Chapter 3: Transport Layer

Our goals:

- understand principles behind transport layer services:
  - multiplexing/ demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport, unreliable delivery of segments
  - TCP: connection-oriented transport, reliable delivery of byte stream

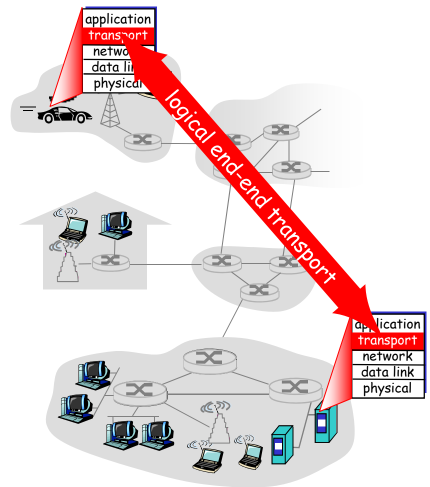
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer

  (my slides for Section 3.4 do not follow Kurose & Ross)
- 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

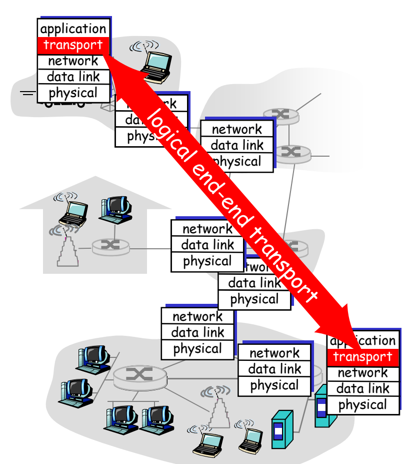
Transport services and protocols

- provide *logical communication* between app processes on different hosts
- transport protocol runs in end systems (primarily)
  - send side: breaks app messages into ***segments***, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer



Internet transport-layer protocols

- unreliable, unordered datagram delivery by **UDP**
  - no-frills extension of "best-effort" IP
- reliable, in-order byte delivery by **TCP**
  - connection setup
  - flow control
  - congestion control
- services not available:
  - delay guarantees
  - bandwidth guarantees



Chapter 3 outline

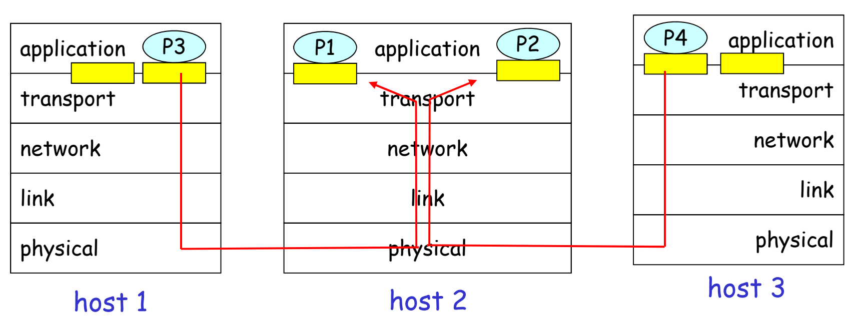
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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

Multiplexing/demultiplexing

Demultiplexing at rcv host:
deliver received segments to correct sockets

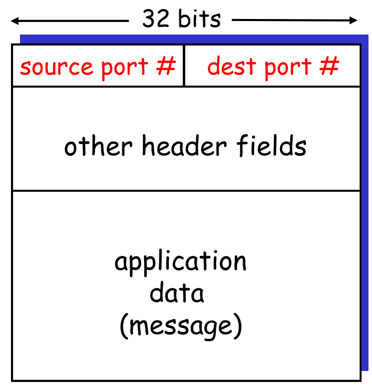
Multiplexing at send host:
gather data from multiple sockets, encapsulate data with header (later used for demultiplexing)

socket process/thread



How demultiplexing works

- host receives IP datagrams
- It uses IP addresses in layer-3 header & port numbers in layer-4 header to direct segment to appropriate socket



TCP/UDP segment format

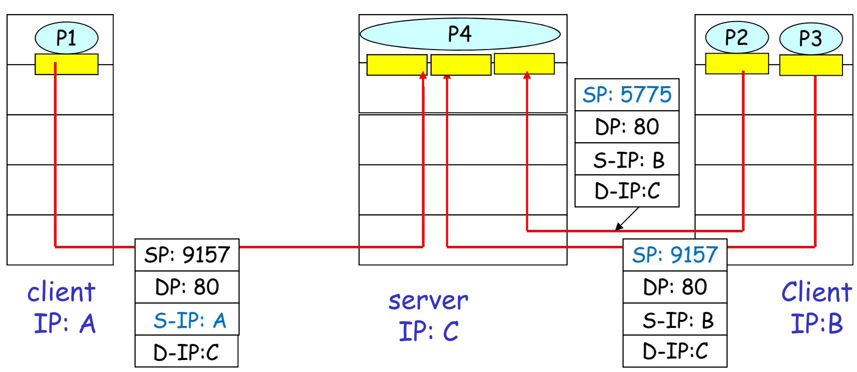
Connectionless demultiplexing

- UDP socket identified by two-tuple:
  (dest IP address, dest port number)
- When host receives UDP segment:
  - directs UDP segment to socket with destination port number
- IP datagrams from *different* sources directed to *same* UDP socket

Connection-oriented demux

- Server has welcome and connection sockets
  - welcome socket is identified by server's IP address and a port number
- TCP connection socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Server may support many simultaneous TCP connection sockets with clients:
  - each connection socket and the welcome socket have the *same port* number in server host
  - receiving host uses *all four values* to direct segment to appropriate connection socket

Connection-oriented demux (cont)

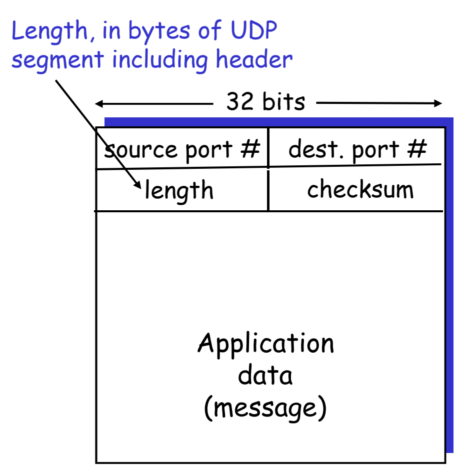


Chapter 3 outline

- 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]

- "best effort" service, UDP segments (aka datagrams) may be:
  - lost
  - delivered out of order to appl
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others



UDP segment format

UDP (more)

- suitable for interactive streaming multimedia applications
  - loss tolerant
  - min rate required
- other UDP uses, e.g.
  - DNS
  - SNMP
  - DHCP
- reliable transfer over UDP?

  add reliability in application layer
  - application-specific error recovery

Advantages of UDP

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

Internet checksum

Sender:

- treat segment as a sequence of 16-bit integers (with checksum field initialized to zero)
- add integers using 1's complement arithmetic ***and*** take 1's complement of the sum
- put result as checksum value into checksum field
- detail: pseudoheader consisting of protocol no., IP addresses, segment length field (again) included in checksum calculation

Receiver:

- compute 1's complement sum of received segment (checksum field included)
- check if computed sum equals sixteen 1's:
  - NO - error detected
  - YES - no error detected *But maybe errors nonetheless?* More later ....

