Chapter 3
Transport Layer

(TCP Essentials)

A note on the use of these ppt slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) that you mention their source (after all, we’d like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

© All material copyright 1996-2012
J.F Kurose and K.W. Ross, All Rights Reserved

These slides are an adapted version of the original material

Transport Layer 3-1
Chapter 3 outline

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management
- 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

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

- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number for data</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number for data</td>
</tr>
<tr>
<td>head len</td>
<td>Length of the header</td>
</tr>
<tr>
<td>Urg Pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>Urg</td>
<td>Urgent data flag</td>
</tr>
<tr>
<td>PSH</td>
<td>Push data flag</td>
</tr>
<tr>
<td>RST, SYN, FIN</td>
<td>Connection establishment, teardown commands</td>
</tr>
<tr>
<td>options</td>
<td>Variable length options</td>
</tr>
<tr>
<td>application data</td>
<td>Variable length application data</td>
</tr>
<tr>
<td>checksum</td>
<td>Internet checksum</td>
</tr>
<tr>
<td># bytes rcvr willing to accept</td>
<td>Number of bytes receiver willing to accept</td>
</tr>
</tbody>
</table>

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

Transport Layer 3-4
**TCP seq. numbers, ACKs**

**Sequence numbers:**
- Byte stream “number” of first byte in segment’s data

**Acknowledgements:**
- 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

![Diagram of TCP sequence numbers and acknowledgements]
TCP seq. numbers, ACKs

Host A

User types 'C'

host ACKs receipt of echoed 'C'

Seq=42, ACK=79, data = 'C'

Seq=79, ACK=43, data = 'C'

Seq=43, ACK=80

Host B

host ACKs receipt of 'C', echoes back 'C'

simple telnet scenario
TCP round trip time, timeout

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
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
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

Let’s 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 ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments
TCP sender (simplified)

- NextSeqNum = InitialSeqNum
- SendBase = InitialSeqNum

wait for event

Data received from application above

Create segment, seq. #: NextSeqNum
Pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
If (timer currently not running)
Start timer

ACK received, with ACK field value y

If (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    If (there are currently not-yet-acked segments)
        Start timer
    Else stop timer
}

Timeout
Retransmit not-yet-acked segment with smallest seq. #
Start timer
TCP: retransmission scenarios

lost ACK scenario

premature timeout
TCP: retransmission scenarios

Host A

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data

Host B

Seq=100, 20 bytes of data

ACK=100

X

ACK=120

Seq=120, 15 bytes of data

cumulative ACK

timeout
# TCP ACK generation

[RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th><strong>event at receiver</strong></th>
<th><strong>TCP receiver action</strong></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>
TCP 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.

TCP fast retransmit

if sender receives 3 ACKs for same data
(“triple duplicate ACKs”), resend unacked segment with smallest seq #
- likely that unacked segment lost, so don’t wait for timeout
TCP fast retransmit

Host A

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

Host B

ACK=100

ACK=100

ACK=100

ACK=100

timeout

fast retransmit after sender receipt of triple duplicate ACK
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

- The application may remove data from TCP socket buffers.
- Slowly delivered by the TCP receiver.

Flow control: Receiver controls the sender, so the sender won't overflow the receiver's buffer by transmitting too much, too fast.

Diagram: Receiver protocol stack with application process, TCP socket, TCP code, and IP code.
TCP flow control

- receiver “advertises” free buffer space by including \( rwnd \) value in TCP header of receiver-to-sender segments
  - \( RcvBuffer \) size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust \( RcvBuffer \)
- sender limits amount of unacked (“in-flight”) data to receiver’s \( rwnd \) value
- guarantees receive buffer will not overflow

\[ \text{buffered data} \]
\[ \text{free buffer space} \]
\[ \text{TCP segment payloads} \]
\[ \text{receiver-side buffering} \]

\( to \text{ application process} \)
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
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

```
Socket clientSocket =
    newSocket("hostname", "port number");

Socket connectionSocket =
    welcomeSocket.accept();
```
Agreeing to establish a connection

2-way handshake:

Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side
Agreeing to establish a connection

2-way handshake failure scenarios:

- choose x
- req_conn(x)
- ESTAB
- req_conn(x)
- acc_conn(x)
- connection x completes
- server forgets x
- ESTAB
- half open connection! (no client!)

