Principles of Reliable Transport

To do ...

- A reliable transport, piece by piece
- ACKs, timers, retransmissions and sequence numbers
Reliable Data Transfer

- What do you get from a reliable data transfer
  - Nothing corrupted, nothing lost and all in order
- A framework for discussion
A finite state machine (FSM) definition

Sending side

- **Initial state**: Wait for call from above
- **Event causing the transition**: rdt_send(data)
  - packet=make pkt(data)
  - udt_send(packet)
- **Action taken**: Sends the data from the upper layer via rdt_send, makes a packet and sends it

Receiving side

- **Wait for call from bellow**: rdt_rcv(packet)
  - extract(packet, data)
  - deliver_data(data)

- **No bit corruption**
- **No loss**
- **In order**
Assume there is no loss for now …

First, how do people handle a call over a noisy connection?
- “OK / yeah / …” – Positive ACK
- “Sorry? / Please repeat that / …” – Negative ACK
- Automatic Repeat reQuest (ARQ) protocols

Three things you need in an ARQ
- Error detection – requires some extra bits, besides the data itself
- Receiver feedback – no other way for the sender to know; ACKs and NACKs are examples of this feedback
- Retransmission
RDT Over a Channel with Bit Errors – rdt2.0

Sending side

```
rdt_send(data)
```

```
packet=make_pkt(data)
udt_send(packet)
```

```
rdt_rcv(rcvpacket) && isACK(rcvpkt)

udt_send(sndpkt)
```

```
rdt_rcv(rcvpacket) && isNACK(rcvpkt)

Lambda
```

Receiving side

```
rdt_rcv(rcvpacket) && corrupt(rcvpkt)

sndpkt=make_pkt(NACK)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpacket) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK)
udt_send(sndpkt)
```

Bit corruption
No loss
In order

When in ‘Wait for ACK or NACK’, sender can’t get any more data from the upper layers - **Stop-and-wait**
rdt2.0 has a fatal flaw!

- What happens if ACKs/NAKs are corrupted?
  - Sender doesn’t know what happened at the receiver!
  - How about retransmit if received a garbled ACK/NACK?  Duplicates?

- Handling duplicates
  - Sender retransmits current packet if ACK/NAK corrupted
  - Sender adds sequence number to each packet
  - Receiver discards (doesn’t deliver up) duplicate packet

