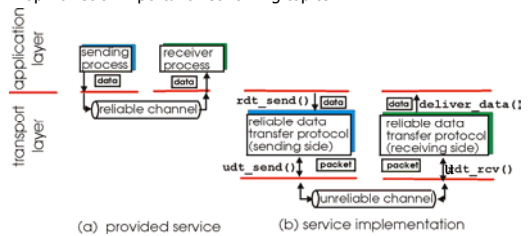


Reliable Data Transfer

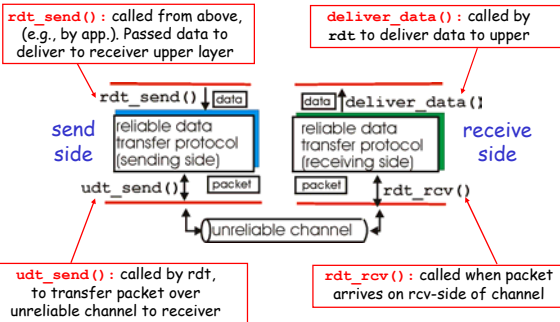
Principles of Reliable data transfer

- 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



Reliable data transfer: getting started

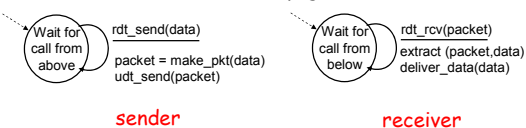
We'll:

- 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 read data from underlying channel



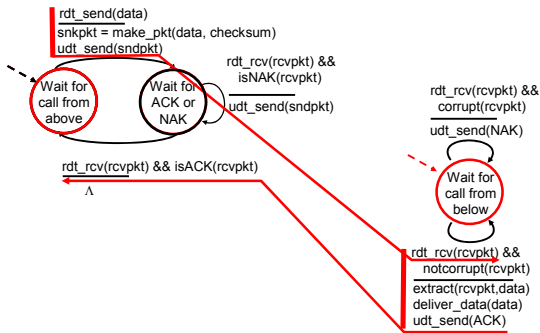
Rdt2.0: channel with bit errors

- 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):

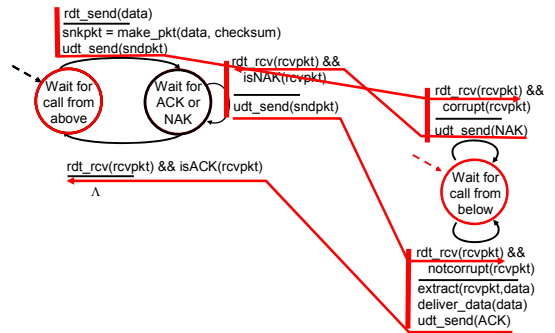
- Error detection
- Receiver feedback: control msgs (ACK,NAK) rcvr->sender

rdt2.0: operation with no errors



7

rdt2.0: error scenario



8

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!

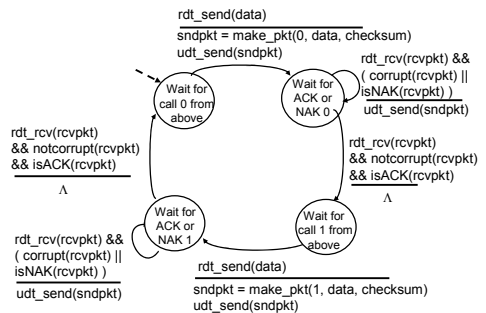
Handling duplicates:

- Sender retransmits current pkt if ACK/NAK garbled
- Sender adds *sequence number* to each pkt
- Receiver discards (doesn't deliver up) duplicate pkt

