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)

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

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
- New mechanisms in rdt2.0 (beyond rdt1.0):
  - Error detection
  - Receiver feedback: control msgs (ACK, NAK) rcvr->sender
**rt2.0: operation with no errors**

- `rtt_send(data)`
  - `sndpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`
- Wait for call from below
- `rtt_rcv(rcvpkt)`
  - `& isACK(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
- `udt_send(ACK)`
- `rtt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `rtt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`
- `rtt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(NAK)`
- `rtt_rcv(rcvpkt) && corrupt(rcvpkt)`
- Wait for ACK or NAK
- Wait for call from below

**rtt2.0: error scenario**

- `rtt_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `rtt_send(data)`
- Wait for call from below
- `rtt_rcv(rcvpkt)`
  - `& isACK(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`
- `rtt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `rtt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`
- `rtt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(NAK)`
- `rtt_rcv(rcvpkt) && corrupt(rcvpkt)`
- Wait for ACK or NAK
- Wait for call from below

**rt2.0 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 retransmits current pkt if ACK/NAK garbled
  - Sender adds sequence number to each pkt
  - Receiver discards (doesn’t deliver up) duplicate 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

**rdt2.1: sender, handles garbled ACK/NAKs**

- `rtt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
- Wait for call from below
- `rtt_rcv(rcvpkt)`
  - `& isNAK(rcvpkt)`
  - `& (corrupt(rcvpkt) || isNAK(rcvpkt))`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(sndpkt)`
  - `rtt_send(data)`
- Wait for call from below

**rdt2.1: receiver, handles garbled ACK/NAKs**

- `rtt_rcv(rcvpkt)`
  - `& notcorrupt(rcvpkt)`
  - `& has_seq0(rcvpkt)`
  - `sndpkt = make_pkt(NAK, checksum)`
  - `udt_send(sndpkt)`
- Wait for call from below
- `rtt_rcv(rcvpkt)`
  - `& notcorrupt(rcvpkt)`
  - `& has_seq1(rcvpkt)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`
- Wait for call from below

**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**

**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. # already handles this
  - Receiver must specify seq # of pkt being ACKed
- Requires countdown timer

**rdt3.0 sender (slightly different from book)**

**rdt3.0 in action**

- (a) operation with no loss
- (b) lost packet
- (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:
  \[ T_{\text{transmit}} = \frac{L}{R} \text{ (packet length in bits)} \]
  \[ T_{\text{transmit}} = \frac{8\text{kb/pkt}}{10^{9} \text{ b/sec}} = 8 \text{ microsec} \]

- Utilization – fraction of time sender busy sending
  \[ U_{\text{sender}} = \frac{L}{RTT + L / R} \]
  \[ U_{\text{sender}} = \frac{0.008}{30.008} = 0.00027 \]

- 1KB pkt every 30 msec -> 33kB/sec throughput 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