CSC358 Intro. to Computer Networks

Lecture 6: Reliable data transfer: rdt2.2, rdt3.0, GBN, and SR


ahchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~ahchinaei/

Many slides are (inspired/adapted) from the above source
© all material copyright; all rights reserved for the authors

Office Hours: T 17:00–18:00 R 9:00–10:00 BA4222

TA Office Hours: W 16:00-17:00 BA3201 R 10:00-11:00 BA7172
csc358ta@cdf.toronto.edu
Reliable data transfer: getting started

**send side**

- **rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**receive side**

- **deliver_data()**: called by rdt to deliver data to upper
- **rdt_rcv()**: called when packet arrives on rcv-side of channel
rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

receiver

- Wait for call from above
- rdt_send(data)
- packet = make_pkt(data)
- udt_send(packet)

- Wait for call from below
- rdt_rcv(packet)
- extract (packet, data)
- deliver_data(data)
underlying channel may flip bits in packet
- checksum to detect bit errors

the question: how to recover from errors:
- **acknowledgements (ACKs):** receiver explicitly tells sender that pkt received OK
- **negative acknowledgements (NAKs):** receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK

new mechanisms in rdt2.0 (beyond rdt1.0):
- error detection
- feedback: control msgs (ACK, NAK) from receiver to sender
rdt2.0: FSM specification

sender

wait for call from above

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

wait for ACK or NAK

\[ \text{rdt\_recv(rcvpkt) \&\& isNAK(rcvpkt)} \]
\[ \text{udt\_send(sndpkt)} \]

\[ \text{rdt\_recv(rcvpkt) \&\& isACK(rcvpkt)} \]
\[ \Lambda \]

receiver

wait for call from below

\[ \text{rdt\_recv(rcvpkt) \&\& notcorrupt(rcvpkt)} \]
\[ \text{extract(rcvpkt, data)} \]
\[ \text{deliver\_data(data)} \]
\[ \text{udt\_send(ACK)} \]

\[ \text{udt\_send(NAK)} \]
rdt2.0: operation with no errors

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

Wait for call from above

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`

Wait for ACK or NAK

- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`

Wait for call from below

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

- `udt_send(NAK)`
rdt2.0: error scenario

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
udt_send(NAK)

Wait for call from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
**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)

Wait for call 0 from above

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

Lambda

Wait for ACK or NAK 0

rdt_send(data) || (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)

Lambda

Wait for call 1 from above

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

Lambda

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```
Wait for 0 from below

Wait for 1 from below

\[
\text{rdt}_{\text{rcv}}(\text{rcvpkt}) \&\& \text{notcorrupt}(\text{rcvpkt}) \\
&\& \text{has_seq0}(\text{rcvpkt}) \\
\underline{\text{extract}(\text{rcvpkt}, \text{data})} \\
\underline{\text{deliver_data}(\text{data})} \\
\text{sndpkt} = \text{make_pkt}(\text{ACK, checksum}) \\
\text{udt_send}(\text{sndpkt})
\]

\[
\text{rdt}_{\text{rcv}}(\text{rcvpkt}) \&\& \text{notcorrupt}(\text{rcvpkt}) \\
&\& \text{has_seq1}(\text{rcvpkt}) \\
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver_data}(\text{data}) \\
\text{sndpkt} = \text{make_pkt}(\text{ACK, checksum}) \\
\text{udt_send}(\text{sndpkt})
\]

\[
\text{rdt}_{\text{rcv}}(\text{rcvpkt}) \&\& \text{notcorrupt}(\text{rcvpkt}) \\
&\& \text{has_seq1}(\text{rcvpkt})
\]

\[
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver_data}(\text{data}) \\
\text{sndpkt} = \text{make_pkt}(\text{ACK, checksum}) \\
\text{udt_send}(\text{sndpkt})
\]

\[
\text{rdt}_{\text{rcv}}(\text{rcvpkt}) \&\& \text{notcorrupt}(\text{rcvpkt}) \\
&\& \text{has_seq0}(\text{rcvpkt}) \\
\text{sndpkt} = \text{make_pkt}(\text{NAK, checksum}) \\
\text{udt_send}(\text{sndpkt})
\]

\[
\text{rdt}_{\text{rcv}}(\text{rcvpkt}) \&\& \text{notcorrupt}(\text{rcvpkt}) \\
&\& \text{has_seq1}(\text{rcvpkt}) \\
\text{sndpkt} = \text{make_pkt}(\text{NAK, checksum}) \\
\text{udt_send}(\text{sndpkt})
\]

\[
\text{rdt}_{\text{rcv}}(\text{rcvpkt}) \&\& \text{notcorrupt}(\text{rcvpkt}) \\
&\& \text{has_seq0}(\text{rcvpkt}) \\
\text{sndpkt} = \text{make_pkt}(\text{NAK, checksum}) \\
\text{udt_send}(\text{sndpkt})
\]
**rdt2.1: summary**

**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) ||
  isACK(rcvpkt,1) )
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
```

```
udt_send(sndpkt)
```

receiver FSM fragment

```
rdt_rcv(rcvpkt) &&
  has_seq1(rcvpkt))
udt_send(sndpkt)
```

```
wait for 0 from below
```

```
extract(rcvpkt, data)
deliver_data(data)
```

```
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)
```

sender FSM fragment

wait for call 0 from above

```
udt_send(sndpkt)
```
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

- **Wait for call 0 from above**
  - `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `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_send(data)`
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`

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

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

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

(a) no loss

(b) packet loss
rdt3.0 in action

Sender

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

Receiver

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

(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:
  \[ D_{trans} = ? \]
  - \( U_{sender} \): utilization – fraction of time sender busy sending
    \[ U_{sender} = ? \]
  - 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, \( t = RTT + \frac{L}{R} \)

\[
U_{sender} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027
\]
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, \( t = \frac{L}{R} \)

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

ACK arrives, send next packet, \( t = RTT + \frac{L}{R} \)

\[ U_{\text{sender}} = \frac{\frac{3L}{R}}{RTT + \frac{L}{R}} = \frac{0.0024}{30.008} = 0.00081 \]

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

![Diagram showing Go-Back-N sender window]

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- 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)
```
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember $\text{expectedseqnum}$

- out-of-order pkt:
  - discard (don’t buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #
**GBN 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
- 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

**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

- send pkt2
- send pkt3
- send pkt4
- send pkt5

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

**ignore duplicate ACK**

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
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 3-28
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)

<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)
send pkt3

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3
rcv ack0, send pkt4
rcv ack1, send pkt5
record ack3 arrived
pkt 2 timeout
send pkt2
record ack4 arrived
record ack5 arrived
rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2
receive pkt4, buffer,
send ack4
receive pkt5, buffer,
send ack5
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)?
Next

- Midterm on Chapters 1 and 2
- TCP
- Congestion control