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)

Principles of reliable data transfer

(a) provided service
(b) service implementation
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

<table>
<thead>
<tr>
<th>state 1</th>
<th>state 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

event causing state transition
actions taken on state transition

state: when in this “state” next state uniquely determined by next event

event actions

Stan Kurkavsky
**Rdt 1.0: 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

```
Wait for call from above
rdt_send(data)
packet = make_pkt(data)
udt_send(packet)
```

sender

```
Wait for call from below
rdt_rcv(packet)
extract (packet, data)
deliver_data(data)
```

receiver

Stan Kurkavsky

**Rdt 2.0: 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 (ACK, NAK) rcvr->sender

Stan Kurkavsky
Rdt 2.0: FSM specification

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

Wait for call from above
```

```
rdt_rrev(rcvpkt) && isNAK(rcvpkt)
  udt_send(sndpkt)

Wait for ACK or NAK
```

```
udt_send(ACK)
```

sender

```
rdt_rrev(rcvpkt) && isACK(rcvpkt)
  \Lambda
```

```
Wait for call from below
```

```
rdt_rrev(rcvpkt) && notcorrupt(rcvpkt)
  extract(rcvpkt.data)
  deliver_data(data)
  udt_send(ACK)
```

```
```
```
receiver
```

```
```

Rdt 2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

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

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

```
```
```
```
Stan Kurkavsky
```
Rdt 2.1: sender, handles garbled ACK/NAKs

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

rdt_rcv(rcvpkt)
   && not corrupt(rcvpkt)
   && isACK(rcvpkt)
   \(\land\)

rdt_rcv(rcvpkt)
   && ( corrupt(rcvpkt) || isNAK(rcvpkt) )
   sndpkt = make_pkt(1, data, checksum)
   udt_send(sndpkt)
```

Rdt 2.1: receiver, handles garbled ACK/NAKs

```
rdt_rcv(rcvpkt)
   && not corrupt(rcvpkt)
   && has_seq0(rcvpkt)
   extract(rcvpkt, data)
   deliver_data(data)
   sndpkt = make_pkt(ACK, checksum)
   udt_send(sndpkt)

rdt_rcv(rcvpkt)
   && ( corrupt(rcvpkt) || not corrupt(rcvpkt) && has_seq1(rcvpkt) )
   sndpkt = make_pkt(NAK, checksum)
   udt_send(sndpkt)
```

Stan Kurkavsky
**Rdt 2.1: 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

**Rdt 2.2: 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*
Rdt 2.2: sender, receiver fragments

```
rdt_send(data)

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

wait for call 0 from above

Wait for ACK 0

sender FSM fragment

Rdt 3.0: 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

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

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
```
Rdt 3.0 sender

Rdt 3.0 in action

Stan Kurkovsky
**Rdt 3.0 in action**

![Diagram of Rdt 3.0 in action](image)

**Rdt 3.0: stop-and-wait operation**

- \( L \) = packet length in bits
- \( R \) = transmission rate, bps
- \( U \) = utilization – fraction of time sender busy sending

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

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: go-Back-N, selective repeat

Pipelining: increased utilization

$U_{sender} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008$

Increase utilization by a factor of 3!
**Go-Back-N**

**Sender:**
- k-bit seq # in pkt header
- "window" of up to \(N\), consecutive unack'ed pkts allowed

- ACK(\(n\)): ACKs all pkts up to, including seq # \(n\) - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- \(timeout(n)\): retransmit pkt \(n\) and all higher seq # pkts in window

\[send_{base} \quad nextseqnum\]
\[\text{already ack'ed} \quad \text{sent, not yet ack'ed} \quad \text{usable, not yet sent} \quad \text{not usable}\]
\[\text{window size} \quad N\]

---

**GBN: sender extended FSM**

```
rdt_send(data)
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)
    udt_send(sndpkt[nextseqnum])
if (base == nextseqnum)
    start_timer
    nextseqnum++
}
else
    refuse_data(data)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
if (base == nextseqnum)
    stop_timer
else
    start_timer
```
**GBN: receiver extended FSM**

- **ACK-only**: always send ACK for correctly-received pkt with highest \textit{in-order} seq #
  - may generate duplicate ACKs
  - need only remember \textit{expectedseqnum}
- **out-of-order pkt**:
  - discard (don't buffer) \textit{-> no receiver buffering!}
  - Re-ACK pkt with highest \textit{in-order} seq #

```c
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++
```

---

**GBN in action**

Sender:
- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)
- rcv ACK0
- send pkt4
- rcv ACK1
- send pkt5
- pkt2 timeout
- send pkt2
- send pkt3
- send pkt4
- send pkt5

Receiver:
- rcv pkt0
- send ACK0
- rcv pkt1
- send ACK1
- rcv pkt3, discard send ACK1
- rcv pkt4, discard send ACK1
- rcv pkt5, discard send ACK1
- rcv pkt2, deliver send ACK2
- rcv pkt3, deliver send ACK3
Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows

Stan Kurkovsky
Selective repeat

**Sender**

data from above:

- if next available seq # in window, send pkt

**timeout(n):**

- resend pkt n, restart timer

**ACK(n) in [sendbase,sendbase+N]:**

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

**Receiver**

pkt n in [rcvbase,rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

**otherwise:**

- ignore

Selective repeat in action

Stan Kurkovsky
Selective repeat: dilemma

Example:
- seq #’s: 0, 1, 2, 3
- window size=3
  - receiver sees no difference in two scenarios!
  - incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

(a) receive packet with seq number 0

(b) receive packet with seq number 0