CS 4284
Systems Capstone
Networking
Godmar Back
Principles of Reliable Data Transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
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**

- `rdt_rcv()` : called when packet arrives on rcv-side of channel.
- `deliver_data()` : called by rdt to deliver data to upper.

- `rdt_rcv()` : called when packet arrives on rcv-side of channel.
- `deliver_data()` : called by rdt to deliver data to upper.
Upcalls vs Downcalls

- Consumer/Producer pattern for receive()
- `deliver_data()` is an upcall from k to k+1
  - data producer, coordinates with data consumer at *application layer*
  or
- “`receive_data()`” as downcall from k+1 to k
  - data consumer, coordinates with data producer in *transport layer*
  - don’t confuse with `rdt_rcv()` upcall from k-1 to k
- Both choices are possible/acceptable for k+1/k interface in project:
  - difference is who deals with concurrency + signaling
- You don’t have a choice for k/k-1 interface:
  - UDP socket API doesn’t (usually) support callbacks
Reliable Data Transfer: getting started

We’ll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

**State:** when in this “state” next state uniquely determined by next event

**Event causing state transition**

**Actions taken on state transition**
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 read data from underlying channel

sender

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

receiver

```
Wait for call from below
```  
```
rdt_rcv(packet)
extract (packet, data)
```  
```
deliver_data(data)
```
rdt2.0: Channel with Bit Errors

- 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
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender
  - retransmission
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

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

\[ \text{rdt\_rcv(rcvpkt)} \land \text{isACK(rcvpkt)} \]

receiver

Wait for call from below

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

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

\[
\begin{align*}
\text{rdt\_send(data)} & \\
nkpkt = \text{make\_pkt(data, checksum)} & \\
\text{udt\_send(sndpkt)} & \\
\text{rdt\_rcv(rcvpkt) \&\& isNAK(rcvpkt)} & \\
\text{udt\_send(sndpkt)} & \\
\text{rdt\_rcv(rcvpkt) \&\& isACK(rcvpkt)} & \\
\end{align*}
\]
**rdt2.0: error scenario**

- `rdt_send(data)`
  - `snkpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
  - `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `wait for call from below`
  - `rdt_send(data)`
  - `snkpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`
- `wait for call from above`
  - `wait for ACK or NAK`
  - `wait for call from below`
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 adds sequence number to each pkt
• sender retransmits current pkt if ACK/NAK garbled
• receiver discards (doesn’t deliver up) duplicate pkt

Sender sends one packet, then waits for receiver response
rdt2.1: Sender FSM, w/ garbled ACK/NAKs

- Send packet:
  - `sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - rdt_send(data)

- Wait for ACK or NAK 0
  - `udt_send(sndpkt)`
  - `rdt_rcv(rcvpkt) && ( corrupt(rcvpkt) || isNAK(rcvpkt))`

- Reread packet:
  - `udt_send(sndpkt)`

- Wait for call 0 from above

- Send packet:
  - `sndpkt = make_pkt(1, data, checksum)
  - udt_send(sndpkt)
  - rdt_send(data)

- Wait for ACK or NAK 1
  - `udt_send(sndpkt)`

- Wait for call 1 from above

- Send packet:
  - `sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - rdt_send(data)"
rdt2.1: Receiver, w/ garbled ACK/NAKs

\[
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \text{not} \text{corrupt}(\text{rcvpkt}) \\
\land \text{has}_\text{seq}0(\text{rcvpkt})
\]

\[
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver}_\text{data}(\text{data}) \\
\text{sndpkt} = \text{make}_\text{pkt}(\text{ACK}, \text{chksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\]

\[
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \text{not} \text{corrupt}(\text{rcvpkt}) \\
\land \text{has}_\text{seq}1(\text{rcvpkt})
\]

\[
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver}_\text{data}(\text{data}) \\
\text{sndpkt} = \text{make}_\text{pkt}(\text{ACK}, \text{chksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\]

\[
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \text{not} \text{corrupt}(\text{rcvpkt}) \\
\land \text{has}_\text{seq}1(\text{rcvpkt})
\]

\[
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver}_\text{data}(\text{data}) \\
\text{sndpkt} = \text{make}_\text{pkt}(\text{ACK}, \text{chksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\]
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

Wait for call 0 from above

```
rdt_send(data)
```

```
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
```

Wait for ACK 0

```
rdrv_send(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,1) )
```

```
udt_send(sndpkt)
```

receiver FSM fragment

Wait for 0 from below

```
rdrv_send(sndpkt)
```

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

```
udt_send(sndpkt)
```

```
rdrv_send(rcvpkt) &&
notcorrupt(rcvpkt) &&
has_seq1(rcvpkt)
```

```
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```

CS 4284 Spring 2013
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 FSM

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

rdt_rcv(rcvpkt)
\Lambda

Wait for call 0 from above

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

Wait for ACK0

rdt_send(data)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
\Lambda

timeout udt_send(sndpkt)
start_timer

Wait for ACK1

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

Wait for call 1 from above

rdt_send(data)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
\Lambda

timeout udt_send(sndpkt)
start_timer

\Lambda
```

CS 4284 Spring 2013
rdt3.0 in action

(a) operation with no loss

(b) lost packet

sender

rcv pkt0

ACK0

send pkt1

rcv ACK0

sender

rcv pkt0

ACK0

send pkt1

rcv pkt1

ACK1

send pkt0

rcv ACK1

sender

send pkt0

rcv pkt1

ACK1

timeout

resend pkt1

rcvACK1

sender

send pkt0

rcv pkt0

ACK0

rcv pkt0

ACK0

rcv pkt0

ACK0

rcv pkt0

ACK0

sender

rcv pkt0

ACK0

rcv(pkt1)

