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

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

Diagram:

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

- **receive side**
  - reliable data transfer protocol (sending side)
  - reliable data transfer protocol (receiving 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
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
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

- **sender**
  - `rdt_send(data)`
    - `sndpkt = make_pkt(data, checksum)`
    - `udt_send(sndpkt)`
  - `rdt_rcv(rcvpkt) AND isACK(rcvpkt)`
    - \(\Lambda\)
  - `rdt_rcv(rcvpkt) AND isNAK(rcvpkt)`
    - `.udt_send(sndpkt)`
  - `wait for call from above`
  - `wait for ACK or NAK`

- **receiver**
  - `rdt_rcv(rcvpkt) AND corrupt(rcvpkt)`
    - `udt_send(NAK)`
  - `wait for call from below`
  - `rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`

\(\Lambda\) = empty action, i.e., do nothing
Rdt2.0: Operation With No Errors

wait for call from above

\[
\text{rdt\_send(data)}
\]

\[
\text{snkpkt = make\_pkt(data, checksum)}
\]

\[
\text{udt\_send(sndpkt)}
\]

wait for ACK or NAK

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

\[
\text{udt\_send(sndpkt)}
\]

wait for call from below

\[
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)}
\]

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

\[
\text{deliver\_data(data)}
\]

\[
\text{udt\_send(ACK)}
\]

\[
\text{Λ}
\]
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) &&
   notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

```
Lambda
```

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

- **sender**
  - \( \text{rdt\_send}(\text{data}) \)
  - \( \text{snkpkt} = \text{make\_pkt}(\text{data}, \text{checksum}) \)
  - \( \text{udt\_send}(\text{sndpkt}) \)

- **receiver**
  - \( \text{rdt\_rcv}(\text{rcvpkt}) \) AND \( \text{corrupt}(\text{rcvpkt}) \)
  - \( \text{udt\_send}(\text{NAK}, \text{chksum}) \)
  - \( \text{Wait for call from below} \)

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

- \( \text{extract}(\text{rcvpkt}, \text{data}) \)
  - \( \text{deliver\_data}(\text{data}) \)
  - \( \text{udt\_send}(\text{ACK}, \text{chksum}) \)

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

\[
\text{Wait for call 0 from above} \\
\text{sndpkt = make_pkt(0, data, checksum)} \\
\text{udt_send(sndpkt)} \\
\text{rdt_send(data)} \\
\text{udt_send(sndpkt)} \\
\text{rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt)} \\
\Lambda
\]

\[
\text{rdt_rcv(rcvpkt) AND [corrupt(rcvpkt) OR isNAK(rcvpkt)]} \\
\text{udt_send(sndpkt)} \\
\text{rdt_send(data)} \\
\text{sndpkt = make_pkt(1, data, checksum)} \\
\text{udt_send(sndpkt)} \\
\text{Wait for call 1 from above} \\
\text{Wait for ACK or NAK 1} \\
\text{Wait for ACK or NAK 0} \\
\Lambda
\]
Rdt2.1: Receiver, Handles Garbled ACK/NAKs

rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt)
AND has_seq0(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) AND corrupt(rcvpkt)
sndpkt = make_pkt(NAK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) AND
NOT corrupt(rcvpkt) AND
has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) AND
NOT corrupt(rcvpkt) AND
has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(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
  – 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 fragment

```
rdt_send(data)

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

```
Wait for call 0 from above
```

```
Wait for ACK 0
```

```
rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND has_seq1(rcvpkt)
```

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

receiver FSM fragment

```
rdt_send(data)

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

```
Wait for 0 from below
```

```
rdt_rcv(rcvpkt) AND [corrupt(rcvpkt) OR has_seq1(rcvpkt)]
```

```
rdt_send(sndpkt)
```

```
rdt_send(data)

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

```
Wait for ACK 0
```

```
rdt_rcv(rcvpkt) AND [corrupt(rcvpkt) OR isACK(rcvpkt, 1)]
```

```
udt_send(sndpkt)
```

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

```
Λ
```
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  receiver  sender  receiver
0        ACK0      0        ACK0
1        ACK1      1        x
0        ACK0      1        ACK1
1        ACK1      0        ACK0

no loss  forward loss
Rdt3.0 in Action (No Corruption)

sender  

0  →  ACK0  
1  →  ACK1

receiver

0  →  ACK0
1  →  ACK1

timeout

reverse loss

sender  

0  →  ACK0
1  →  ACK1

receiver

0  →  ACK0
1  →  ACK1

retx 0?

premature timeout
Rdt3.0 Sender

- \( \text{sndpkt} = \text{make_pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt_send}(	ext{sndpkt}) \)
- \( \text{start_timer} \)

- \( \text{Wait for call 0 from above} \)
- \( \text{rdt_recv(rcvpkt)} \)
- \( \Lambda \)
- \( \text{stop_timer} \)

- \( \text{timeout} \)
- \( \text{udt_send(sndpkt)} \)
- \( \text{start_timer} \)

- \( \text{rdt_recv(rcvpkt)} \)
- \( \Lambda \)

- \( \text{rdt_send(data)} \)
- \( \text{sndpkt} = \text{make_pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt_send(sndpkt)} \)
- \( \text{start_timer} \)

- \( \text{Wait for call 1 from above} \)
- \( \text{rdt_recv(rcvpkt)} \)
- \( \Lambda \)
- \( \text{stop_timer} \)

- \( \text{timeout} \)
- \( \text{udt_send(sndpkt)} \)
- \( \text{start_timer} \)

- \( \text{rdt_recv(rcvpkt)} \)
- \( \Lambda \)

- \( \text{rdt_recv(rcvpkt)} \)
- \( \text{rdt_recv(rcvpkt AND (corrupt(rcvpkt) OR isACK(rcvpkt, 1)))} \)

- \( \text{Must not retransmit: ACK1 may be from a premature timeout on pkt1} \)
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:

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

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

- 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 \(\rightarrow\) 264 Kbps throughput.
- **Network protocol limits use of physical resources!**
Rdt3.0: Stop-and-Wait Operation

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

\[
U_{\text{sender}} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027
\]
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 \((RTT + L/R) \approx 30\) ms to complete, while 10% require \([\text{timeout} + RTT + 2L/R] \approx 1.03\) sec
  – Average per-packet delay \(0.9 \times 0.03 + 0.1 \times 1.03\) sec = 130 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 system notes
  – P1, P3-P11, P13-P14, P20-P21