Chapter 3: Roadmap

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
Principles of Reliable Data Transfer

- Important in application, transport, link layers

- Characteristics of unreliable channel will determine complexity of reliable data transfer (rdt) protocol
Reliable Data Transfer: Getting Started

- **send side**: 
  - `rdt_send()`: called by layer above to pass data to `rdt`
  - `udt_send()`: called by `rdt` to pass packets to lower layer

- **receive side**: 
  - `deliver_data()`: called by `rdt` to deliver data to upper layer
  - `rdt_rcv()`: called by lower layer when it has a packet to deliver to `rdt`
Reliable Data Transfer: Getting Started

We will:

- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - With receiver feedback, packets travel in both directions!
- Use **finite state machines** (FSM) to specify both sender and receiver

- From any state, the next state is uniquely determined by next event

```
state 1
  ───[event]─→ state 2
```
Rdt1.0: Transfer Over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - No reordering

- Separate FSMs for sender and receiver:
  - Sender transmits app data into underlying channel
  - Receiver passes data from underlying channel to app

sender

receiver

\[
\text{packet} = \text{make(pkt(data))}
\]

\[
\text{udt}\text{.send(packet)}
\]

\[
\text{rdt}\text{.send(data)}
\]

\[
\text{Wait for call from above}
\]

\[
\text{rdt}\text{.rcv(packet)}
\]

\[
\text{extract (packet, data)}
\]

\[
\text{deliver_data(data)}
\]
Rdt2.0: Channel With Bit Errors

• Underlying channel may flip bits in packet (no loss)
  – Checksum to detect bit errors (assume perfect detection)

• Question: how to recover from errors?

• One possible approach is to use two feedback msgs:
  – Positive acknowledgments (ACKs): receiver explicitly tells sender that packet was received OK
  – Negative acknowledgments (NAKs): receiver explicitly tells sender that packet had errors
    – Sender retransmits packet on receipt of NAK

• New mechanisms in rdt 2.0 (beyond rdt 1.0):
  – Error detection
  – Receiver feedback (control msgs ACK/NAK)
  – Retransmission
**Rdt2.0: FSM Specification**

**Send**

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

**Receive**

- `rdt_rcv(rcvpkt)` AND `corrupt(rcvpkt)`
  - `udt_send(NAK)`

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

- `rdt_rcv(rcvpkt)` AND `isACK(rcvpkt)`
  - `Λ`

- `Λ` = empty action, i.e., do nothing
Rdt2.0: Operation With No Errors

```
wait for call from above

snkpkt = make_pkt(data, checksum)
udt_send(snkpkt)

wait for ACK or NAK

rdt_send(data)

wait for call from below

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(snkpkt)

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

```
wait for call from above

rdt_rcv(rcvpkt) && isACK(rcvpkt)

udt_send(snkpkt)