Internet Checksum Example

- Notes
  - In ones complement arithmetic, a negative integer -*x* is represented as the complement of *x*, i.e., each bit of *x* is inverted
  - 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

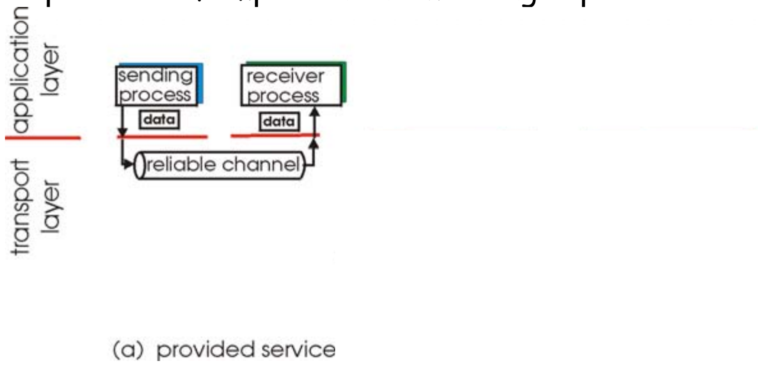
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer

  (my slides do not follow Kurose & Ross)
- 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

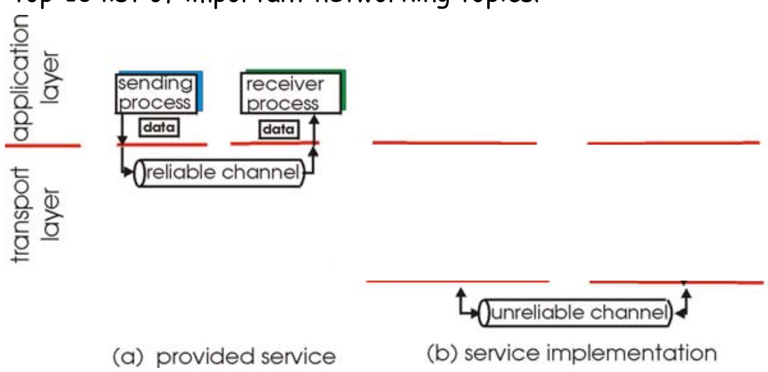
Principles of Reliable data transfer

- important in application, transport, link layers
- top-10 list of important networking topics!



Principles of Reliable data transfer

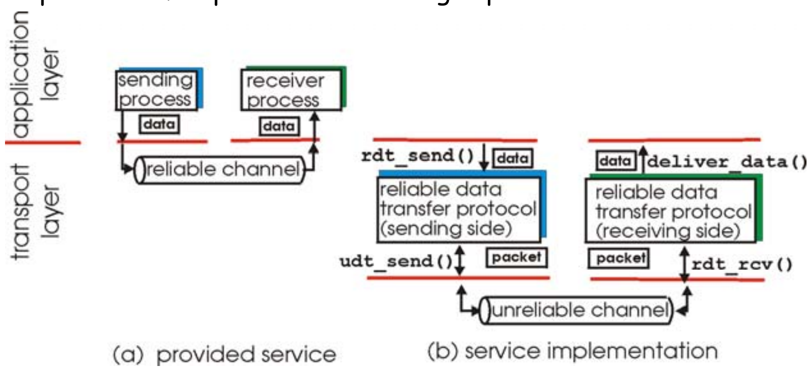
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



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

10/17/2017

Transport Layer (SSL) 3-19

Channel Abstractions

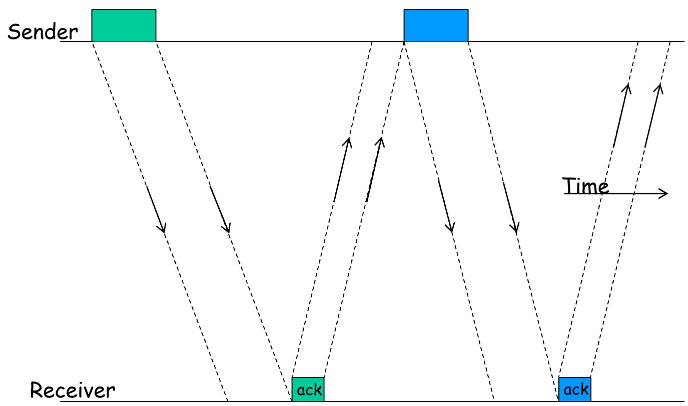
- Lossy FIFO channel
  - delivers a subsequence in FIFO order
  - example: delivery service provided by a physical link
- Lossy, reordering, duplicative (LRD) channel
  - example: delivery service provided by IP or by UDP protocol

10/17/2017

Transport Layer (SSL) 3-20

Stop-and-wait ARQ (automatic repeat request)

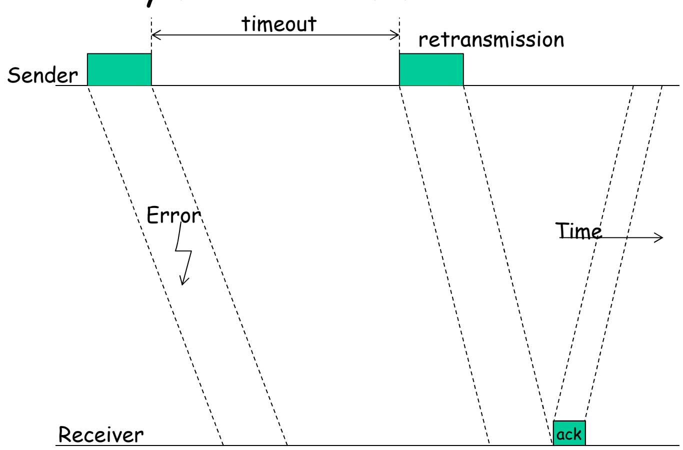
- Error-free operation



10/17/2017

Stop-and-wait ARQ

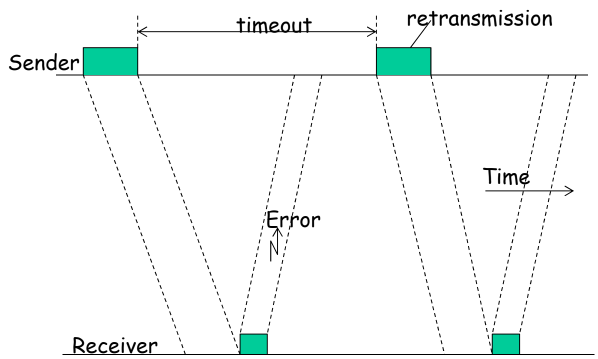
- Retransmission after timeout
- Recovery from loss of frame



10/17/2017

Stop-and-wait ARQ

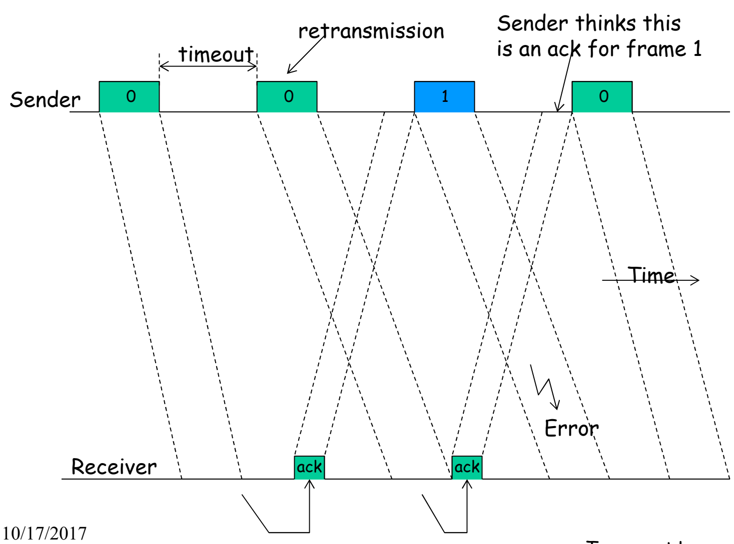
- Retransmission after timeout to recover from loss of ack
- Receiver gets *duplicate* frame
  - Sequence number needed in frame



