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

![Diagram showing the data flow](image)
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

**state 1**

**state 2**

**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

sender

receiver
Rdt2.0: channel with bit errors

• underlying channel may flip bits in packet
  – 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 $\{\text{ACK}, \text{NAK}\}$
**rdt2.0: FSM specification**

**sender**

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

**receiver**

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `^`  
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`
rdt2.0: operation with no errors

rdt_send(data)

snkpkt = make_pkt(data, checksum)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

Wait for
call from
above

Wait for
ACK or
NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Wait for
call from
below

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
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) && isACK(rcvpkt)
  - udt_send(ACK)

- 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
What happens if ACK/NAK corrupted?

• sender doesn’t know what happened at receiver!
• can’t just retransmit: possible duplicate

Handling duplicates:
sender retransmits current pkt if ACK/NAK garbled
sender adds sequence number to each pkt
receiver discards (doesn’t deliver up)

stop and wait pkt:
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)

rdt_rcv(rcvpkt) &&
  ( notcorrupt(rcvpkt) && isACK(rcvpkt) )

Lambda

Lambda

Lambda

Lambda

r
```
rdt2.1: receiver, handles garbled ACK/NAKs

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  && has_seq0(rcvpkt)
  
  extract(rcvpkt, data)
  deliver_data(data)
  sndpkt = make_pkt(ACK, chksum)
  udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  (corrupt(rcvpkt)
    sndpkt = make_pkt(NAK, chksum)
    udt_send(sndpkt)
  
  rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq1(rcvpkt)
  
  sndpkt = make_pkt(ACK, chksum)
  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
  - 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**

- \( \text{rdt}_{\text{send}}(\text{data}) \)
  \[ \text{sndpkt} = \text{make_pkt}(0, \text{data}, \text{checksum}) \]
  \( \text{udt}_{\text{send}}(\text{sndpkt}) \)

- **Wait for call 0 from above**

- **Wait for ACK 0**

**receiver FSM fragment**

- \( \text{rdt}_{\text{rcv}}(\text{rcvpkt}) \) &&
  \( (\text{corrupt}(\text{rcvpkt}) || \text{has_seq1}(\text{rcvpkt})) \)
  \( \text{udt}_{\text{send}}(\text{sndpkt}) \)

- Wait for 0 from below

- \( \text{rdt}_{\text{rcv}}(\text{rcvpkt}) \) && \( \text{notcorrupt}(\text{rcvpkt}) \) && \( \text{isACK}(\text{rcvpkt}, 0) \)

- Extract (\text{rcvpkt}, \text{data})

- Deliver data (\text{data})

- \( \text{sndpkt} = \text{make_pkt}(\text{ACK1}, \text{checksum}) \)
  \( \text{udt}_{\text{send}}(\text{sndpkt}) \)