notcorrupt(rcvpkt)
```

```
correct(rcvpkt)
```

```
wait for call from below
```

```
rdf_send(NAK)
```

```
wait for call from above
```

```
rdf_send(ACK)
```

```
rdf_send(NAK)
```

```
rdf_send(ACK)
```
Rdt2.0: Error Scenario

rdt_send(data)

\[ \text{snkpkt} = \text{make_pkt(data, checksum)} \]
udt_send(sndpkt)

\[ \text{rdt_rcv(rcvpkt)} && \text{isNAK(rcvpkt)} \]
udt_send(sndpkt)

\[ \text{rdt_send(sndpkt)} \]

\[ \text{rdt_send(NAK)} \]

\[ \text{rdt_send(ACK)} \]

Wait for call from above

Wait for ACK or NAK

\[ \text{rdt_rcv(rcvpkt)} && \text{isACK(rcvpkt)} \]

\[ \Lambda \]

Wait for call from below

Any problems with this protocol?
Rdt2.0a: Handles Corrupted Feedback

1. \( \text{rdt\_send(data)} \)
2. \( \text{snkpkt = make\_pkt(data, checksum)} \)
3. \( \text{udt\_send(sndpkt)} \)

**Sender**
- \( \text{rdt\_rcv(rcvpkt) AND isACK(rcvpkt) AND NOT corrupt(rcvpkt)} \)
- \( \Lambda \)

**Receiver**
- \( \text{rdt\_rcv(rcvpkt) AND corrupt(rcvpkt)} \)
- \( \text{udt\_send(NAK, checksum)} \)
- \( \text{Wait for call from below} \)
- \( \text{extract(rcvpkt, data)} \)
- \( \text{deliver\_data(data)} \)
- \( \text{udt\_send(ACK, checksum)} \)

Any problems?
Rdt2.0 and Rdt2.0a Have Fatal Flaws

- Rdt 2.0 does not work when ACK/NAK is corrupted
  - Sender doesn’t know what happened at receiver!
- Rdt 2.0a delivers duplicate packets to application

Proper algorithm:
- Sender adds sequence number to each pkt
- Sender retransmits current pkt if ACK/NAK is garbled
- Receiver discards (doesn’t deliver up) duplicate pkt

Stop-and-Wait protocol: sender sends one packet, then waits for receiver’s 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

Wait for ACK or NAK 0

rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt)
Λ

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_rcv(rcvpkt) AND [corrupt(rcvpkt) OR isNAK(rcvpkt)]
udt_send(sndpkt)

rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) AND
[corrupt(rcvpkt) OR isNAK(rcvpkt)]
udt_send(sndpkt)
Rdt2.1: Receiver, Handles Garbled ACK/NAKs

Rdt2.1: Receiver, Handles Garbled ACK/NAKs

Wait for 0 from below

- $\text{rdt}_\text{rcv}(\text{rcvpkt})$ AND NOT corrupt($\text{rcvpkt}$) AND has_seq0($\text{rcvpkt}$)
  - extract($\text{rcvpkt}$, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)

Wait for 1 from below

- $\text{rdt}_\text{rcv}(\text{rcvpkt})$ AND corrupt($\text{rcvpkt}$)
  - sndpkt = make_pkt(NAK, chksum)
  - udt_send(sndpkt)

Rdt2.1: Receiver, Handles Garbled ACK/NAKs
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
  - Protocol must remember whether current pkt has 0 or 1 sequence number

**Receiver:**
- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is the expected packet seq #
- **Note:** receiver cannot know if its last ACK/NAK was received correctly at sender
Rdt2.2: NAK-free Protocol

- Same functionality as rdt2.1, using ACKs only
  - Most protocols are easier to generalize without NAKs
- Instead of NAKs, receiver sends an ACK for last packet received correctly
  - 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**
- \( \text{rdt}_\text{send}(\text{data}) \)
- \( \text{sndpkt} = \text{make}_\text{pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt}_\text{send}(\text{sndpkt}) \)
- Wait for call 0 from above

**Receiver FSM**
- \( \text{rdt}_\text{recv}(\text{rcvpkt}) \) AND \( \text{NOT corrupt}(\text{rcvpkt}) \) AND \( \text{isACK}(\text{rcvpkt}, 0) \)
- Wait for ACK 0
- \( \text{udt}_\text{send}(\text{sndpkt}) \)
- \( \text{sndpkt} = \text{make}_\text{pkt}(\text{ACK1}, \text{checksum}) \)
- \( \text{extract}(\text{rcvpkt}, \text{data}) \)
- \( \text{deliver}_\text{data}(\text{data}) \)
- \( \text{sndpkt} = \text{make}_\text{pkt}(\text{ACK1}, \text{checksum}) \)
- \( \text{udt}_\text{send}(\text{sndpkt}) \)
- \( \text{rdt}_\text{recv}(\text{rcvpkt}) \) AND \( \text{corrupt}(\text{rcvpkt}) \) OR \( \text{has_seq1}(\text{rcvpkt}) \)