- With a “stop-and-wait” protocol, one bit seq # is enough
  - To tell between a resend and a new packet
  - NACKs/ACKs don’t need seq # since there’s no loss (so, it must be about the most recently sent packet
Rdt2.1 sender

```
rdt_send(data)

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

Wait for call 0 from above

rdt_rcv(rcvpacket) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ

rdt_send(data)

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

Wait for call 0 from above

rdt_rcv(rcvpacket) && (corrupt(rcvpkt) || isNACK(rcvpkt))
udt_send(sndpkt)

Wait for ACK or NACK 0

rdt_rcv(rcvpacket) && notcorrupt(rcv pkt) && isACK(rcvpkt)
Λ

Wait for ACK or NACK 1

rdt_send(data)

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

Wait for call 1 from above
```

```
```

```
```
Rdt2.1 receiver

```c
rdt_rcv(rcvpacket) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```c
rdt_rcv(rcvpacket) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, checksum)
udt_send(sndpkt)
```

```c
rdt_rcv(rcvpacket) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```c
rdt_rcv(rcvpacket) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```c
rdt_rcv(rcvpacket) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, checksum)
udt_send(sndpkt)
```

```c
rdt_rcv(rcvpacket) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```
NACKs and ACKs

- rdt2.1 uses positive and negative acknowledgments
  - When a corrupted packet arrives, send a NACK
  - When an out of order packet (previous one) arrives, send an ACK

- Replace NACKs with an ACK for last correctly received packet
  - “This is the last thing I understood of what you said” → receiver did not correctly receive the packet following the one being ACKed twice
  - Of course now the ACK needs to include the seq # of the packet ACKed

rdt2.2
### rdt2.2: sender, receiver fragments

**Sender FSM fragment**

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

### Receiver FSM fragment

- `wait for call 0 from above`
- `wait for ACK 0`
- `wait for 0 from below`
- `udt_send(sndpkt)`

**Note we are not creating a new `sndpkt`**

- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))`
- `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)`
  - `udt_send(sndpkt)`

**Sender FSM fragment continued**

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK1, checksum)`
  - `udt_send(sndpkt)`

- `wait for ACK 0`

**Diagram notes**

- The diagram illustrates the state transitions and actions for both the sender and receiver FSMs, detailing how they handle data transmission, acknowledgment, and sequencing.
Two additional concerns to address
- Detecting packet loss
- What to do in the event of packet loss

Who needs to detect loss? We make the sender responsible

How? Time-out, either the message got lost or the ACK did
  - Retransmit anyway

But how long? > RTT + processing time ... hard to estimate
  - And you want to recover ASAP → Duplicate packets

Because pckt seq # alternate between 0/1 – alternating-bit protocol

rdt3.0
rdt3.0 in Operation

**Operation with no loss**

- **Sender**
  - send pkt0
  - rcv ACK0
  - send pkt1
  - rcv ACK1
  - send pkt0
  - rcv pkt0
  - send ACK0

- **Receiver**
  - pkt0
  - ACK0
  - pkt1
  - ACK1
  - pkt0
  - ACK0

**Lost packet**

- **Sender**
  - send pkt0
  - rcv pkt0
  - send ACK0
  - timeout
  - resend pkt1

- **Receiver**
  - pkt0
  - ACK0
  - pkt1
  - ✘
  - send ACK1
  - pkt1
  - timeout

- **Sender**
  - rcv ACK0
  - send pkt1

- **Receiver**
  - pkt1
  - ACK1
  - pkt0
  - ACK0
  - rcv pkt0
  - send ACK0
rdt3.0 in Operation

Sender
- send pkt0
- rcv ACK0
- send pkt1
- timeout
- send pkt1
- rcv ACK1
- send pkt0
- rcv ACK0

Receiver
- rcv pkt0
- ACK0
- rcv pkt1
- pkt1
- ACK1
- send ACK0
- rcv pkt0
- ACK0
- rcv pkt1
- pkt1
- ACK1
- send ACK1
- rcv pkt0
- ACK0

Premature timeout

Sender
- send pkt0
- rcv ACK0
- send pkt1
- timeout
- send pkt1
- rcv ACK1
- send pkt0
- rcv ACK0

Receiver
- rcv pkt0
- ACK0
- rcv pkt1
- pkt1
- ACK1
- send ACK0
- rcv pkt0
- ACK0
- rcv pkt1
- pkt1
- ACK1
- send ACK1
- rcv pkt0
- ACK0

Lost ACK
Stop-and-wait

- rdt3.0 – functionally correct but stop-and-wait … poor utilization
  - Utilization – fraction of time the sender is busy sending bits

![Diagram of stop-and-wait protocol]

- first packet bit transmitted, $t = 0$
- last packet bit transmitted, $t = L / R$
- first packet bit arrives
- last packet bit arrives, send ACK ($t = RTT/2 + L/R$)
- Assuming a tiny ACK pkt so we can ignore transmission time
- ACK arrives, send next packet, $t = RTT + L / R$
Stop-and-wait

- Consider two host on two sides of the US (~30 msec RTT)
  - Transmission rate $R$ (e.g., 1 Gbps) and packet size $L$ (e.g., 1KB)

\[ d_{\text{trans}} = \frac{L}{R} = \frac{8,000 \text{ bits/packet}}{10^9 \text{ bits/sec}} = 8 \text{ microsec} \]

\[ t = \frac{L}{R} = 30.008 \text{ msec} \]

Utilization:
\[ Utilization = \frac{\frac{L}{R}}{\text{RTT} + \frac{L}{R}} = 0.00027 \]
Pipelining hides latency to increase throughput

- Pipelining – Allows multiple “in-flight” pkts, not yet ACKed

- Window size is the max number of in-flight packets
- Finite to limit data buffering needed at each end, and load placed on the network
Pipelining increases link utilization

\[ \text{Utilization} = \frac{3L/R}{RTT + L/R} = 0.00081 \]