- choose x
- req_conn(x)
- ESTAB
- req_conn(x)
- acc_conn(x)
- data(x+1)
- client terminates
- ESTAB
- choose x
- req_conn(x)
- ESTAB
- data(x+1)
- accept data(x+1)
- server forgets x
- ESTAB
- accept data(x+1)
TCP 3-way handshake

**client state**

- **LISTEN**: Choose init seq num, x, and send TCP SYN msg.
- **SYNSENT**: Received SYNACK(x), indicating server is live; send ACK for SYNACK; this segment may contain client-to-server data.
- **ESTAB**: Server state

**server state**

- **LISTEN**: Choose init seq num, y, and send TCP SYNACK msg, acking SYN.
- **SYN_RCVD**: Received ACK(y), indicates client is live.

---

Transport Layer 3-24
TCP 3-way handshake: FSM

Socket connectionSocket =
    welcomeSocket.accept();

SYN(seq=x)

create new socket for communication back to client

SYNACK(seq=y,ACKnum=x+1)

ACK(ACKnum=y+1)

Socket clientSocket =
    newSocket("hostname","port number");

SYN(seq=x)
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled
TCP: closing a connection

**client state**

- ESTAB
  - clientSocket.close()
  - FIN_WAIT_1
    - FINbit=1, seq=x
    - can no longer send but can receive data
  - FIN_WAIT_2
    - wait for server close
  - TIMED_WAIT
    - timed wait for 2*max segment lifetime
  - CLOSED

**server state**

- ESTAB
  - CLOSE_WAIT
  - LAST_ACK
    - FINbit=1, seq=y
    - can still send data
    - ACKbit=1; ACKnum=y+1
  - CLOSED

- FIN_WAIT_1
  - ACKbit=1; ACKnum=x+1
  - can no longer send data

Transport Layer 3-27
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!
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: additive increase multiplicative decrease

- approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - additive increase: increase $cwnd$ by 1 MSS every RTT until loss detected
  - multiplicative decrease: cut $cwnd$ in half after loss

AIMD saw tooth behavior: probing for bandwidth
TCP Congestion Control: details

-send sequence number space

 TCP sending rate:
- roughly: send cwnd bytes, wait RTT for ACKS, then send more bytes

rate \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}

- sender limits transmission:
  \begin{align*}
  \text{LastByteSent} - \text{LastByteAcked} & \leq \text{cwnd} \\
  \text{LastByteSent} & \leq \text{cwnd}
  \end{align*}

- \textbf{cwnd} is dynamic, function of perceived network congestion
TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - Initially \( cwnd = 1 \) MSS
  - Double \( cwnd \) every RTT
  - Done by incrementing \( cwnd \) for every ACK received

- **Summary:** initial rate is slow but ramps up exponentially fast
TCP: detecting, reacting to loss

- loss indicated by timeout:
  - $cwnd$ set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - $cwnd$ is cut in half window then grows linearly

- TCP Tahoe always sets $cwnd$ to 1 (timeout or 3 duplicate acks)
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?
A: when \texttt{cwnd} gets to 1/2 of its value before timeout.

Implementation:
- variable \texttt{ssthresh}
- on loss event, \texttt{ssthresh} is set to 1/2 of \texttt{cwnd} just before loss event
Summary: TCP Congestion Control

- **Slow Start**
  - $cwnd = 1$ MSS
  - $ssthresh = 64$ KB
  - $dupACKcount = 0$
  - Retransmit missing segment

- **Fast Recovery**
  - $cwnd = cwnd + MSS$
  - $dupACKcount = 0$
  - Transmit new segment(s), as allowed

- **Congestion Avoidance**
  - $cwnd \geq ssthresh$
  - $cwnd = cwnd + MSS \cdot (MSS/cwnd)$
  - $dupACKcount = 0$
  - Transmit new segment(s), as allowed

- **New ACK**
  - $cwnd = cwnd + MSS$
  - $dupACKcount = 0$
  - Transmit new segment(s), as allowed

- **New ACK**
  - $cwnd = ssthresh$
  - $dupACKcount = 0$
  - Transmit new segment(s), as allowed

- **Duplicate ACK**
  - $dupACKcount = 3$
  - Retransmit missing segment

- **Timeout**
  - $ssthresh = cwnd/2$
  - $cwnd = 1$ MSS
  - $dupACKcount = 0$
  - Retransmit missing segment

- **Retransmit Missing Segment**