Chapter III: Transport Layer

UG3 Computer Communications & Networks (COMN)

Myungjin Lee
myungjin.lee@ed.ac.uk

Slides copyright of Kurose and Ross
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 corrupted
• 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) && (corrupt(rcvpkt) || isNAK(rcvpkt))` → `udt_send(sndpkt)`
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)` → `Λ`
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))` → `udt_send(sndpkt)`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)` → `Λ`
rdt2.1: receiver, handles garbled ACK/NAKs

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) 
& & has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
& & has_seq1(rcvpkt)

sdnpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, checksum)
udt_send(sndpkt)

Wait for 0 from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) 
& & has_seq1(rcvpkt)

sdnpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

Wait for 1 from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) 
& & has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, checksum)
udt_send(sndpkt)
Example 1

\[\text{rdt\_send(data)}\]
\[\text{sndpkt = make\_pkt(0, data, checksum)}\]
\[\text{udt\_send(sndpkt)}\]

Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

Wait for 0 from below

Wait for 1 from below
Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 0 from above

Wait for ACK or NAK 1

Wait for call 1 from above

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)

udt_send(sndpkt)
**rdt2.1 Example 1**

```
wait for 0 from below
wait for call 0 from above
wait for ACK or NAK 0

wait for 1 from below
wait for call 1 from above
wait for ACK or NAK 1

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) || isNAK(rcvpkt))
```

33
rdt2.1 Example 1

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt2.1 Example 1

Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)

Λ

Wait for 0 from below

Wait for 1 from below
rdt2.1 Example 1

Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1
rdt2.1 Example 2

Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Wait for 0 from below

Wait for 1 from below
rdt2.1 Example 2

- Wait for call 0 from above
- Wait for ACK or NAK 0
- Wait for call 1 from above
- Wait for ACK or NAK 1

```
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))
udt_send(sndpkt)
```
rdt2.1 Example 2

Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

Wait for 0 from below

Wait for 1 from below

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

\[ \text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt}) \]

Wait for 0 from below

Wait for 1 from below
Example 2

Wait for call 0 from above
Wait for ACK or NAK 0
Wait for call 0 from below
Wait for ACK or NAK 1

Wait for call 1 from above
Wait for call 1 from below

Wait for 0 from below
Wait for 1 from below
**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 “expected” pkt should have seq # of 0 or 1

**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) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))
udt_send(sndpkt)
```

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

L
**rdt3.0: channels with errors and loss**

**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)** &
  - (corrupt(rcvpkt) || isACK(rcvpkt,1))
- **Lambda**

- **Wait for call 0 from above**
- **Wait for ACK0**
- **Wait for call 1 from above**
- **Wait for ACK1**

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

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

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

- **rdt_rcv(rcvpkt)** &
  - (corrupt(rcvpkt) || isACK(rcvpkt,0))
- **Lambda**

- **rdt_send(data)**
  - `sndpkt = make_pkt(1, data, checksum)`
  - ```
      udt_send(sndpkt)
    ```
  - **start_timer**
**rdt3.0 in action**

(a) no loss

**sender**
- send pkt0
- rcv ack0
- send pkt1
- rcv ack1
- send pkt0

**receiver**
- pkt0
- ack0
- pkt1
- ack1
- pkt0
- ack0

(b) packet loss

**sender**
- send pkt0
- rcv ack0
- send pkt1
- timeout
- resend pkt1
- rcv ack1
- send pkt0
- pkt1

**receiver**
- send pkt0
-_rcv pkt0
- send ack0
- pkt0
- ack0

**sender**
- send pkt0
- rcv pkt0
- send ack0

**receiver**
- send pkt0
- rcv pkt0
- send ack0
rdt3.0 in action

**sender**
- send pkt0
- rcv pkt0
- send ack0
- rcv ack0
- send pkt1
- rcv pkt1
- send ack1
- rcv ack1
- send pkt0
- rcv pkt0
- send ack0

**receiver**
- send pkt0
- rcv pkt0
- send ack0
- rcv ack0
- send pkt1
- rcv pkt1
- send ack1
- rcv ack1

(c) ACK loss
- timeout
  - resend pkt1
  - rcv pkt1
  - send ack1
  - rcv ack1
  - do nothing

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

  - \( U_{\text{sender}} \): utilization – fraction of time sender busy sending
    \[ U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027 \]

  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput 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, $t = L / R$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$

$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027$
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

- first packet bit transmitted, $t = 0$
- last bit transmitted, $t = L / R$
- first packet bit arrives
- last bit of 2\textsuperscript{nd} packet arrives, send ACK
- last bit of 3\textsuperscript{rd} packet arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$
- last packet bit arrives, send ACK

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

$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$
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

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

\[ \text{base} = 1 \]
\[ \text{nextseqnum} = 1 \]
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)
base = getacknum(rcvpkt) + 1
If (base == nextseqnum)
    stop_timer
else
    start_timer
```
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 #
ACK-only: always send ACK for correctly-received pkt with highest \textit{in-order} seq \#  
\begin{itemize}
  \item may generate duplicate ACKs
  \item need only remember expectedseqnum
\end{itemize}

• out-of-order pkt:  
\begin{itemize}
  \item discard (don’t buffer): \textit{no receiver buffering!}
  \item re-ACK pkt with highest in-order seq \#
\end{itemize}
GBN in action

sender window \( (N=4) \)  

\begin{align*}
\text{sender} & \quad \text{receiver} \\
\text{send } \text{pkt0} & \quad \text{receive pkt0, send ack0} \\
\text{send } \text{pkt1} & \quad \text{receive pkt1, send ack1} \\
\text{send } \text{pkt2} & \quad \text{receive pkt3, discard, } (\text{re})\text{send ack1} \\
\text{send } \text{pkt3} & \quad \text{rcv ack0, send pkt4} \\
\text{send } \text{pkt4} & \quad \text{rcv ack1, send pkt5} \\
\text{send } \text{pkt5} & \quad \text{rcv pkt2, deliver, send ack2} \\
\end{align*}
GBN in action

**sender window (N=4)**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**sender**

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

**receiver**

- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, discard,
  (re)send ack1
- receive pkt4, discard,
  (re)send ack1
- receive pkt5, discard,
  (re)send ack1
- rcv pkt2, deliver, send ack2
- rcv pkt3, deliver, send ack3
- rcv pkt4, deliver, send ack4
- rcv pkt5, deliver, send ack5

**pkt 2 timeout**

- ignore duplicate ACK

**loss**

- x
Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender only resends packets for which ACK not received
  - sender timer for each unACKed packet
- sender window
  - $N$ consecutive seq #’s
  - limits seq #s of sent, unACKed packets
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
## 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-1]:**
- 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 \((N=4)\)  

sender  

receiver

Q: what happens when ack2 arrives?
Selective repeat in action

Sender window (N=4)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2
- record ack3 arrived
- record ack4 arrived
- record ack5 arrived

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)?

receiver can’t see sender side. receiver behavior identical in both cases! something’s (very) wrong!

will accept packet with seq number 0

(a) no problem

(b) oops!

will accept packet with seq number 0