10/17/2017

Stop-and-wait ARQ

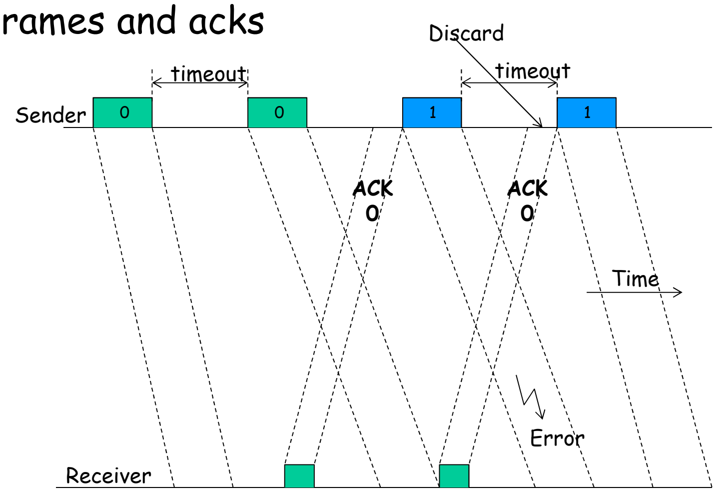
- Sequence number also needed in ack



10/17/2017

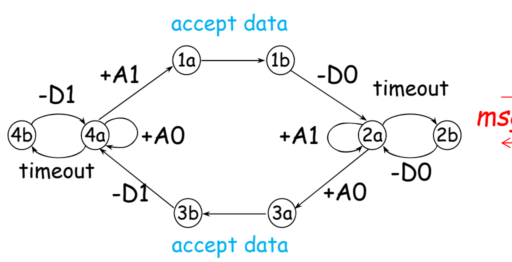
Stop-and-wait ARQ

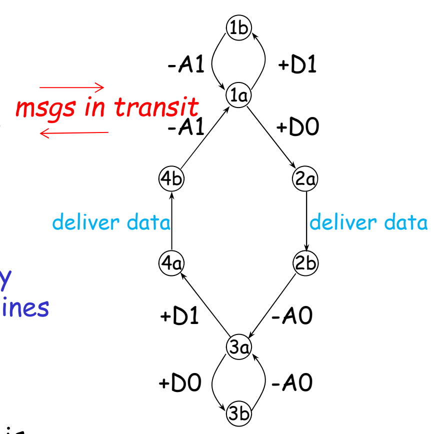
- Operation with 1-bit sequence numbers in frames and acks



Alternating-Bit Protocol

Sender P1 (initial state = 1a) **Receiver P2** (initial state = 1a)





- Sender and Receiver specified by communicating finite-state machines
- Notation for edge labels

-m send message m

+m receive message m if it is waiting to be received

Protocol state space is infinite

Alternating-Bit Protocol (cont.)

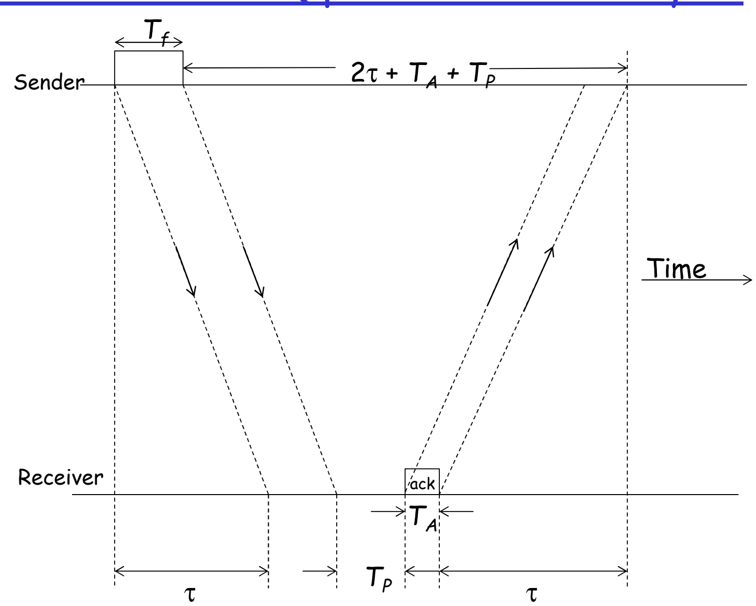
- **Assertion**: If Sender and Receiver communicate via lossy FIFO channels, the alternating-bit protocol provides reliable in-order data delivery.
- **Assumption**: A frame is retransmitted infinitely many times if it is lost infinitely many times.

Note: A real protocol is typically designed to retransmit a fixed number of times (say k).

10/17/2017

Transport Layer (SSL) 3-27

Stop-and-wait ARQ performance analysis



10/17/2017

Transport Layer (SSL) 3-28

$P \;=\;$ probability transmission is unsuccessful

$b_i \;=\;$ Prob[success after i transmissions] for $i = 1, 2, \ldots$

$\boxed{b_i \;=\; P^{i-1}(1-P)}$

Average number of transmissions per frame

$$\boxed{N_f \;=\; \sum_{i=1}^{\infty} i\, b_i = \sum_{i=1}^{\infty} i\, P^{i-1}\,(1-P)}$$

$$\boxed{=\; (1-P)\sum_{i=1}^{\infty} i\, P^{i-1}}$$

$$\boxed{=\; (1-P)\frac{d}{dP}\sum_{i=1}^{\infty} P^{i} = (1-P)\frac{d}{dP}\sum_{i=0}^{\infty} P^{i}}$$

$$\boxed{=\; (1-P)\frac{d}{dP}\frac{1}{1-P} = (1-P)\frac{1}{(1-P)^2}}$$

$$\boxed{=\; \frac{1}{1-P} = N_f}$$

Timeout duration $\boldsymbol{T} > 2\tau + T_A + T_P$

Each unsuccessful transmission uses

$T_f + \boldsymbol{T}$

Each successful transmission uses

$T_f + 2\tau + T_A + T_P$

Average time per frame

$(N_f - 1)\,(T_f + \boldsymbol{T}) + (T_f + 2\tau + T_A + T_P)$

Max. utilization (throughput) of stop-and-wait

$$\mathrm{U} = \frac{T_f}{\frac{P}{1-P}(T_f + T) + T_f + 2\tau + T_A + T_P}$$

Propagation delay versus transmission time

Assume $P = 0$, $T_A = 0$, $T_P = 0$

$$U \cong \boxed{\frac{T_f}{T_f + 2\tau}} \qquad \text{(upper bound)}$$

$$= \frac{1}{1 + \frac{2\tau}{T_f}} = \frac{1}{1 + 2a} \qquad \text{where } a = \frac{\tau}{T_f}$$

Note:

$$\tau = \boxed{\frac{\text{distance}}{\text{propagation speed}}}$$

$$T_f = \boxed{\frac{\text{frame length}}{\text{transmission rate}}}$$

Performance of AB protocol

- AB protocol works, but performance degrades for channels with large *delay-bandwidth* product

