Reliable Data Transfer
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

**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

**deliver_data()**: called by rdt to deliver data to upper

**rdt_rcv()**: called when packet arrives on rcv-side of channel
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**

**event actions**
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

```
# Send function
rdt_send(data)
packet = make_pkt(data)
udt_send(packet)
```

```
# Receive function
rdt_rcv(packet)
extract (packet, data)
deliver_data(data)
```
Rdt2.0: channel with bit errors

- Underlying channel may flip bits in packet
  - Recall: UDP 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
  - Human scenarios using ACKs, NAKs?

New mechanisms in rdt2.0 (beyond rdt1.0):
  - Error detection
  - Receiver feedback: control msgs (ACK,NAK) rcvr->sender
rdt2.0: operation with no errors

- \( \text{rdt\_send(data)} \)
- \( \text{snkpkt = make\_pkt(data, checksum)} \)
- \( \text{udt\_send(sndpkt)} \)
- \( \text{rdt\_rcv(rcvpkt) \&\& isNAK(rcvpkt)} \)
- \( \text{udt\_send(sndpkt)} \)
- \( \text{wait for \( \Lambda \) or call from below} \)
- \( \text{rdt\_rcv(rcvpkt) \&\& isACK(rcvpkt)} \)
- \( \text{extract(rcvpkt, data)} \)
- \( \text{deliver\_data(data)} \)
- \( \text{udt\_send(ACK)} \)
- \( \text{udt\_send(NAK)} \)
- \( \text{wait for \( \text{ACK or NAK} \) or corrupt(rcvpkt)} \)
- \( \text{notcorrupt(rcvpkt)} \)
rdt2.0: error scenario

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

Wait for call from above

Wait for ACK or NAK

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

Wait for call from below

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

Λ

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

What to do?
- Sender ACKs/NAKs receiver’s ACK/NAK? What if sender ACK/NAK lost?
- Retransmit, but this might cause retransmission of correctly received pkt!

Handling duplicates:
- Sender retransmits current pkt if ACK/NAK garbled
- 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**

```
wait for call 0 from above

rdt send(data)
"
sndpkt = make_pkt(0, data, checksum)
udt send(sndpkt)

wait for ACK or NAK 0

rdt rcv(rcvpkt) &&
  ( corrupt(rcvpkt) ||
    isNAK(rcvpkt) )
udt send(sndpkt)

wait for call 0 from above

rdt rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
  && isACK(rcvpkt)
Λ

wait for call 1 from above

rdt send(data)
"

sndpkt = make_pkt(1, data, checksum)
udt send(sndpkt)

wait for ACK or NAK 1

rdt rcv(rcvpkt) &&
  ( corrupt(rcvpkt) ||
    isNAK(rcvpkt) )
udt send(sndpkt)

wait for call 1 from above

rdt rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
  && isACK(rcvpkt)
Λ
```
**rdt2.1: receiver, handles garbled ACK/NAKs**

- **Wait for 0 from below**
  - \( rdt\_rcv(rcvpkt) \) \&\& \( \text{not corrupt}(rcvpkt) \) \&\& \( \text{has\_seq0}(rcvpkt) \)
  - \( \text{extract}(rcvpkt, data) \)
  - \( \text{deliver\_data}(data) \)
  - \( \text{sndpkt} = \text{make\_pkt}(\text{ACK}, \text{chksum}) \)
  - \( \text{udt\_send}(\text{sndpkt}) \)

- **Wait for 1 from below**
  - \( rdt\_rcv(rcvpkt) \) \&\& \( \text{not corrupt}(rcvpkt) \) \&\& \( \text{has\_seq1}(rcvpkt) \)
  - \( \text{extract}(rcvpkt, data) \)
  - \( \text{deliver\_data}(data) \)
  - \( \text{sndpkt} = \text{make\_pkt}(\text{ACK}, \text{chksum}) \)
  - \( \text{udt\_send}(\text{sndpkt}) \)

- \( rdt\_rcv(rcvpkt) \) \&\& \( \text{corrupt}(rcvpkt) \)
  - \( \text{sndpkt} = \text{make\_pkt}(\text{NAK}, \text{chksum}) \)
  - \( \text{udt\_send}(\text{sndpkt}) \)

- \( rdt\_rcv(rcvpkt) \) \&\& \( \text{corrupt}(rcvpkt) \)
  - \( \text{sndpkt} = \text{make\_pkt}(\text{NAK}, \text{chksum}) \)
  - \( \text{udt\_send}(\text{sndpkt}) \)
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**
- `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)`
  - `wait for ACK 0`

**Receiver FSM Fragment**
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
  - `&& isACK(rcvpkt,0)`
- `wait for 0 from below`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK1, checksum)`
  - `udt_send(sndpkt)`

---

**Notes:**
- `corrupt(rcvpkt)`
- `isACK(rcvpkt,1)`
- `notcorrupt(rcvpkt)`
- `has_seq1(rcvpkt)`
rdt3.0: channels with errors and loss

**New assumption:** underlying channel can also lose packets (data or ACKs)
- Checksum, seq. #, ACKs, retransmissions will be of help, but not enough

**Q:** How to deal with loss?
- Sender waits until certain data or ACK lost, then retransmits

**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 (slightly different from book)**

- **rdt_send(data)**
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - start_timer
- **rdt_rcv(rcvpkt)**
  - Wait for call 0 from above
  - Wait for ACK0
  - Wait for call 1 from above
  - Wait for ACK1
- **timeout**
  - udt_send(sndpkt)
  - start_timer
- **rdt_rcv(rcvpkt)**
  - && notcorrupt(rcvpkt)
  - && isACK(rcvpkt, 1)
  - stop_timer
- **rdt_send(data)**
  - sndpkt = make_pkt(1, data, checksum)
  - udt_send(sndpkt)
  - start_timer
- **rdt_rcv(rcvpkt)**
  - && (corrupt(rcvpkt) || isACK(rcvpkt, 1))
  - udt_send(sndpkt)
  - start_timer
- **rdt_rcv(rcvpkt)**
  - && notcorrupt(rcvpkt)
  - && isACK(rcvpkt, 0)
  - stop_timer
- **rdt_rcv(rcvpkt)**
  - && (corrupt(rcvpkt) || isACK(rcvpkt, 0))
  - udt_send(sndpkt)
  - start_timer
- **timeout**
  - udt_send(sndpkt)
  - start_timer
rdt3.0 in action

(a) operation with no loss

(b) lost packet
rdt3.0 in action

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

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

\[ T_{\text{transmit}} = \frac{L}{R} \]  

\[ R \text{ (transmission rate, bps)} = \frac{8\text{kb/pkt}}{10^{9} \text{ b/sec}} = 8 \text{ microsec} \]

- \( U_{\text{sender}} \): utilization – fraction of time sender busy sending

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

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- Network protocol limits use of physical resources!
**rdt3.0 sender in perl**

- use IO::Select for timer
- do a `can_read()` with appropriate timeout
- when `can_read()` returns:
  - timeout expired? ==> retransmit, repeat `can_read()`
  - socket ready? ==> read packet
    - corrupt? ==> retransmit, repeat `can_read()`
    - wrong ACK# ==> retransmit, repeat `can_read()`
    - not corrupt and correct ACK# ==> wait for call from above