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
  - TCP: connection-oriented transport
  - TCP congestion control

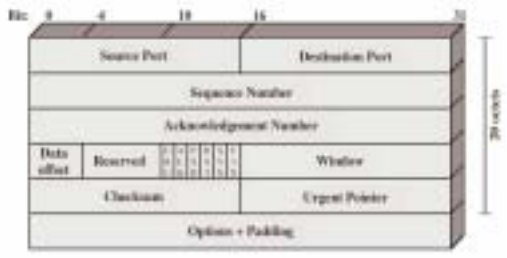
Chapter 3 outline

- 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
- 3.8 TCP fairness
- 3.9 Delay Modeling

TCP basics

- Reliable, ordered delivery
  - uses sequence numbers, acknowledgements, timeouts and retransmissions
  - End-to-end semantics (ACK *after* data recd)
- Provides flow and congestion control
  - uses sliding window based buffers and feedback from receiver/network to adjust transmission rate

TCP header



TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- 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

TCP sender events:

data rcvd from app:

- 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 rcvd:

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

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      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 of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
      }

} /* end of loop forever */
```

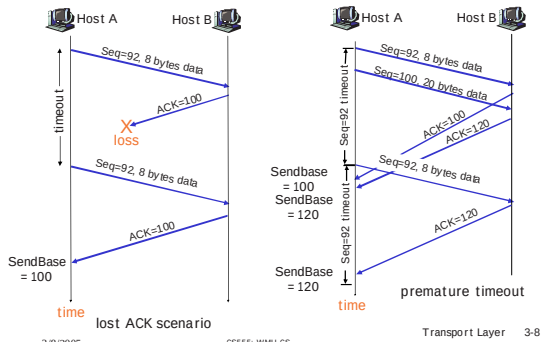
TCP sender (simplified)

Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

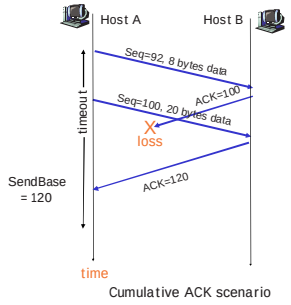
3/8/2005 CS555: WMU-CS Transport Layer 3-7

TCP: retransmission scenarios



3/8/2005 CS555: WMU-CS Transport Layer 3-8

TCP retransmission scenarios (more)



3/8/2005 CS555: WMU-CS Transport Layer 3-9

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

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 supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  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
                }
```

a duplicate ACK for already ACKed segment

fast retransmit

Timeouts and retransmission

- TCP manages four different timers for each connection
  - retransmission timer: when awaiting ACK
  - persist timer: keeps window size information flowing
  - keepalive timer: when other end crashes or reboots
  - 2MSL timer: for the TIME_WAIT state

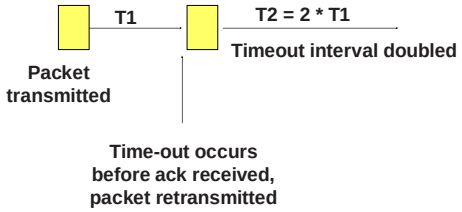
RTT estimation

Exponential Averaging Filter:

- Measure SampleRTT for segment/ACK pair
- Compute weighted average of RTT
  - EstimatedRTT = α EstimatedRTT + $(1 - \alpha)$ sampleRTT
  - Retransmit Time Out Interval
    RTO = β * EstimatedRTT
- Typically $\alpha = 0.9$; $\beta = 2$

Timeout: Exponential backoff

- Double RTO on each timeout



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
  - bit errors
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
- 3.8 TCP fairness
- 3.9 Delay Modeling

Bit Errors

- Send ACK (duplicate) when checksum not matched, forcing sender to retransmit

TCP Flow Control

- receive side of TCP connection has a receive buffer:

- app process may be slow at reading from buffer

flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works

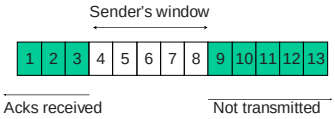
(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
  = **RcvWindow**
  = **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

Window based flow control

- Window size minimum of
  - receiver's advertised window - determined by available buffer space at the receiver
  - congestion window - determined by sender, based on network feedback



Advertised window

- AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - NextByteRead)
- SendingWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)

- Sender uses persist timer to probe receiver when AdvertisedWindow=0.

Congestion window

Limits amount of data in transit

- MaxWin = MIN (CongestionWindow, AdvertisedWindow)
- EffectiveWin = MaxWin - (LastByteSent - LastByteAcked)

Congestion control

- On detecting a packet loss
  - TCP sender assumes that network congestion has occurred
  - TCP sender drastically reduces the congestion window
- Reducing congestion window reduces amount of data that can be sent per RTT

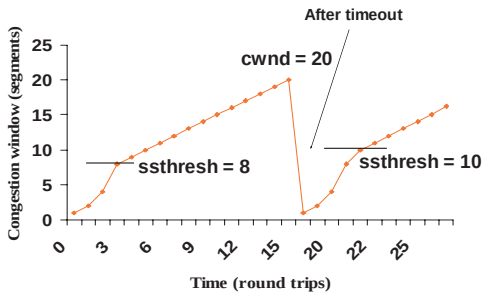
AIMD: Additive increase Multiplicative decrease

- Source infers congestion upon RTO
- Increase CongestionWindow (linearly, by 1 segment per RTT) when congestion goes down
- Decrease CongestionWindow (multiplicatively, by factor of 1/2) when congestion goes up

Slow start and Congestion avoidance

- AIMD may be too conservative
- CongestionWindow: cwnd
- Slow Start
  - Increase cwnd exponentially upto a threshold (ssthresh)
- Congestion Avoidance
  - Increase cwnd linearly after ssthresh

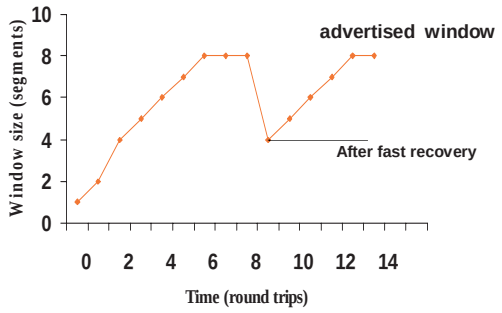
Typical TCP behaviour



Fast retransmit and Fast recovery

- Waiting for TCP sender timeouts leads to idle periods
- Fast retransmit: use duplicate (triplicate) ACKs to trigger retransmission
- Fast recovery
  - remove the slow start phase;
  - go directly to half the last successful cwnd

Fast retransmit and Fast recovery

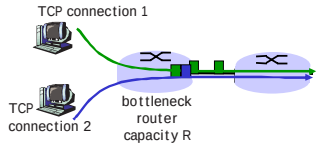


Chapter 3 outline

- 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
- 3.8 TCP fairness
- 3.9 Delay Modeling

TCP Fairness

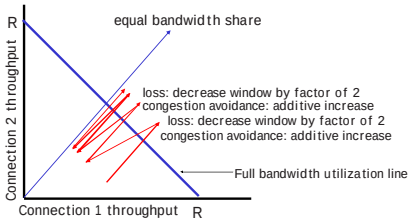
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

HW 5?

- RDT protocol implementation using Java
  - Ideas to be liberally borrowed from http://zoo.cs.yale.edu/classes/cs433/assignments/prog1/
  - Get started even before I customize it to our environment

HW 6? TBD

- FTP response time modeling using OPNET?
- UDP app versus TCP app response times?