- example: 1 Gbps link, 15 ms prop. delay, 1KByte packet

$$T_{transmit} = \frac{8\text{Kbits}}{10^{**}9 \text{ bits/sec}} = 8 \text{ microsec}$$

$$U = \frac{8 \text{ microsec}}{30008 \text{ microsec}} = 0.00027$$

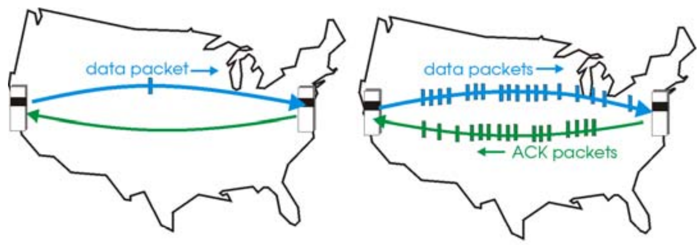
  - the protocol limits use of available bandwidth

- Note: If the sender and receiver are connected by the Internet, then τ is the end-to-end Internet delay

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver

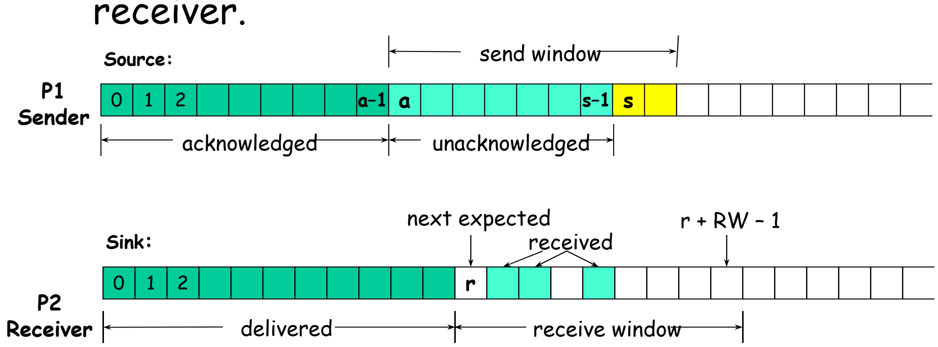


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Pipelined protocols: (i) concurrent logical channels (used in ARPANET), (ii) sliding window protocol (TCP)

Sliding Window Protocol

- Consider an infinite array, Source, at the sender, and an infinite array, Sink, at the receiver.

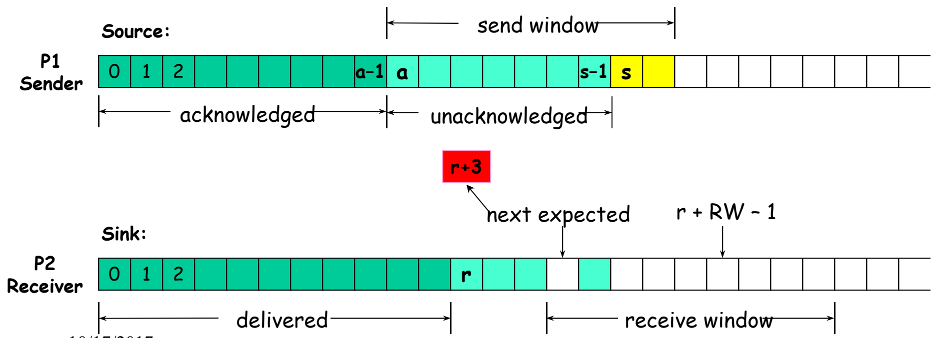


RW receive window size

SW send window size $(s - a \leq SW)$

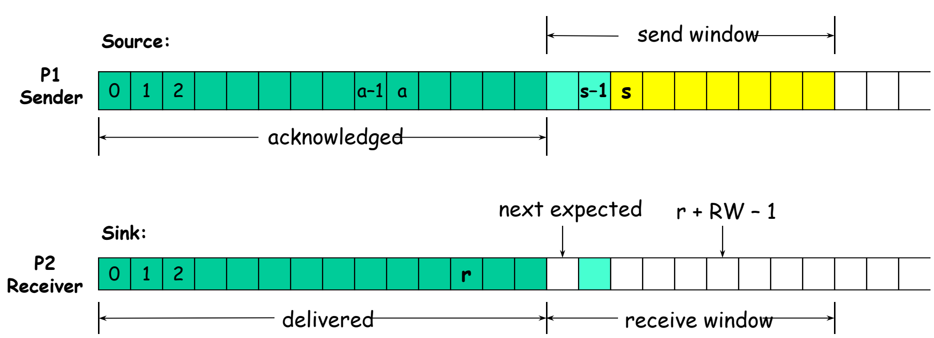
Sliding Windows in Action

- Data unit r has just been received by P2
  - Receive window slides forward
- P2 sends cumulative ack with sequence number it expects to receive next (r+3)



Sliding Windows in Action

- P1 has just received cumulative ack with r+3 as next expected sequence number
  - Send window slides forward



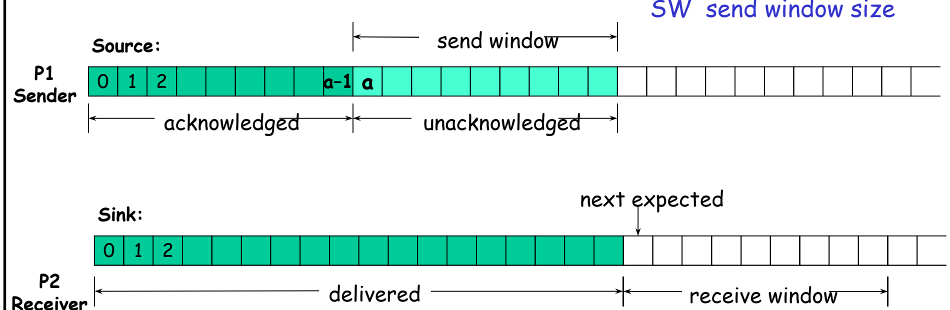
Sliding Window protocol

- Functions provided
  - error control (reliable delivery)
  - in-order delivery
  - flow and congestion control (by varying send window size)
- TCP uses cumulative acks (needed for correctness)
- Other kinds of acks (to improve performance)
  - selective nack
  - selective ack (TCP SACK)
  - bit-vector representing entire state of receive window (in addition to first sequence number of window)

Sliding Windows for Lossy FIFO Channels

- A small number of bits in packet header for sequence number
- Necessary and sufficient condition for correct operation: $SW + RW \leq MaxSeqNum$
- *Necessity*:

RW receive window size
SW send window size



Sliding Windows for Lossy FIFO Channels

- Sufficiency can only be demonstrated by using a formal method to prove that the protocol provides reliable in-order delivery. See Shankar and Lam, *ACM TOPLAS*, Vol. 14, No. 3, July 1992.
- Interesting special cases
  - SW = RW = 1
    alternating-bit protocol
  - SW = 7, RW = 1
    out-of-order arrivals not accepted, e.g., HDLC
  - SW = RW

Sliding Windows for LRD Channels

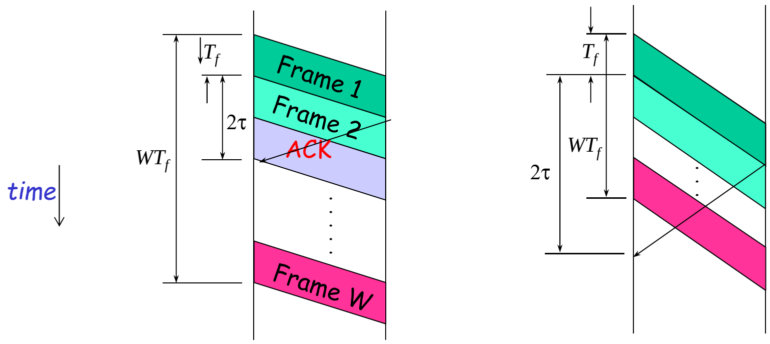
- Assumption: Packets have bounded lifetime L
- Be careful how fast sequence numbers are consumed (i.e., by arrival of data to be sent into network)

