CMSC 332
Computer Networks
TCP (I)

Professor Szajda
• A number of students have pointed out that there is a difference between the way networking and other areas represent “kilo” units.
  ‣ i.e., KB
  ‣ Some have proposed “kibi-”, “mibi-” and “gibi-” instead.
• In networking, “Kilo” is treated as 1000, not 1024.
• Some problems may ask you to show the difference between KB transferred and KB stored. This is the reason.
Last Time

• Discussed a variety of algorithms that can give us guarantees of reliable delivery.
  ‣ What were they?
  ‣ How do they differ?

• Finite State Machines (FSMs) are a powerful means of representing protocols.
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

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order byte steam:**
  - no “message boundaries”

- **pipelined:**
  - TCP congestion and flow control set window size

- **send & receive buffers**

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **connection-oriented:**
  - handshaking (exchange of control msgs) init’s sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
TCP segment structure

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum**: (as in UDP)

**TCP Segment Structure**

- **Source port #**
- **Destination port #**
- **Sequence number**
- **Acknowledgment number**
- **Receive window**
- **Checksum**
- **Urgent data pointer**
- **Options (variable length)**

**Application Data**

- **URG**: urgent data
- **ACK**: ACK # valid
- **PSH**: push data now
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum**: (as in UDP)

- **Receive window**
- **Checksum**
- **Urgent data pointer**
- **Options (variable length)**

- **Application data**
- **(variable length)**

- **# bytes rcvr willing to accept**
- **Counting by bytes of data (not segments!)**

---

CMSC 332: Computer Networks
TCP seq. #’s and ACKs

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 receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor

User types ‘C’

Host A
Seq=42, ACK=79, data = ‘C’

Host B
Seq=79, ACK=43, data = ‘C’

host ACKs receipt of ‘C’, echoes back ‘C’

Seq=43, ACK=80

host ACKs receipt of echoed ‘C’

simple telnet scenario
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
  ‣ ignore retransmissions

• **SampleRTT** will vary, want estimated RTT “smoother”
  ‣ average several recent measurements, not just current **SampleRTT**
TCP Round Trip Time and Timeout

\[
\text{EstimatedRTT} = (1 - \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}
\]

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: \( \alpha = 0.125 \)
Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr
TCP Round Trip Time and Timeout

**Setting the timeout**

- **EstimatedRTT** plus “safety margin”
  - large variation in **EstimatedRTT** → larger safety margin
- first estimate of how much **SampleRTT** deviates from **EstimatedRTT**:
  
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]

  (typically, \( \beta = 0.25 \))

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
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 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: \( \text{TimeOutInterval} \)

**timeout:**
- retransmit segment that caused timeout
- restart timer

**Ack rcvd:**
- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments
TCP sender (simplified)

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 */

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
TCP: retransmission scenarios

**Host A**
- Seq=100, 20 bytes data
- ACK=100

**Host B**
- Seq=92, 8 bytes data
- ACK=120
- Seq=92 timeout
- ACK=120

**Host A**
- Seq=92, 8 bytes data
- ACK=100

**Host B**
- Seq=92 timeout
- ACK=120

**SendBase**
- Host A: 100
- Host B: 120

**Lost ACK scenario**

**Premature timeout**
TCP retransmission scenarios (more)

Cumulative ACK scenario

Host A
Seq=92, 8 bytes data
ACK=100

Host B
Seq=100, 20 bytes data
ACK=100

X
loss

SendBase = 120

SendBase = 120

Cumulative ACK scenario
# TCP ACK generation

[RFC 1122, RFC 2581]

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

Figure 3.37 Resending a segment after triple duplicate ACK
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
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 Flow Control

- receive side of TCP connection has a receive 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

- app process may be slow at reading from buffer
TCP Flow control: how it works

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

- spare room in buffer
  
  \[= \text{RcvWindow} \]
  
  \[= \text{RcvBuffer} - \left[ \text{LastByteRcvd} - \text{LastByteRead} \right] \]

- Rcvr advertises spare room by including value of \textbf{RcvWindow} in segments

- Sender limits unACKed data to \textbf{RcvWindow}
  - guarantees receive buffer doesn’t overflow
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

**Recall:** TCP sender, receiver establish "connection" before exchanging data segments

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

- **client**: connection initiator
  
  ```c
  connect(sock, &addr, ...);
  ```

- **server**: contacted by client
  
  ```c
  accept(sock, &addr, ...);
  ```

**Three way handshake:**

**Step 1:** client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

**Step 2:** server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
**Closing a connection:**

client closes socket:

```c
close(sock);
```

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

**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.
**Step 3:** client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle

Closed

TIME_WAIT

FIN_WAIT_2

FIN_WAIT_1

SYN_SENT

ESTABLISHED

LAST_ACK

CLOSE_WAIT

SYN_RCVD

LISTEN

CLOSED

wait 80 seconds

receive FIN send nothing

receive ACK send nothing

send SYN

receive SYN & ACK send ACK

send FIN

receive ACK send nothing

receive FIN send ACK

receive ACK send nothing

client application initiates a TCP connection

client application initiates close connection

server application creates a listen socket
SYN Flooding

- Classic Internet attack sends a huge number of SYN packets to a host, but never responds with the third handshake message.

- In so doing, an adversary forces a receiver to dedicate a huge amount of resources to bogus requests.
  - And therefore makes those resources unavailable to legitimate users.

- There are ways to prevent this (SYN Cookies), but a surprising number of systems are still vulnerable.
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:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput
Causes/costs of congestion: scenario 2

- one router, **finite** buffers
- sender retransmission of lost packet
Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- “perfect” retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes $\lambda_{out}$ larger (than perfect case) for same $\lambda'_{in}$

```
\begin{align*}
\text{a.} & \quad \lambda_{out} = \frac{R}{2} \\
\text{b.} & \quad \lambda_{out} = \frac{R}{3} \\
\text{c.} & \quad \lambda_{out} = \frac{R}{4}
\end{align*}
```

“costs” of congestion:
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{in}$ and $\lambda'_{in}$ increase?

Host A

$\lambda_{in}$: original data

$\lambda'_{in}$: original data, plus retransmitted data

finite shared output link buffers

Host B

$\lambda_{out}$
Another “cost” of congestion:

• when packet dropped, any “upstream transmission capacity used for that packet was wasted!
Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at
Case study: ATM ABR congestion control

**ABR: available bit rate:**
- “elastic service”
- if sender’s path “underloaded”:
  - sender should use available bandwidth
- if sender’s path congested:
  - sender throttled to minimum guaranteed rate

**RM (resource management) cells:**
- sent by sender, interspersed with data cells
- bits in RM cell set by switches (“network-assisted”)
  - **NI bit:** no increase in rate (mild congestion)
  - **CI bit:** congestion indication
- RM cells returned to sender by receiver, with bits intact
Case study: ATM ABR congestion control

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender’s send rate thus maximum supportable rate on path

- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell
Next Time

• Read Section 3.7
  ‣ Congestion control in TCP