X(loss)

rcv pkt1

ACK1

rcv pkt0

ACK0

rcv pkt0

ACK0
rdt3.0 in action (2)

(c) lost ACK

(d) premature timeout
Evolution of Assumptions

- **rdt1.0**
  - Perfect channel
  - Bit errors

- **rdt2.0**
  - Error detection: checksums, receiver feedback: acknowledgements; retransmissions
  - Automatic Repeat reQuest (ARQ protocol)
  - (Redundant) retransmissions → duplicates → sequence numbers
    - 0/1: “alternating bit” protocol

- **rdt2.1**
  - Corrupted acks

- **rdt2.2**
  - Eliminate NAK: use ACK+seq num

- **rdt2.3**
  - Timeouts

- **rdt3.0**
  - Lossy channel

- **rdt3.1**
  - Lossy channel with bit errors

• **rdt1-3** are stop-and-wait protocols
Performance of rdt3.0

• rdt3.0 works, but performance stinks
• example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[
T_{\text{transmit}} = \frac{L}{R} = \frac{8\text{kb/pkt}}{10^{\Phi \Phi 9} \text{ b/sec}} = 8 \text{ microsec}
\]

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

– \(U_{\text{sender}}\): utilization – fraction of time sender busy sending
– 1KB pkt every 30 msec \(\rightarrow\) 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, $t = L / R$

- **receiver**
  - First packet bit arrives
  - Last packet bit arrives, send ACK

- ACK arrives, send next packet, $t = RTT + L / R$

**Equation:**

$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$
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 (“GBN”), Selective Repeat (“SR”)
Pipelining: Increased Utilization

first packet bit transmitted, $t = 0$
last bit transmitted, $t = L / R$

ACK arrives, send next packet, $t = RTT + L / R$

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

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

Increase utilization by a factor of 3!
Go-Back-N

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

```
<table>
<thead>
<tr>
<th>send_base</th>
<th>nextseqnum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>window size</td>
<td>N</td>
</tr>
</tbody>
</table>
```

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- **timeout(n):** retransmit pkt n and all higher seq # pkts in window
GBN: Sender
Extended FSM

\[\text{rsd\_send}(\text{data})\]
\[
\text{if} \ (\text{nextseqnum} < \text{base+N}) \ {\}
\quad \text{sndpkt[nextseqnum]} = \text{make\_pkt}(\text{nextseqnum}, \text{data}, \text{chksum})
\quad \text{udt\_send(sndpkt[nextseqnum])}
\quad \text{if} \ (\text{base} == \text{nextseqnum})
\quad \quad \text{start\_timer}
\quad \quad \text{nextseqnum}++
\}
\text{else}
\quad \text{refuse\_data(data)}
\]

\[\text{timeout}\]
\[\text{start\_timer}\]
\[\text{udt\_send(sndpkt[base])}\]
\[\text{udt\_send(sndpkt[base+1])}\]
\[\ldots\]
\[\text{udt\_send(sndpkt[\text{nextseqnum-1}])}\]

\[\text{rdr\_rcv(rcvpkt) && corrupt(rcvpkt)}\]

\[\text{base} = \text{getacknum(rcvpkt)}+1\]
\[\text{If} \ (\text{base} == \text{nextseqnum})\]
\quad \text{stop\_timer}
\quad \text{else}
\quad \quad \text{start\_timer}

CS 4284 Spring 2013
ACK-only: always send ACK for correctly-received pkt with highest \textit{in-order} seq #

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

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

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)

receiver

rcv pkt0
send ACK0
rcv pkt1
send ACK1

rcv pkt3, discard
send ACK1

rcv pkt4, discard
send ACK1

rcv pkt5, discard
send ACK1

pkt2 timeout

send pkt2
send pkt3
send pkt4
send pkt5

rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

CS 4284 Spring 2013
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
  – again limits seq #s of sent, unACKed pkts
Selective Repeat: Sender, Receiver Windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

CS 4284 Spring 2013
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
SR in action

pkt0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 sent
0 1 2 3 4 5 6 7 8 9

pkt2 sent
0 1 2 3 4 5 6 7 8 9

pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9

X
(loss)

pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

CS 4284 Spring 2013
Selective Repeat & Window Sizes

Example:
- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in these two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship is required between seq # size and window size?
The Problem With Reordering

- Alternating Bit Protocol is special case of GBN(1)

### Diagram

- **Sender**
  - retransmits Pkt0
  - sends Pkt1
  - retransmitted Pkt0 is overtaken by Pkt1

- **Receiver**
  - Pkt0
  - Pkt1
  - Ack0
  - Ack1
  - Pkt0

**Error:** receiver expects Pkt0, and interprets delayed retransmitted Pkt0 as containing new data!
Reordering

• What if packets can be reordered by underlying layer?
  – Does not usually happen for RDT in data link layer
  – But very possible for RDT over network layer

• Model as loss + spontaneous reemission of first packet by channel

• Solution: do not reuse sequence numbers unless there’s reasonable guarantee that packets with old numbers cannot reemerge
Summary

- Sliding Window Protocols
  - GBN Go-Back-N
  - SR Selective Repeat
  - Alternating-Bit-Protocol = special case of GBN(1)

- Problems:
  - Receiver’s view may not match sender’s view: sequence number range vs window size: seqlrange >= 2 * wsize (assuming no reordering)
  - When channel can reorder: need additional timing constraints before reusing sequence numbers
    - No 100.000% perfect solution possible