  $(\text{send rate}) \times L < \text{MaxSeqNum}$
- TCP
  - 32-bit sequence numbers
  - counts bytes
  - assumes that datagrams will be discarded by IP if too old

Sliding Window Protocol Performance Analysis

Go to slide 3-46

- Assumptions
  - ack transmission time is negligible, $T_A = 0$
  - receiver processing time is negligible, $T_P = 0$
  - send window size is W



(a) $WT_f > 2\tau + T_f$ (b) $WT_f < 2\tau + T_f$

Performance for Error-Free Channels

- Maximum utilization

$$U = \begin{cases} 1 & WT_f > 2\tau + T_f \\ \dfrac{WT_f}{T_f + 2\tau} & WT_f \le 2\tau + T_f \end{cases}$$

- Define $a = \tau / T_f$

$$U = \begin{cases} 1 & W > 2a + 1 \\ \dfrac{W}{1 + 2a} & W \le 2a + 1 \end{cases}$$

W=1 is special case of alternating-bit protocol

Performance Analysis for Error-Prone Channels

- Define

  N_f = Average number of transmissions per frame

- Maximum utilization

$$U = \begin{cases} \dfrac{1}{N_f} & W > 2a+1 \\ \dfrac{W/N_f}{1+2a} & W \le 2a+1 \end{cases}$$

- To determine N_f for two cases
  - Selective repeat (optimistic performance)
  - Go-back-N (pessimistic performance)

Performance Analysis of Error-Prone Channels

P = probability a transmission is unsuccessful

- Selective repeat (-> upper bound on U)

$$N_f = \frac{1}{1-P}$$

$$\boxed{U = \begin{cases} 1-P & W > 2a+1 \\ \dfrac{W(1-P)}{1+2a} & W \le 2a+1 \end{cases}}$$

- Go-back-N (-> lower bound on U)
  Each lost frame requires the retransmission of N frames where $1 \le N \le W$

Go-back-N (cont.)

- With probability $(1-P)P^i$, a frame requires $1+iN$ transmissions to succeed, for $i=0,1,\ldots$

$$\boxed{N_f = \sum_{i=0}^{\infty}(1+iN)(1-P)P^i}$$

$$\boxed{= (1-P)\sum_{i=0}^{\infty}P^i + NP(1-P)\sum_{i=0}^{\infty} i\,P^{i-1}}$$

$$\boxed{= 1+NP(1-P)\frac{d}{dP}\sum_{i=0}^{\infty}P^i}$$

$$\boxed{= 1+NP(1-P)\frac{d}{dP}\frac{1}{1-P}}$$

$$\boxed{= 1+NP(1-P)\frac{1}{(1-P)^2}}$$

$$\boxed{= 1+\frac{NP}{1-P}}$$

10/17/2017

Go-back-N (cont.)

What is N ? Go to slide 3-41

From previous slide

$$N_f = 1+\frac{NP}{1-P}$$

For $WT_f > 2\tau + T_f$ *Case (a)*

$$\boxed{NT_f \cong T_f + 2\tau}$$

$$\boxed{N \cong 1+2a}$$

$$N_f = 1+\frac{(1+2a)P}{1-P} = \frac{1-P+P+2aP}{1-P} = \boxed{\frac{1+2aP}{1-P}}$$

For $WT_f \le 2\tau + T_f$ *Case (b)*

$$N = W$$

$$N_f = 1+\frac{WP}{1-P} = \boxed{\frac{1-P+WP}{1-P}}$$

10/17/2017

Go-back-N (cont.)

From previous slide

- Recall (from slide 3-43)

$$U = \begin{cases} \dfrac{1}{N_f} & W > 2a+1 \\ \dfrac{W/N_f}{1+2a} & W \le 2a+1 \end{cases}$$

$$\boxed{N_f = \frac{1+2aP}{1-P}}$$

$$\boxed{N_f = \frac{1-P+WP}{1-P}}$$

- Maximum utilization

$$U = \begin{cases} \dfrac{1-P}{1+2aP} & W > 2a+1 \\ \dfrac{W(1-P)}{(1+2a)(1-P+WP)} & W \le 2a+1 \end{cases}$$

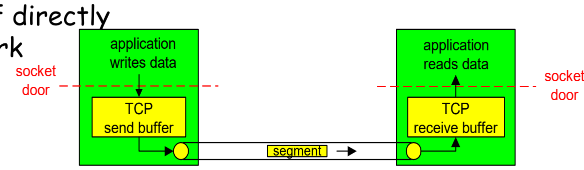
Chapter 3 outline

- 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

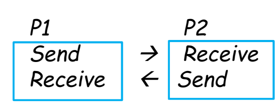
TCP: Overview

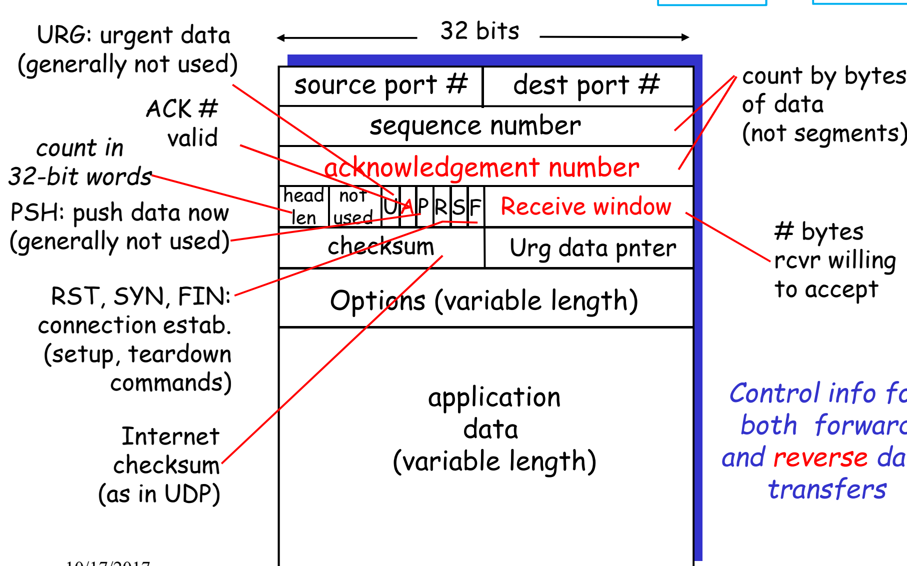
RFCs: 793, 1122, 1323, 2018, 2581

- connection-oriented
  - handshaking initializes sender, receiver **state** before data exchange
- point-to-point
  - two sender-receiver pairs
  - bi-directional data flows in same connection
- MSS: maximum segment size
  - less than MTU of directly connected network
- *reliable, in-order byte steam service*
  - no "message boundaries"
  - send and receive buffers
- pipelined
  - send window size determined by TCP congestion and flow control



TCP segment structure





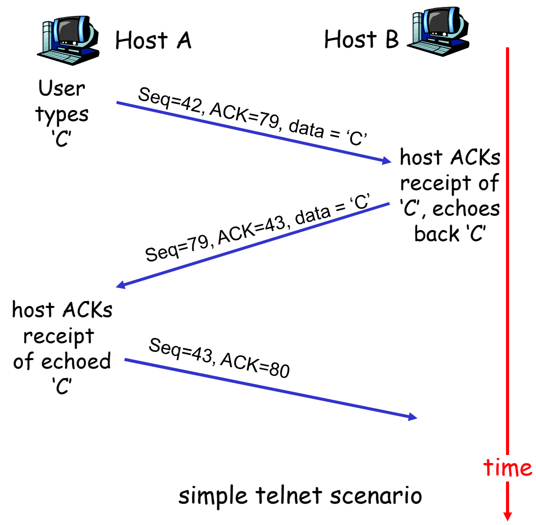
Control info for both forward and reverse data transfers

