** for the same data, it decides that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Why Wait for 3 Duplicate ACKs?

- Why did the TCP designers chose to have TCP wait until it has received **three** duplicate ACKs before performing a fast retransmit, rather than performing a fast retransmit after the first duplicate ACK for a segment is received?

Why Wait for 3 Duplicate ACKs?

- **Suppose packets n, n+1, and n+2 are sent, and that packet n is received and ACKed.**
- **If the next packets are received in the order n+2, n+1, then the receipt of packet n+2 will generate a duplicate ACK for n**
  - **→ and would trigger a retransmission with the policy of only one duplicate ACK.**
- **By waiting for a *triple* duplicate ACK, we require that *two* packets after packet n are correctly received, while n+1 was not received.**
- **This scheme – waiting for two packets (rather than 1) is a tradeoff between triggering a quick retransmission when needed, but not retransmitting prematurely in the face of packet reordering.**

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control