**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
- **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

**state**: when in this "state" next state uniquely determined by next event

---

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
**Rdt2.0: channel with bit errors**

- underlying channel may flip bits in packet
  - recall: UDP 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

**rdt2.0: FSM specification**

**sender**

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- rdt_send(snkpkt)

**receiver**

- rdt_rcv(rcvpkt)
- udt_send(rcvpkt)

- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
- udt_send(snkpkt)

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

- 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) && 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)

### rdt2.0: error scenario

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
  - udt_send(sndpkt)
- rdt_send(data)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  - extract(rcvpkt.data)
  - deliver_data(data)
  - udt_send(ACK)
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

What to do?
- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

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
### 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, checksum)`
- `udt_send(sndpkt)`

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

- `wait for 0 from below`
- `wait for 1 from below`

### 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**

```plaintext
rdt_send(data)

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

Wait for call 0 from above

rdt_rcv(rcvpkt) &&
  ( corrupt(rcvpkt) ||
    has_seq1(rcvpkt) ) // sender FSM fragment
udt_send(sndpkt)

Wait for ACK 0

rdt_rcv(rcvpkt) &&
  isACK(rcvpkt,1)
  udt_send(sndpkt)

receiver FSM fragment

Wait for 0 from below

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

Transport Layer 3-13
```
**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

**Q:** how to deal with loss?
- sender waits until certain data or ACK lost, then retransmits
- drawbacks?

**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**

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

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)
  stop_timer

timeout
  udt_send(sndpkt)
  start_timer

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

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
  stop_timer

rdt_rcv(rcvpkt)
  Wait for call 1 from above

rdt_rcv(rcvpkt)
  Wait for ACK0

rdt_rcv(rcvpkt)
  Wait for call 0 from above
```

---

**Transport Layer 3-15**
**rdt3.0 in action**

- **1.** Operation with no loss:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends ACK 0.

- **2.** Lost packet:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends packet 1.
  - Receiver receives packet 1 and sends ACK 1.
  - Receiver sends ACK 1.

- **3.** Lost ACK:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends packet 1.
  - Receiver receives packet 1 and sends ACK 1.
  - Receiver sends packet 1 (detect duplicate) and sends ACK 1.

- **4.** Premature timeout:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends packet 1.
  - Receiver receives packet 1 and sends ACK 1.
  - Receiver sends packet 1 (detect duplicate) and sends ACK 1.

*Transport Layer 3-17*

---

**rdt3.0 in action**

- **1.** Operation with no loss:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends ACK 0.

- **2.** Lost packet:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends packet 1.
  - Receiver receives packet 1 and sends ACK 1.
  - Receiver sends ACK 1.

- **3.** Lost ACK:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends packet 1.
  - Receiver receives packet 1 and sends ACK 1.
  - Receiver sends packet 1 (detect duplicate) and sends ACK 1.

- **4.** Premature timeout:
  - Sender sends packet 0 and receives ACK 0.
  - Receiver sends packet 1.
  - Receiver receives packet 1 and sends ACK 1.
  - Receiver sends packet 1 (detect duplicate) and sends ACK 1.

*Transport Layer 3-18*
**Performance of rdt3.0**

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L}{R} = \frac{8\text{kb/pkt}}{10^{-9} \text{ b/sec}} = 8 \text{ microsec} \]

\[ \text{Throughput} = \frac{L}{\text{RTT} + L / R} = \frac{8000}{0.030008} = 267 \text{ Kbps} \]

- 1KB pkt every 30 msec -> 267Kbps throughput over 1 Gbps link
- Utilization - fraction of time link is busy sending = 0.000267
- network protocol limits use of physical resources!

**rdt3.0: stop-and-wait operation**

\[ \text{Throughput} = \frac{L}{\text{RTT} + L / R} = \frac{8000}{0.030008} = 267 \text{ Kbps} \]
Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets
- 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

Throughput = \frac{3 \cdot L}{RTT + \frac{L}{R}} = \frac{24000}{0.030008} = 800 \text{ Kbps}
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

```
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)
} else
    refuse_data(data)
```