TCP seq. #'s and ACKs

Seq. #
- sequence number of first byte in segment's data

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

Q: how receiver handles out-of-order segments?
- TCP spec doesn't say, up to implementor



simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies, may be too short or too long
- 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
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current `SampleRTT`

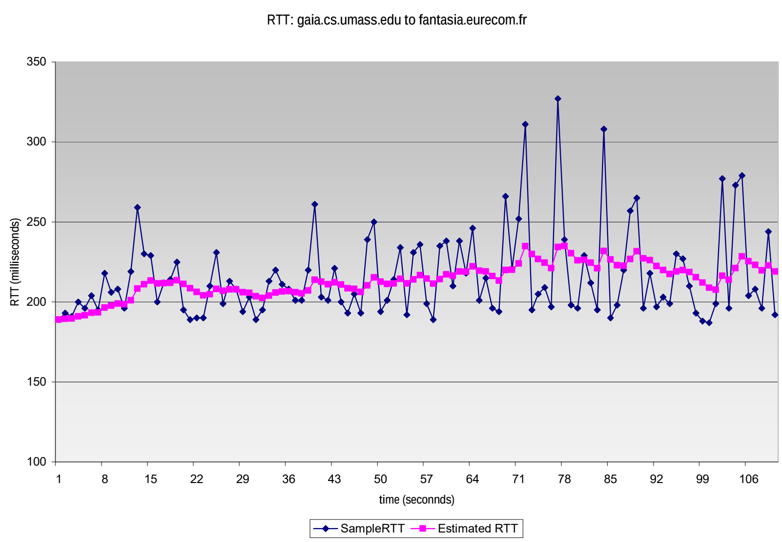
TCP Round Trip Time and Timeout

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

- Exponentially weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

10/17/2017

Example RTT estimation:



10/17/2017

Setting the timeout interval

- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> larger safety margin
- estimate how much SampleRTT deviates from EstimatedRTT and update

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```


Chapter 3 outline

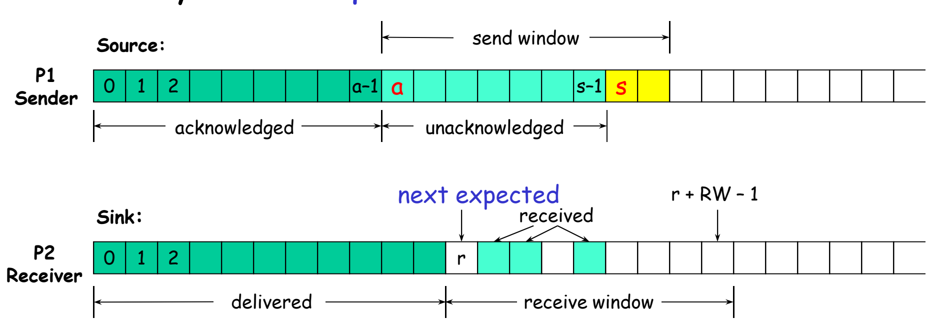
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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

TCP reliable data transfer

- TCP creates reliable service on top of IP's unreliable service
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - **duplicate acks**
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

Sliding Window Protocol

At the sender, a will be pointed to by SendBase, and s by NextSeqNum



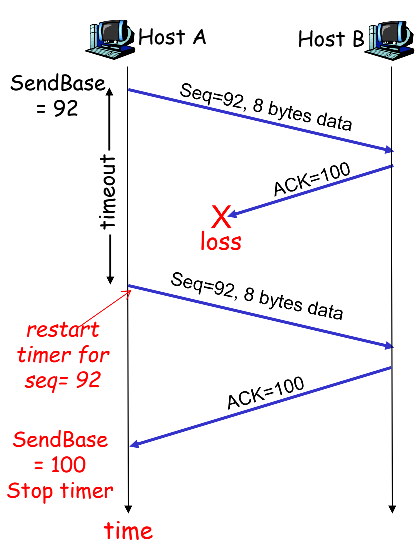
RW receive window size
SW send window size ($s - a \leq SW$)

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
 loop (forever) {
   switch(event)

   event: data received from application above
                        and send window has enough room
       create TCP segment with sequence number NextSeqNum
       if (timer currently not running)
           start timer
       pass segment to IP
       NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
       retransmit not-yet-acknowledged segment with
            smallest sequence number
       start timer

    event: ACK received, with ACK field value = y
       if (y > SendBase) {
           SendBase = y
          if (there are currently not-yet-acknowledged segments)
                start timer;
          else stop timer
          }
 } /* end of loop forever */
```

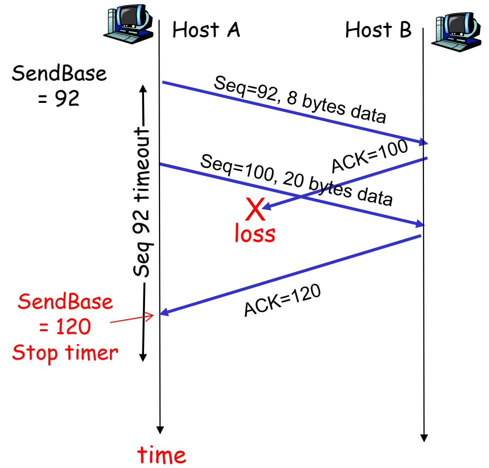
Note:
- y > SendBase means new data ack'ed

TCP: retransmission scenario (1)



lost ACK scenario

TCP retransmission scenario (2)

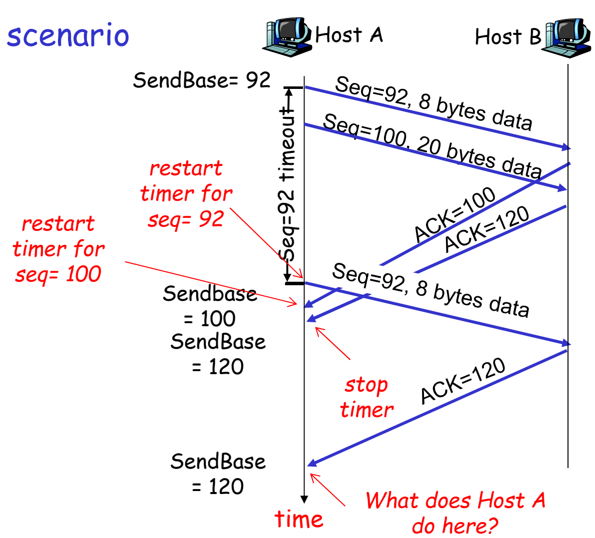


Cumulative ACK scenario

10/17/2017

TCP: retransmission scenario (3)

premature timeout scenario

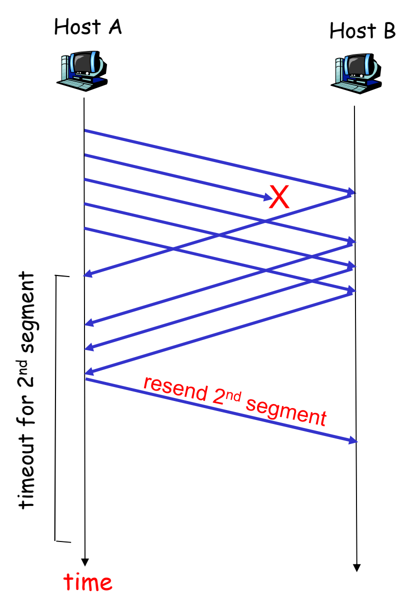


10/17/2017

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 ***duplicate ACKs*** for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment *before timer expires*

10/17/2017



Resending a segment after triple duplicate ACK without waiting for timeout

10/17/2017

Fast retransmit algorithm:

```
event: ACK received, with ACK field value = y
          if (y > SendBase) {
              SendBase = y
              if (there is a not-yet-acknowledged segment)
                  start timer
           }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
                 reset timer for y
               }
```

a duplicate ACK for already ACKed segment

fast retransmit

10/17/2017

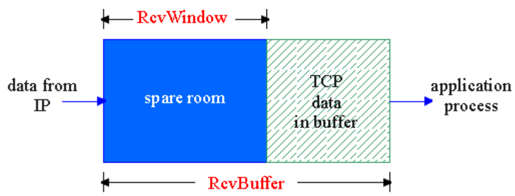
Chapter 3 outline

- 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

10/17/2017

TCP Flow Control

flow control
sender won't overrun receiver's buffers by transmitting too much, too fast



buffer at receive side of a TCP connection

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- `RcvWindow` **field** in TCP segment header

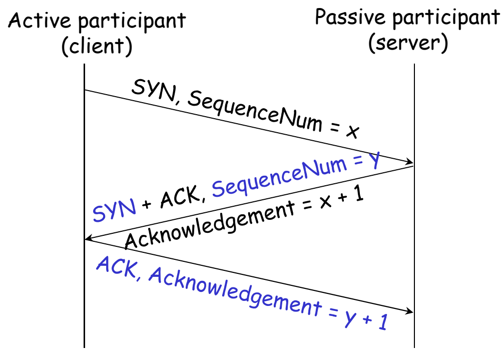
sender: keeps amount of transmitted, unACKed data less than most recently received `RcvWindow` `value`

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 Connection Management

- initialize TCP variables
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)



Three way handshake

Step 1: client end system sends TCP SYN control segment to server - initial seq number chosen at random

Step 2: server end system receives SYN, replies with SYNACK control segment
- allocates buffers
- specifies server-to-receiver initial seq. # (chosen at random)

Step 3: client end system replies with ack # (likely piggybacked in segment with app data)

TCP Connection Management (cont.)

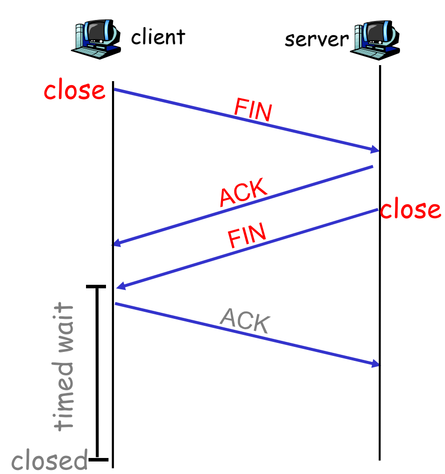
Closing a connection:

client closes socket

Step 1: client end system sends TCP FIN control message to server

Step 2: server receives FIN, replies with ACK.

Later no more data to send. It closes connection, sends FIN.



TCP Connection Management (cont.)

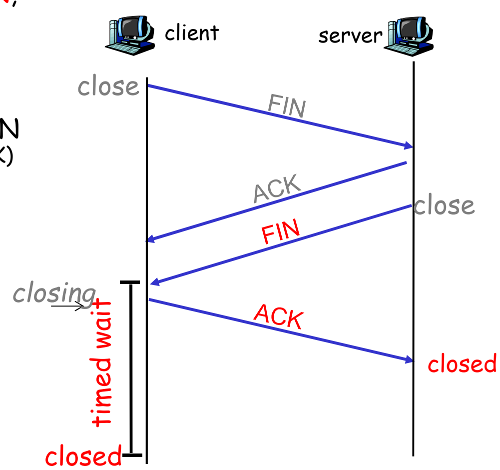
Step 3: client receives FIN, replies with ACK and enters "timed wait"

- will respond with ACK to a retransmitted FIN (due to loss of previous ACK)

Step 4: server receives ACK. Its connection is closed.

Step 5: client closes connection at the end of timed wait

Note: protocol spec allows simultaneous FINs



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

Principles of Congestion Control

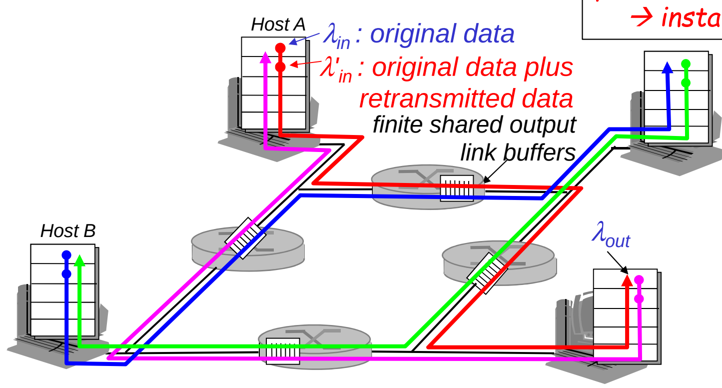
Congestion:
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control
- manifestations:
  - long delays (queueing in router buffers)
  - lost packets (buffer overflow at routers)
- a top-10 problem!

Causes/costs of congestion: scenario

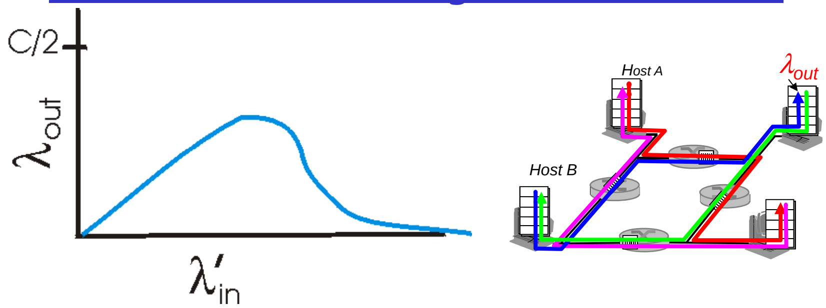
- four senders
- multi-hop paths
- Timeout & retransmit

Q: what happens as λ_{in} and λ'_{in} increase at every sender?

positive feedback → instability



Causes/costs of congestion: scenario



Cost of congestion

- when a packet is dropped, any upstream transmission capacity used for that packet was *wasted*
- behavior on right side of above graph called congestion collapse

Approaches towards congestion control

End-to-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system's observed loss (or delay)
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion, e.g., SNA, DECbit, ATM
  - TCP/IP explicit congestion notification (ECN)
  - explicit sending rate for sender

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 Congestion Control

- end-to-end control (no network assistance)
- sender limits transmission:
  `LastByteSent-LastByteAcked ≤ CongWin`
- Roughly, the send buffer's

$$\text{throughput} \leq \frac{\text{CongWin}}{\text{RTT}} \text{ bytes/sec}$$

where CongWin is in bytes
and throughput is λ'_{in} in slide 3-74

How does sender determine CongWin?

- loss event = **timeout** *or* **3 duplicate acks**
- TCP sender reduces `CongWin` after a loss event

three mechanisms:

- slow start
- reduce to 1 segment after timeout event