- \( \text{udt}_\text{send}(\text{sndpkt}) \)
- \( \text{sndpkt} = \text{make}_\text{pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt}_\text{send}(\text{sndpkt}) \)
- \( \text{rdt}_\text{send}(\text{data}) \)
- Wait for 0 from below
- \( \text{rdt}_\text{recv}(\text{rcvpkt}) \) AND \( \text{corrupt}(\text{rcvpkt}) \) OR \( \text{isACK}(\text{rcvpkt}, 1) \)
- \( \text{udt}_\text{send}(\text{sndpkt}) \)
- \( \Lambda \)
Rdt3.0: Channels With Errors and Loss

- **New assumption:** underlying channel can also lose packets (data or ACKs)
  - Still no reordering
- Checksum, sequence numbers, ACKs, retransmissions will be of help, but not enough
- Why not?

- **Approach:** sender waits a “reasonable” amount of time for ACK
  - Retransmits if no ACK received in this time
  - Sender requires a timer
- If packet (or ACK) is delayed beyond the timer:
  - Retransmission will be duplicate, but the use of seq. #’s already handles this
  - Receiver must specify seq # of packet being ACKed
Rdt3.0 in Action (No Corruption)

sender

0

1

receiver

ACK0

ACK1

sender

0

1

receiver

ACK0

ACK1

no loss

forward loss

sender

0

1

timeout

1

1

X

receiver

ACK0

ACK1

ACK0
Rdt3.0 in Action (No Corruption)

sender             receiver
0                  ACK0
1                  ACK1
1                  ACK1
0                  ACK0

timeout

sender             receiver
0                  ACK0
1                  ACK1
1                  ACK1
0                  ACK0

reverse loss       premature timeout

retx 0?
Rdt3.0 Sender

rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above

Wait for ACK0

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

rdt_rcv(rcvpkt) AND
NOT corrupt(rcvpkt) AND isACK(rcvpkt,1)
stop_timer

udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
Λ
Wait for call 1 from above

Wait for ACK1

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

rdt_rcv(rcvpkt) AND
(corrupt(rcvpkt) OR isACK(rcvpkt,0))
Λ

rdt_rcv(rcvpkt) AND
NOT corrupt(rcvpkt) AND isACK(rcvpkt,0)
stop_timer

ftimeout

udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
Λ

Must not retransmit: ACK1 may be from a premature timeout on pkt1
Performance of Rdt3.0

• Rdt 3.0 works, but performance is low

• **Example**: 1 Gbps link, 15 ms end-to-end propagation delay, 1 KB packets, no loss or corruption:

\[ T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8 \text{ Kbits/pkt}}{10^9 \text{ bits/sec}} = 8 \text{ microsec} \]

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

• Server spends 0.008 ms being busy and 30 ms being idle, thus its link utilization is **only 0.027%**

• 1-KB pkt every 30 ms → 264 Kbps throughput

*Network protocol limits use of physical resources!*
Rdt3.0: Stop-and-wait Operation

<table>
<thead>
<tr>
<th>Event</th>
<th>Time (t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>first bit transmitted</td>
<td>t = 0</td>
</tr>
<tr>
<td>last bit transmitted</td>
<td>t = L / R</td>
</tr>
<tr>
<td>first packet bit arrives</td>
<td></td>
</tr>
<tr>
<td>last packet bit arrives, ACK departs</td>
<td></td>
</tr>
<tr>
<td>ACK arrives, send next packet</td>
<td>t = RTT + L / R</td>
</tr>
</tbody>
</table>

\[
U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027
\]
Performance of Rdt3.0

• Next assume that 10% of data packets are corrupted/lost (no loss in retransmissions or ACKs) and the timeout is 1 second
  - 90% of packets take \( (RTT + \frac{L}{R}) \approx 30 \text{ ms} \) to complete, while 10% require \( [\text{timeout} + RTT + 2\frac{L}{R}] \approx 1.03 \text{ sec} \)
  - Average per-packet delay \( 0.9 \times 0.03 + 0.1 \times 1.03 \text{ sec} = 130 \text{ ms} \)
  - Average rate 7.7 pkts/s or 61.5 Kbps

• Rdt3.0 similar to HTTP 1.0 or non-pipelined HTTP 1.1
• Next time we’ll improve this using pipelining, which allows multiple unack’ed packets at any time
• Quiz #2: chapter 2 problems and systems notes
  - P1, P3-P11, P13-P14, P20-P21