stop and wait
Sender sends one packet, then waits for receiver response

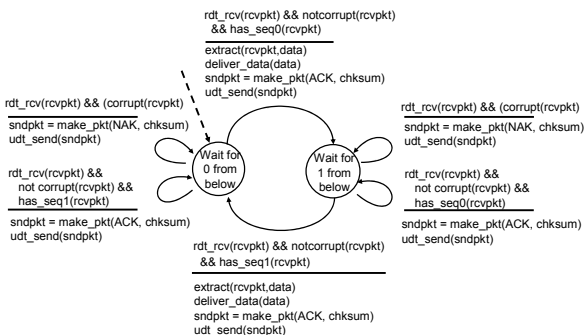
9

rdt2.1: sender, handles garbled ACK/NAKs



10

rdt2.1: receiver, handles garbled ACK/NAKs



11

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

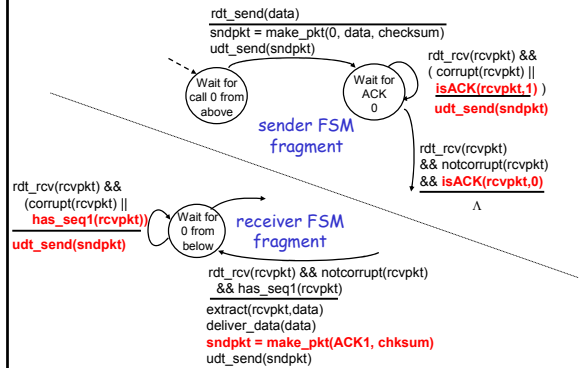
12

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*

13

rdt2.2: sender, receiver fragments



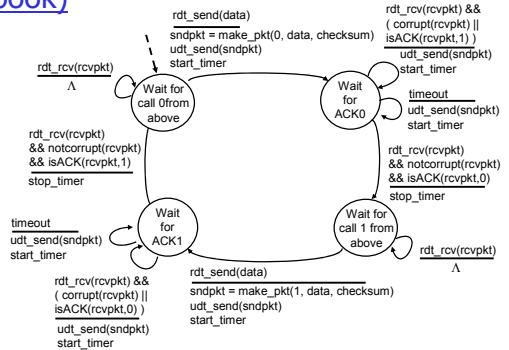
14

rdt3.0: channels with errors and loss

- New assumption:** underlying channel can also lose packets (data or ACKs)
- Approach:** sender waits "reasonable" amount of time for ACK
- Checksum, seq. #, ACKs, retransmissions will be of help, but not enough
 - 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
- Q:** How to deal with loss?
- Sender waits until certain data or ACK lost, then retransmits

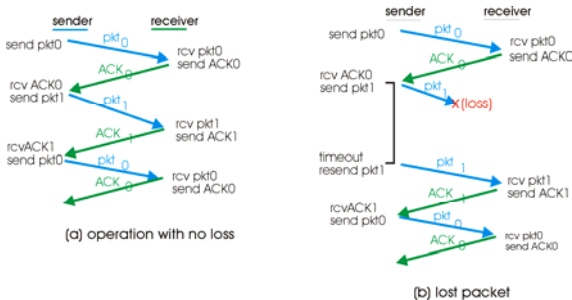
15

rdt3.0 sender (slightly different from book)



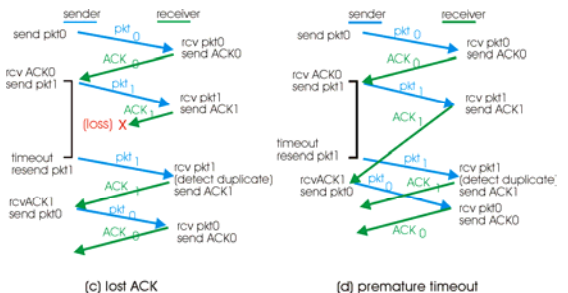
16

rdt3.0 in action



17

rdt3.0 in action



18

Performance of rdt3.0

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

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb}/\text{pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

- U_{sender} : **utilization** – fraction of time sender busy sending

$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- 1KB pkt every 30 msec -> 33kB/sec thrupt over 1 Gbps link
- Network protocol limits use of physical resources!

19

rdt3.0 sender in perl

- use IO::Select for timer
- do a can_read() with appropriate timeout
- when can_read() returns:
 - timeout expired? ==> retransmit, repeat can_read()
 - socket ready? ==> read packet
 - corrupt? ==> retransmit, repeat can_read()
 - wrong ACK# ==> retransmit, repeat can_read()
 - not corrupt and correct ACK# ==> wait for call from above

20