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

**send side**

- **rdt_send():** called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

**receive side**

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

**udt_send():** called by rdt, to transfer packet over unreliable channel to receiver
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 reads data from underlying channel

sender

receiver
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

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

Lambda

- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
- udt_send(snkpkt)
- rdt_send(sndpkt)
- rdt_send(sndpkt)
- rdt_rcv(rcvpkt) && isACK(rcvpkt)
- rdt_send(sndpkt)
- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
- udt_send(NAK)
rdt2.0: error scenario

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

Wait for call from above

Wait for ACK or NAK

\[
\text{rdt\_recv}(\text{rcvpkt}) \land \text{isNAK}(\text{rcvpkt})
\]
\[
\text{udt\_send}(\text{sndpkt})
\]

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

\[
\Lambda
\]

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

Wait for call from below

Wait for ACK or NAK

\[
\text{rdt\_recv}(\text{rcvpkt}) \land \text{corrupt}(\text{rcvpkt})
\]
\[
\text{udt\_send}(\text{NAK})
\]
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**

- **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)`

- **rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)**
  - `Λ`

- **Wait for call 0 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) )`

- **rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)**
  - `Λ`

- **Wait for call 1 from above**
rdt2.1: receiver, handles garbled ACK/NAKs

\[
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \\
\neg \text{has_seq}_1(\text{rcvpkt}) \\
\text{sndpkt} = \text{make_pkt}(\text{ACK}, \text{chksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\]

\[
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \\
\neg \text{has_seq}_0(\text{rcvpkt}) \\
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver}_\text{data}(\text{data}) \\
\text{sndpkt} = \text{make_pkt}(\text{NAK}, \text{chksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\]

\[
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \\
\neg \text{has_seq}_1(\text{rcvpkt}) \\
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \\
\neg \text{has_seq}_0(\text{rcvpkt}) \\
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver}_\text{data}(\text{data}) \\
\text{sndpkt} = \text{make_pkt}(\text{ACK}, \text{chksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\]
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 \textit{explicitly} include seq \# of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: \textit{retransmit current pkt}
**rdt2.2: sender, receiver fragments**

**Sender FSM fragment**

1. `rdt_send(data)`
2. `sndpkt = make_pkt(0, data, checksum)`
3. `udt_send(sndpkt)`

**Wait for call 0 from above**

4. `rdt_recv(rcvpkt) &&`
5. `isACK(rcvpkt,1)`
6. `udt_send(sndpkt)`

**Wait for ACK 0**

7. `rdt_recv(rcvpkt) &&`
8. `notcorrupt(rcvpkt)`
9. `isACK(rcvpkt,0)`

**Receiver FSM fragment**

10. `rdt_recv(rcvpkt) &&`
11. `notcorrupt(rcvpkt)`
12. `isACK(rcvpkt,0)`

13. `.udt_send(sndpkt)`

14. `rdt_recv(rcvpkt) &&`
15. `corrupt(rcvpkt) ||`
16. `has_seq1(rcvpkt))`

17. `udt_send(sndpkt)`

18. `Wait for 0 from below`

19. `extract(rcvpkt, data)`
20. `deliver_data(data)`
21. `sndpkt = make_pkt(ACK1, checksum)`

22. `udt_send(sndpkt)`
**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)
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:

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

- Utilization: \( U_{\text{sender}} = \frac{L / R}{RTT + L / R} \)
\[ = \frac{.008}{30.008} = 0.00027 \]

- 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