Chapter 3
Transport Layer

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 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) in substantially unaltered form, that you mention their source (after all, we’d like people to use our book!)
- If you post any slides in substantially unaltered form 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 1996-2010
J.F Kurose and K.W. Ross, All Rights Reserved

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.
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
  - Bit error: Ack, seq.#
  - Loss: Time out
  - Pipelining
  - Selective Repeat
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
rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver!
- can’t 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
**rdt2.1: sender, handles garbled ACK/NAKs**

- rdt_send(data)
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
  - Wait for call 0 from above
  - Wait for ACK or NAK 0
    - rdt_send(data)
      - sndpkt = make_pkt(1, data, checksum)
      - udt_send(sndpkt)
    - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
      - Wait for call 1 from above
      - Wait for ACK or NAK 1
        - rdt_send(data)
          - sndpkt = make_pkt(2, data, checksum)
          - udt_send(sndpkt)
        - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
        - Wait for call 2 from above
        - Wait for ACK or NAK 2
          - rdt_send(data)
            - sndpkt = make_pkt(3, data, checksum)
            - udt_send(sndpkt)
          - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
          - Wait for call 3 from above
          - Wait for ACK or NAK 3
            - rdt_send(data)
              - sndpkt = make_pkt(4, data, checksum)
              - udt_send(sndpkt)
            - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
            - Wait for call 4 from above
            - Wait for ACK or NAK 4
              - rdt_send(data)
                - sndpkt = make_pkt(5, data, checksum)
                - udt_send(sndpkt)
              - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
              - Wait for call 5 from above
              - Wait for ACK or NAK 5
**rdt2.1: receiver, handles garbled ACK/NAKs**

- \( rdt\_rcv(rcvpkt) \&\& \text{notcorrupt}(rcvpkt) \&\& \text{has_seq0}(rcvpkt) \)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)

- \( rdt\_rcv(rcvpkt) \&\& \text{notcorrupt}(rcvpkt) \&\& \text{has_seq1}(rcvpkt) \)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)

- \( rdt\_rcv(rcvpkt) \&\& \text{corrupt}(rcvpkt) \)
  - sndpkt = make_pkt(NAK, chksum)
  - udt_send(sndpkt)

- \( rdt\_rcv(rcvpkt) \&\& \text{not corrupt}(rcvpkt) \&\& \text{has_seq0}(rcvpkt) \)
  - sndpkt = make_pkt(NAK, chksum)
  - udt_send(sndpkt)

- \( rdt\_rcv(rcvpkt) \&\& \text{not corrupt}(rcvpkt) \&\& \text{has_seq1}(rcvpkt) \)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)
rdt2.1: discussion

**Sender:**
- seq # added to pkt
- two seq. #’s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “current” pkt has 0 or 1 seq. #

**Receiver:**
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
**rdt2.2: sender, receiver fragments**

Sender FSM fragment:
- `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK1, checksum)`
  - `udt_send(sndpkt)`
- `Wait for 0 from below`

Receiver FSM fragment:
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK1, checksum)`
  - `udt_send(sndpkt)`
- `Wait for call 0 from above`
- `Wait for ACK 0`
New assumption: underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits “reasonable” amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
**rdt3.0 sender**

- **Wait for call 0 from above**
  - `rdt_rcv(rcvpkt)`
  - `udt_send(sndpkt)`
  - `start_timer`

- **Wait for ACK0**
  - `rdt_rcv(rcvpkt)`
  - `&& notcorrupt(rcvpkt)`
  - `&& isACK(rcvpkt,1)`
  - `stop_timer`

- **Wait for call 1 from above**
  - `rdt_rcv(rcvpkt)`
  - `udt_send(sndpkt)`
  - `start_timer`

- **Wait for ACK1**
  - `rdt_rcv(rcvpkt)`
  - `&& notcorrupt(rcvpkt)`
  - `&& isACK(rcvpkt,0)`
  - `stop_timer`

- **timeout**
  - `udt_send(sndpkt)`
  - `start_timer`

- **rdt_send(data)**
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`

- **rdt_send(data)**
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`
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

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[ d_{\text{trans}} = \frac{L}{R} = \frac{8000\text{bits}}{10^9 \text{bps}} = 8\text{ microseconds} \]

- \( U_{\text{sender}} \): utilization – fraction of time sender busy sending

\[ U_{\text{sender}} = \frac{L/R}{\text{RTT} + L/R} = \frac{0.008}{30.008} = 0.00027 \]

- if RTT=30 msec, 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
rdt3.0: stop-and-wait operation

*sender*

- first packet bit transmitted, $t = 0$
- last packet bit transmitted, $t = L / R$
- ACK arrives, send next packet, $t = RTT + L / R$

*receiver*

- first packet bit arrives
- last packet bit arrives, send ACK

\[
U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027
\]
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
   - Bit error: Ack, seq.#
   - Loss: Time out
   - Pipelining
   - Selective Repeat
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
Pipelined protocols

**pipelining**: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

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

* two generic forms of pipelined protocols: go-Back-N, selective repeat
Pipelining: increased utilization

First packet bit transmitted, \( t = 0 \)

Last bit transmitted, \( t = \frac{L}{R} \)

First packet bit arrives

Last packet bit arrives, send ACK

Last bit of 2\(^{nd}\) packet arrives, send ACK

Last bit of 3\(^{rd}\) packet arrives, send ACK

\[ U_{\text{sender}} = \frac{3 * \frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{.024}{30.008} = 0.0008 \]

Increase utilization by a factor of 3!
Pipelined 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 unack’ed packets

Selective Repeat: big pic
- sender can have up to N unack’ed packets in pipeline
- rcvr sends individual ack for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only unack’ed packet
Go-Back-N

Sender:
- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window
GBN: sender extended FSM

\[
\begin{align*}
\text{rdt\_send(} & \text{data)} \\
\text{if (nextseqnum < base+N)} \{
& \text{sndpkt[} \text{nextseqnum} \text{] = make pkt} \text{(nextseqnum, data, checksum)} \\
& \text{udt\_send(sndpkt[} \text{nextseqnum} \text{])} \\
& \text{if (base == nextseqnum)} \\
& \text{start\_timer} \\
& \text{nextseqnum++} \\
\} \\
\text{else} \\
& \text{refuse\_data(data)} \\
& \text{base = getacknum(rcvpkt) +1} \\
& \text{If (base == nextseqnum)} \\
& \text{stop\_timer} \\
& \text{else} \\
& \text{start\_timer}
\end{align*}
\]
GBN: receiver extended FSM

ACK-only: always send ACK for correctly-received pkt with highest \textit{in-order} seq #
- may generate duplicate ACKs
- need only remember expectedseqnum

- \textbullet{} out-of-order pkt:
  - discard (don’t buffer) -> \textcolor{red}{no receiver buffering}!
  - Re-ACK pkt with highest in-order seq #
GBN in action

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

receiver

rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3