3-packet pipelining increases utilization by a factor of 3!
How to do pipelining

- Packet buffering and acknowledgement become more complex
  - Range of sequence numbers to accommodate for this
    - How to do it depends on how we deal with lost, corrupted or long delayed packets
  - Buffering to hold on packets sent but not yet ACKed

- Need flow/congestion control to prevent overwhelming the receiver/network

- Two basic approaches to pipelined error recovery
  - Go-Back-N
  - Selective repeat
Pipelining option #1: Go Back N

- Window size is $N$, sender can have up to $N$ pkts in flight
  - The range of possible pkts transmitted but not yet ack’ed
  - As it runs, the window “slides” forward – sliding-window protocol
- Receiver sends cumulative ACK: “Got everything up to seq. # x”
  - Discard out-of-order pkt, re-send ACK of last in-order seq. #
  - If sender does not get an ACK after some timeout interval, resend all pkts starting from pkt after the last ACK’ed pkt
- If the sender timeout expires several times w/o receiving any ACK, give up on the connection
GBN: sender extended FSM

rdr_send(data)

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

"Refuse" could be done differently, such as using a synchronization variable that would allow the upper-layer call when the window is not full

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 in-order seq #
  - may generate duplicate ACKs
  - need only remember expected seq#
  - out-of-order pkt:
    - discard (don’t buffer): no receiver buffering! *(Why?!)*
    - re-ACK pkt with highest in-order seq #

BTW, how would you implement this?
Window size is 4

Because of window size, sends 0 to 3 and wait

With every ACK (one less packet in the pipe), the sliding window can move forward and sender can send another packet

Out of order packets are discarded
Go Back N pros / cons

- **Pros:** Easy to implement:
  - Sender just stores # of last ACK and maintains a timer
  - Receiver just stores expected seq number and immediately passes new in-order packets to listening app

- **Cons:** A single lost or delayed pkt invalidates all in-flight data
  - When the window size and the bandwidth*delay product are large, there can be a lot of packets in the pipeline
  - Receiver can throw out a lot of good data, just because it’s “early”
    - I.e. lacks receiver buffering
  - Lose an entire window of data due to one “bad” packet
Pipelining option #2: Selective Repeat

- Only re-send an pkt whose transmission or ACK was lost
- Receiver individually ACKs all received pkts
- Out-of-order pkts are stored by receiver and later reassembled
- Sender keeps one timer per each in-flight packet, and will re-send any pkts not ACK’ed before timeout
- Window of size $N$ limits the max range of un-ACK’ed pkts
  - Receiver drops received pkts with seq # outside the window
  - This prevents pkts from old connection from getting inserted into new connection’s data stream
Selective repeat

**sender**

- **data from above:**
  - if next available seq # in window, send pkt
  - else, return up or buffered

- **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, i.e., consecutively numbered), advance window to next not-yet-received pkt

- **pkt \( n \) in \([rcvbase-N, rcvbase-1]\):**
  - ACK\( (n) \)

- **otherwise:**
  - ignore

---

Note it re-ack, rather than ignore already received pkts
Selective Repeat windows

Sender window advances when lowest packet is ACK’ed.

Receiver window advances when lowest packet is received.

Sender view of sequence numbers

Receiver view of sequence numbers

already ack’ed
sent, not yet ack’ed

usable, not yet sent

not usable

out of order (buffered) but already ack’ed

acceptable (within window)

Expected, not yet received

not usable
Selective repeat in action

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)
rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived

pkt 2 timeout
send pkt2
record ack4 arrived
record ack5 arrived

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3
receive pkt4, buffer,
send ack4
receive pkt5, buffer,
send ack5
rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2
Assumes that 2-bit seq number and a window of size 3

Solution: window length must be < half the max seq number

Timeout, re-send pkt0

Receive pkt with seq number 0

New pkt or a retransmit?
Recap

- The underlying principles of reliable transfer → ready for TCP
- A summary of mechanisms and their use

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum</td>
<td>Detect bit errors in a transmitted packet</td>
</tr>
<tr>
<td>Timer</td>
<td>To timeout/retransmit a packet; since it’s a guess, need to handle duplicates</td>
</tr>
<tr>
<td>Seq number</td>
<td>To detect gaps in sequences of packets sent and detect duplicates as well</td>
</tr>
<tr>
<td>ACK</td>
<td>For the receiver to tell the sender it got it, may be individual or cumulative</td>
</tr>
<tr>
<td>NACK</td>
<td>For the receiver to tell the sender that a pkt wasn’t received correctly</td>
</tr>
<tr>
<td>Window, pipelining</td>
<td>To improve utilization over stop-and-wait mode; the window may be set based on the receiver’s ability to receive and buffer msgs, the level of congestion in the network or both</td>
</tr>
</tbody>
</table>