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
rdt3.0: channels with errors and loss

**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**

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

- **rdt_rcv(rcvpkt) &\& isACK(rcvpkt,0)**
  - stop_timer

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

- **rdt_rcv(rcvpkt) &\& notcorrupt(rcvpkt) &\& isACK(rcvpkt,1)**
  - stop_timer

- **Wait for call 0 from above**

- **Wait for call 1 from above**

- **Wait for ACK0**
  - rdt_rcv(rcvpkt) &\& notcorrupt(rcvpkt) &\& isACK(rcvpkt,0)
  - udt_send(sndpkt)
  - start_timer

- **Wait for ACK1**
  - rdt_rcv(rcvpkt)
  - (corrupt(rcvpkt) || isACK(rcvpkt,0))
  - \(\Lambda\)

- **timeout**
  - \(\Lambda\)

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

- **rdt_rcv(rcvpkt) &\& (corrupt(rcvpkt) || isACK(rcvpkt,1))**
  - \(\Lambda\)
rdt3.0 in action

(a) operation with no loss

(b) lost packet
**rdt3.0 in action**

- ** sender **
  - send pkt0
  - rcv pkt0
  - ACK 0
  - send ACK0
  - timeout
  - resend pkt1
  - rcv pkt1
  - (loss) X
  - rcv ACK1
  - send ACK1
  - rcv pkt1

- ** receiver **
  - pkt 0
  - ACK 0
  - send ACK0
  - pkt 1
  - ACK 1
  - send ACK1
  - pkt 1
  - ACK 1
  - send ACK1

(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

\[ U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{.008}{30.008} = 0.00027 \]

- 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

\[
U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{.008}{30.008} = 0.00027
\]
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*

(a) a stop-and-wait protocol in operation  (b) a pipelined protocol in operation
Pipelining: increased utilization

\[
U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008
\]

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

```plaintext
rdt_send(data)
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
        nextseqnum++
    }
else
    refuse_data(data)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
else
    start_timer

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
else
    start_timer
```
**GBN: receiver extended FSM**

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

- out-of-order pkt:
  - discard (don’t buffer) -> 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 pkt0
- Send pkt4
- Rcv pkt1
- 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