## 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**

Source: Object Reference and Their Interfaces. Material from textbooks is copyrighted by appropriate authors. For example: All material copyright 1996-2006, J.F. Kurose and K.W. Ross, All Rights Reserved.

---

### Reliable data transfer: base design

- **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.
- **send side**
- **receive side**
- **rdt_send**: called by rdt, to transfer packet over unreliable channel to receiver.
- **rdt_rev**: called when packet arrives on rcv-side of channel.
- **deliver_data**: called by rdt to deliver data to upper.

---

### Rdt2.0: reliable transfer over an unreliable channel

- channel with bit errors
  - underlying channel may flip bits in packet
- no packet loss

- Checksum in the header
- ACK/NAK: receiver -> sender
- Sender retransmits on NAK
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 garbled?
- 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

1-bit sequence numbers

Send packet:
- sndpkt = make_pkt(0, data, checksum)
- udt_send(sndpkt)
- rdt_send(data)

Wait for call 0 from above
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
- sndpkt = make_pkt(1, data, checksum)
- udt_send(sndpkt)
- rdt_send(data)

Wait for call 1 from above
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
- udt_send(sndpkt)
**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

Can we avoid 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**

Sender FSM fragment:
- `rdt_send(data)`
- `udt_send(sndpkt)`
- `wait for 0 from above`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`

Receiver FSM fragment:
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `& & has_seq1(rcvpkt)`
- `delivrdt(rcvpkt, data)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK0, checksum)`
- `udt_send(sndpkt)`

---

Can we avoid NAKs?
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
  • yuck: 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

rdt3.0 in action
**Performance of rdt3.0**  

- rdt3.0 works, but performance *stinks*
- example: 1 Gbps link, 15 ms end-end delay (30ms RTT), 1KByte pkt:

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

- \( U_{\text{sender}} = \frac{L}{R} + \frac{L}{R} \quad \frac{30}{30} = 0.00027 \)

- Utilization - fraction of time sender busy sending
- Assume ACK pkts small and thus \( T_{\text{transmit}} \approx 0 \)
- Effective throughput 270Kbps over a 1Gbps link!!
- 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**

- First packet bit transmitted, \( t = 0 \)
- Last packet bit transmitted, \( t = \frac{L}{R} \)
- ACK arrives, send next packet, \( t = \frac{L}{R} + \frac{L}{R} \)
- \( U_{\text{sender}} = \frac{L}{R} + \frac{L}{R} \quad \frac{30}{30} = 0.00027 \)

**How to improve?**
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, \( t = 0 \)
Sender

Last packet bit transmitted, \( t = \frac{L}{R} \)
Receiver

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

Increase utilization by a factor of 3!

Go-Back-N

Sender:
- k-bit seq # in pkt header
- "window" of up to N, consecutive unAcked pkts allowed

Next design timer for the oldest unacked pkt (send_base)

- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
- may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timer to retransmit pkt n and all higher seq # pkts in window
GBN: sender extended FSM

GBN: receiver extended FSM

GBN in action
GBN applet:

http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html