Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
Principles of Reliable data transfer

- important in app., transport, link layers
- One of the most 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

`deliver_data()` : called by `rdt` to deliver data to upper

`udt_send()` : called by `rdt`, to transfer packet over unreliable channel to receiver

`rdt_rcv()` : called when packet arrives on rcv-side of channel
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

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
  - 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

rdt2.0: FSM specification
rdt2.0: operation with no errors

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
- rdt_send(sndpkt)
- extract(rcvpkt, data)
- deliver_data(data)
- udt_send(ACK)
- rdt_send(sndpkt)
- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
- udt_send(NAK)
- rdt_send(sndpkt)
- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
- Wait for ACK or NAK
- Wait for call from below
- Wait for call from above

rdt2.0: error scenario

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
- udt_send(sndpkt)
- extract(rcvpkt, data)
- deliver_data(data)
- udt_send(ACK)
- rdt_send(sndpkt)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
- udt_send(NAK)
- rdt_send(sndpkt)
- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
- Wait for ACK or NAK
- Wait for call from below
- Wait for call from above
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

Handling duplicates:
- sender adds sequence number to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- 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)

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

\wedge

:\text{Wait for call 0 from above}

:: Wait for ACK or NAK 0

:: rdt_send(data)

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

:: udt_send(sndpkt)

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

:: \text{Wait for call 1 from above}

:: rdt_send(data)

:: sndpkt = make_pkt(1, data, checksum)

:: udt_send(sndpkt)

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

:: \text{Wait for ACK or NAK 1}

:: rdt_rcv(rcvpkt)
&& (corrupt(rcvpkt) || isNAK(rcvpkt))

:: udt_send(sndpkt)
```
rdt2.1: receiver, handles garbled ACK/NAKs

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](image)

![receiver FSM fragment](image)
**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

**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**

![Flowchart diagram for rdt3.0 sender]
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:

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

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

□ \( U_{\text{sender}} \): utilization – fraction of time sender busy sending

□ 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link

□ network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

<table>
<thead>
<tr>
<th>Event</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First packet bit sent</td>
<td>0</td>
</tr>
<tr>
<td>Last packet bit sent</td>
<td>( L/R )</td>
</tr>
<tr>
<td>First packet bit received</td>
<td>( RTT )</td>
</tr>
<tr>
<td>ACK arrives, send next packet</td>
<td>( RTT + L/R )</td>
</tr>
</tbody>
</table>

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

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

Pipelining: increased utilization

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

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 deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM
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) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

\begin{align*}
\text{Window Size (W) } & \leq (2^n) - 1, \text{ where } n \text{ is the number of bits used for sequence numbers}
\end{align*}
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**

- 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

- pkt0 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt1 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt2 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt3 sent, window full
  - 0 1 2 3 4 5 6 7 8 9

- pkt0 rcvd, delivered, ACK0 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt1 rcvd, delivered, ACK1 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt2 TIMEOUT, pkt2 resent
  - 0 1 2 3 4 5 6 7 8 9

- pkt3 rcvd, packet 4 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt1 rcvd, packet 3 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt0 rcvd, packet 1 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt0 rcvd, delivered, ACK0 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt4 rcvd, buffer
  - 0 1 2 3 4 5 6 7 8 9

- pkt5 rcvd, buffer
  - 0 1 2 3 4 5 6 7 8 9

- pkt5 rcvd, ACK5 sent
  - 0 1 2 3 4 5 6 7 8 9

- pkt2 rcvd, pkt2, pkt3, pkt4, pkt5 delivered, ACK2 sent
  - 0 1 2 3 4 5 6 7 8 9

- ACK3 rcvd, nothing sent
  - 0 1 2 3 4 5 6 7 8 9
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?

Window Size (W) \(\leq 2^{n-1}\), where \(n\) is the number of bits used for sequence numbers.