A note on the use of these Powerpoint 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 1996-2016
© J.F Kurose and K.W. Ross, All Rights Reserved
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
new assumption: underlying channel can also lose packets (data, 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 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)
  - Wait for call 0 from above

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

- rdt_rcv(rcvpkt)
  - && isACK(rcvpkt,0)
  - rdt_send(data)
    - sndpkt = make_pkt(1, data, checksum)
    - udt_send(sndpkt)
    - start_timer

- rdt_rcv(rcvpkt) &&
  - ( corrupt(rcvpkt) ||
  - isACK(rcvpkt,1) )
  - \[ \Lambda \]

- rdt_rcv(rcvpkt)
  - && notcorrupt(rcvpkt)
  - && isACK(rcvpkt,0)
  - stop_timer

- rdt_rcv(rcvpkt)
  - && isACK(rcvpkt,1)
  - stop_timer

- udt_send(sndpkt)
  - start_timer

- timeout

- Wait for ACK0

- Wait for call 1 from above

- rdt_rcv(rcvpkt)
  - \[ \Lambda \]
rdt3.0 in action

(a) no loss

(b) packet loss
rdt3.0 in action

(c) ACK loss

(d) premature timeout/ delayed ACK
Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[ D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \]

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

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

- 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**

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = \frac{L}{R}$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + \frac{L}{R}$

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

3-packet pipelining increases utilization by a factor of 3!

\[ U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{0.0024}{30.008} = 0.00081 \]
Pipelined protocols: overview

Go-back-N:
- sender can have up to N unacked packets in pipeline
- receiver only sends cumulative ack
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit all unacked packets

Selective Repeat:
- 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 that unacked packet
Go-Back-N: sender

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

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

- rdt_send(data)
- if (nextseqnum < base+N) {
  sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)
  udt_send(sndpkt[nextseqnum])
  if (base == nextseqnum)
    start_timer
    nextseqnum++
  }
  else
    refuse_data(data)

- 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 #

\[
\begin{align*}
\Lambda & \quad \text{udt_send(sndpkt)} \\
\text{default} & \\
\text{expectedseqnum=1} & \\
\text{sndpkt} = & \quad \text{make_pkt(0,ACK,chksum)}
\end{align*}
\]

\[
\begin{align*}
\text{Wait} & \quad \text{rdt_rcv(rcvpkt)} \\
& \quad \& \& \text{notcurrupt(rcvpkt)} \\
& \quad \& \& \text{hasseqnum(rcvpkt,expectedseqnum)} \\
\text{extract(rcvpkt,data)} & \\
\text{deliver_data(data)} & \\
\text{sndpkt} = & \quad \text{make_pkt(expectedseqnum,ACK,chksum)} \\
\text{udt_send(sndpkt)} & \\
\text{expectedseqnum} & \quad \text{++}
\end{align*}
\]
**GBN in action**

**sender window (N=4)**

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt0</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt2</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt3 (wait)</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>receive pkt0, send ack0</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>receive pkt1, send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>receive pkt3, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>receive pkt4, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>receive pkt5, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>ignore duplicate ACK</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv pkt2 timeout</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv pkt2, deliver, send ack2</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv pkt3, deliver, send ack3</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv pkt4, deliver, send ack4</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv pkt5, deliver, send ack5</td>
</tr>
</tbody>
</table>
Selective repeat

• receiver individually acknowledges all correctly received pkts
  • buffers pkts, as needed, for eventual in-order delivery to upper layer

• sender only resends pkts for which ACK not received
  • sender timer for each unACKed pkt

• sender window
  • $N$ consecutive seq #’s
  • limits seq #s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Transport Layer 3-17
Selective repeat

**sender**

Data from above:
- If next available seq # in window, send pkt

Timeout(n):
- Resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:
- Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

**receiver**

Pkt n in [rcvbase, rcvbase+N-1]:
- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

Pkt n in [rcvbase-N,rcvbase-1]:
- ACK(n)

Otherwise:
- Ignore
Selective repeat in action

**sender window (N=4)**

```
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
```

**sender**

- send pkt0
- send pkt1
- send pkt2 (wait)
- send pkt3

**receiver**

- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, buffer, send ack3
- receive pkt4, buffer, send ack4
- receive pkt5, buffer, send ack5

- record ack3 arrived
- record ack4 arrived
- record ack5 arrived

**pkt 2 timeout**

```
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
```

**Q: what happens when ack2 arrives?**
Selective repeat: dilemma

example:
- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?