- AIMD (additive increase multiplicative decrease)

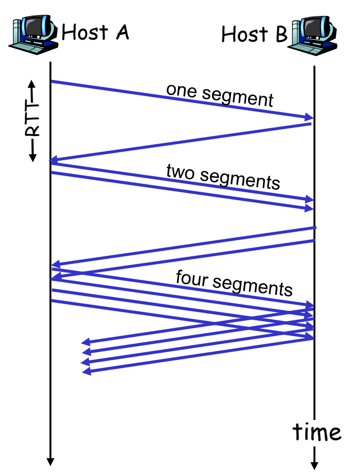
Note: For now consider RcvWindow to be very large such that the send window size is equal to CongWin. They are referred to as rwnd and cwnd, respectively, in the textbook.

TCP Slow Start

- Probing for usable bandwidth
- When connection begins, `CongWin` = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 2500 bytes/sec = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to a higher rate

TCP Slow Start (more)

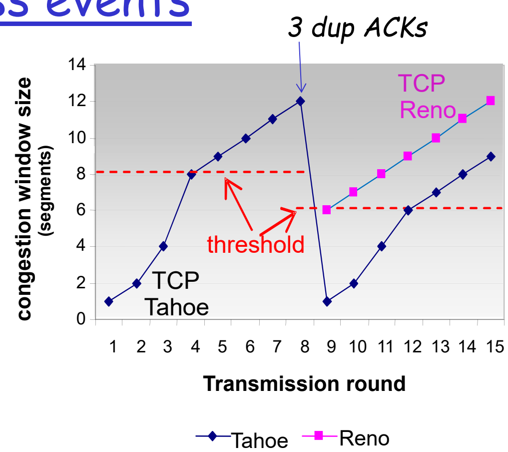
- When connection begins, increase rate exponentially until first loss event or "threshold"
  - double `CongWin` every RTT
  - done by incrementing `CongWin` by 1 MSS for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



Congestion avoidance state & responses to loss events

Q: If no loss, when should the exponential increase switch to linear?

A: When CongWin gets to current value of threshold



Notes: 1. For simplicity, CongWin is in number of segments in the above graph. 2. Reno's window inflation and deflation steps (details) omitted

Implementation:

- For initial slow start, threshold is set to a large value (e.g., 64 Kbytes)
- Subsequently, threshold is variable
- At a loss event, threshold is set to 1/2 of CongWin just before loss event

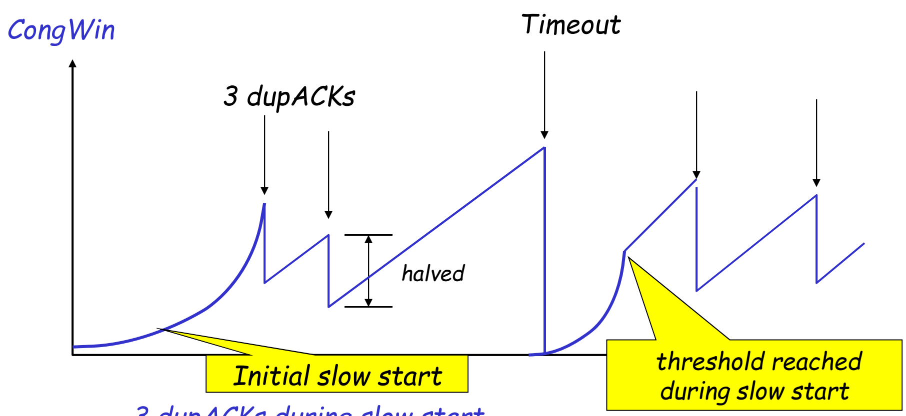
Rationale for Reno's Fast Recovery

- 3 dup ACKs indicate network capable of delivering some segments
- timeout occurring before 3 dup ACKs is "more alarming"

- After 3 dup ACKs:
  - CongWin is cut in half (*multiplicative decrease*)
  - window then grows linearly (*additive increase*)
- But after timeout event:
  - CongWin is set to 1 MSS instead;
  - window then grows exponentially to a threshold, then grows linearly

Additive Increase Multiplicative Decrease (AIMD)

TCP Reno (example scenario)



3 dupACKs during slow start before reaching initial threshold

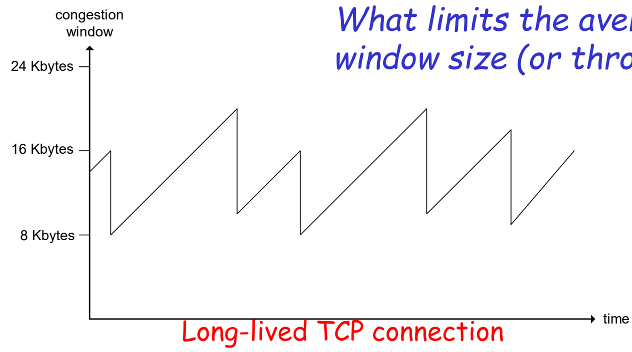
Summary: TCP Congestion Control (Reno)

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially (until loss event or exceeding threshold).
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold`.
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS.

AIMD in steady state (when no timeout)

additive increase:
increase `CongWin` by 1 MSS every RTT in the absence of any loss event: *probing*

multiplicative decrease:
cut `CongWin` in half after loss event (3 dup acks)



What limits the average window size (or throughput)?

TCP Throughput limited by loss rate

- TCP average throughput (approximate) of send buffer under AIMD in terms of loss rate, *L*

$$throughput = \frac{1.22 \times MSS}{RTT\sqrt{L}} \text{ bytes/second}$$

where MSS is number of bytes per segment

- Example: 1500-byte segments, 100ms RTT, to get 10 Gbps throughput, loss rate needs to be very low

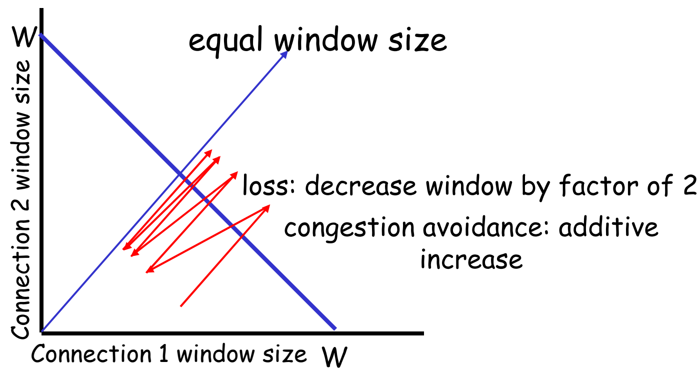
$$L = 2 \cdot 10^{-10}$$

- New version of TCP needed for high-speed applications

Is TCP fair?

Two competing sessions:

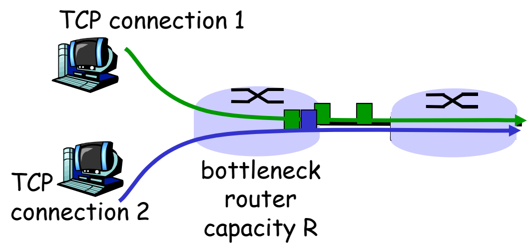
- Additive increase gives slope of 1, as window size increases
- multiplicative decrease reduces window size to half (proportionally)



10/17/2017

Is TCP fair?

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



AIMD only provides convergence to same window size, not necessarily same throughput rate

10/17/2017

No fairness in practice

UDP

- Some multimedia apps use UDP instead of TCP. They
  - can tolerate packet loss,
  - do not want rate throttled by congestion control – send at constant rate

Parallel TCP connections

- nothing prevents an app from opening parallel connections between 2 hosts.
  - Web browsers do this

10/17/2017

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - connection management
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"

10/17/2017