Reliable Data Transfer

Instructor: Anirban Mahanti
Office: ICT 745
Email: mahanti@cpsc.ucalgary.ca
Class Location: ICT 121
Lectures: MWF 12:00 - 12:50


Slides are adapted from the companion web site of the book, as modified by Anirban Mahanti (and Carey Williamson).

Principles of Reliable Data Transfer

- important in application, transport, and 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: FSMs

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 is uniquely determined by next event

<table>
<thead>
<tr>
<th>state</th>
<th>event causing state transition</th>
<th>actions taken on state transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rdt1.0: Data Transfer over a Perfect 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

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

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

Rdt2.0: channel with bit errors [stop & wait protocol]

- Assumptions
  - All packets are received
  - Packets may be corrupted (i.e., bits may be flipped)
  - Checksum to detect bit errors
- How to recover from errors? Use ARQ mechanism
  - acknowledgements (ACKs): receiver explicitly tells sender that packet received correctly
  - negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
  - sender retransmits pkt on receipt of NAK
- What about error correcting codes?

```
rtt2.0: FSM specification
```

```
sender
```

```
Wait for call from above
rtt_send(data)
smkpkt = make_pkt(data, checksum)
udt_send(smkpkt)
```

```
rtd_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(NAK)
rtd_rcv(rcvpkt) && isACK(rcvpkt)
rtd_send(ACK)
```

```
receiver
```

```
rtd_rcv(rcvpkt) && notcorrupt(rcvpkt)
evaluate(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```
rdt2.0: Observations

1. A stop-and-Wait protocol
2. What happens when ACK or NAK has bit errors?
Approach 1: resend the current data packet?

Handling Duplicate Packets

- sender adds sequence number to each packet
- sender retransmits current packet if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate packet

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

- \( \text{rdt}_2.1 \): receives and handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
- \( \text{rdt}_2.1 \): receiver, handles garbled ACK/NAKs.
**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: The case of “Lossy” Channels**

- **Assumption**: Underlying channel can also lose packets (data or ACKs).
- **Approach**: Sender waits “reasonable” amount of time for ACK (a Time-Out).
  - Time-out value?
  - Possibility of duplicate packets/ACKs?
- 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.
**rdt3.0 sender**

- Start timer
- rdt_send(data)
- Wait for ACK
- rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))
- Wait for call 1 from above
- rdt_send(data)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
- Stop timer
- rdt_send(sndpkt)
- Start timer
- Timeout

**rdt3.0 in action**

- (a) operation with no loss
- (b) lost packet
- (c) lost ACK
- (d) premature timeout
**Pipelining: Motivation**

- Stop-and-wait allows the sender to only have a single unACKed packet at any time.
- Example: 1 Mbps link (R), end-2-end round trip propagation delay (D) of 92 ms, 1KB packet (L):
  
  \[
  T_{\text{transmit}} = \frac{L}{R} \text{ (packet length in bits)} = \frac{8 \text{kb/pkt}}{10^6 \text{ bps}} = 8 \text{ ms}
  \]
  
  \[
  U_{\text{sender}} = \frac{L}{R} = \frac{8 \text{ ms}}{100 \text{ ms}} = 0.08
  \]

- 1KB pkt every 100 ms → 80Kbps throughput on a 1 Mbps link
- What does bandwidth x delay product tell us?

**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
- Two generic forms of pipelined protocols
  - **go-Back-N**
  - **selective repeat**
**Pipelining: increased utilization**

First packet bit transmitted, $t = 0$

Last bit transmitted, $t = L / R$

ACK arrives, send next packet, $t = D + L / R$

Increase utilization by a factor of 3!

**Go-Back-N**
- Allow up to $N$ unACKed pkts in the network
  - $N$ is the Window size
- Sender Operation:
  - If window not full, transmit
  - ACKs are cumulative
  - On timeout, send all packets previously sent but not yet ACKed.
  - Uses a single timer - represents the oldest transmitted, but not yet ACKed pkt

**GBN: sender extended FSM**

```plaintext
rtn_send(data)
if (nextseqnum < base+N) {
  sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)
  udt_send(sndpkt[nextseqnum])
  if (base == nextseqnum)
    start_timer
    nextseqnum++
endif
else
  refuse_data(data)
  base = getacknum(rcvpkt)+1
  if (base == nextseqnum)
    stop_timer
    start_timer
  endif
endif
```

Lambda
**GBN: receiver extended FSM**

![Diagram of GBN receiver extended FSM](image)

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

---

**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

Selecting 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 Example
Another Example

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?