```
base = getacknum(rcvpkt)+1
if (base == nextseqnum)
    stop_timer
else
    start_timer
```

Transport Layer 3-23
GBN: receiver extended FSM

GBN in action
**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**

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
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

[Diagram showing the process with packets and ACKs]
**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?

---

**TCP: Overview**

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers

- full duplex data:
  - bi-directional data flow in same connection
- MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

RFCs: 793, 1122, 1323, 2018, 2581
TCP segment structure

<table>
<thead>
<tr>
<th>Source port</th>
<th>Dest port</th>
<th>Sequence number</th>
<th>Acknowledgement number</th>
<th>Receive window</th>
<th>Urg data pointer</th>
<th>Options (variable length)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum** (as in UDP)

TCP seq. #'s and ACKs

**Seq. #'s:**
- byte stream "number" of first byte in segment's data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

**Q**: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor

- host ACKs receipt of 'C', echoes back 'C'
- simple telnet scenario
TCP: reliable data transfer

simplified sender, assuming
- one way data transfer
- no flow, congestion control

wait for event

event: data received from application above
create, send segment

event: timer timeout for segment with seq # y
retransmit segment

event: ACK received, with ACK # y
ACK processing

Comment:
- SendBase-1: last cumulatively ack’ed byte
Example:
- SendBase-1 = 71; y = 73, so the rcvr wants 73+; y > SendBase, so that new data is acked
**TCP ACK generation** [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps, everything else already ACKed</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps, one delayed ACK pending</td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>out-of-order segment arrival higher-than-expect seq. # gap detected</td>
<td>send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>arrival of segment that partially or completely fills gap</td>
<td>immediate ACK if segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

**TCP: retransmission scenarios**

- **Host A** Seq=92, 8 bytes data ACK=100
- **Host B** Seq=92, 8 bytes data

- **Host A** Seq=100, 20 bytes data ACK=100
- **Host B** Seq=92, 8 bytes data

- **Host A** Seq=100 timeout
- **Host B** ACK=120

- **Host A** Seq=92, 8 bytes data
- **Host B** Seq=92 timeout

Lost ACK scenario

Premature timeout, cumulative ACKs
Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs:
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```java
event: ACK received, with ACK field value of y
if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments) {
        start timer
    }
} else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
        resend segment with sequence number y
    }
}
```

- A duplicate ACK for already ACKed segment
- Fast retransmit
TCP Round Trip Time and Timeout

**Q:** how to set TCP timeout value?

- longer than RTT
  - note: RTT will vary
- too short: premature timeout
- unnecessary retransmissions
- too long: slow reaction to segment loss

**Q:** how to estimate RTT?

- **SampleRTT:** measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
  - **SampleRTT** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **SampleRTT**

EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

Setting the timeout

- EstimatedRTT plus "safety margin"
- large variation in EstimatedRTT -> larger safety margin

Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation + x*|SampleRTT-EstimtedRTT|
**Example RTT estimation:**

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>RTT (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>150</td>
</tr>
<tr>
<td>15</td>
<td>200</td>
</tr>
<tr>
<td>22</td>
<td>250</td>
</tr>
<tr>
<td>29</td>
<td>300</td>
</tr>
<tr>
<td>36</td>
<td>350</td>
</tr>
<tr>
<td>43</td>
<td>400</td>
</tr>
<tr>
<td>50</td>
<td>450</td>
</tr>
<tr>
<td>57</td>
<td>500</td>
</tr>
<tr>
<td>64</td>
<td>550</td>
</tr>
<tr>
<td>71</td>
<td>600</td>
</tr>
<tr>
<td>78</td>
<td>650</td>
</tr>
<tr>
<td>85</td>
<td>700</td>
</tr>
<tr>
<td>92</td>
<td>750</td>
</tr>
<tr>
<td>99</td>
<td>800</td>
</tr>
<tr>
<td>106</td>
<td>850</td>
</tr>
</tbody>
</table>

**TCP Flow Control**

- **receive side of TCP connection has a receive buffer:**
  - RevWindow
  - data from IP
  - RevBuffer
  - TCP data to buffer
  - application process

- **app process may be slow at reading from buffer**

- **flow control**
  - sender won't overflow receiver's buffer by transmitting too much, too fast

- **speed-matching service:** matching the send rate to the receiving app's drain rate
TCP Flow control: how it works

(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
  = RcvWindow
  = RcvBuffer - \([\text{LastByteRcvd} - \text{LastByteRead}]\)

- Rcvr informs sender of spare room in buffer
- $\text{RcvWindow}$ field in TCP segment
- Sender limits unACKed data to $\text{RcvWindow}$
- guarantees receive buffer doesn't overflow

TCP Connection Management

**Recall:** TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. $\text{RcvWindow}$)
- client: connection initiator
  \[\text{Socket clientSocket = new Socket("hostname","port number")};\]
- server: contacted by client
  \[\text{Socket connectionSocket = welcomeSocket.accept();}\]

**Three way handshake:**

**Step 1:** client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

**Step 2:** server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
**Closing a connection:**

client closes socket:
```
clientSocket.close();
```

**Step 1:** client end system
sends TCP FIN control segment to server

**Step 2:** server receives
FIN, replies with ACK.
Closes connection, sends FIN.

**Step 3:** client receives FIN,
replies with ACK.
- Enters "timed wait" -
  will respond with ACK
to received FINs

**Step 4:** server, receives
ACK. Connection closed.

**Note:** with small
modification, can handle
simultaneous FINs.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle

Transport Layer 3-49