Q1

a. network layer provides logical-communication between hosts, transport layer provides logical-communication between application processes running on different hosts.

b. This job of delivering the data in a transport-layer segment to the correct socket is called demultiplexing. The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called multiplexing.

Q2

a.

Destination port (4 bytes)	
length	checksum
data	

b.

Source port (2 bytes)	Destination port (2 bytes)
length	Checksum
data	

c.

No. Transport-layer protocols live in the end systems. Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa, but it doesn't have any say about how the messages are moved within the network core.

Q3

a. Source port: 467 Destination port: 23

b. Source port: 513 Destination port: 23

c. Source port: 23 Destination port: 467

d. Source port: 23 Destination port: 513

e. Yes

f. No

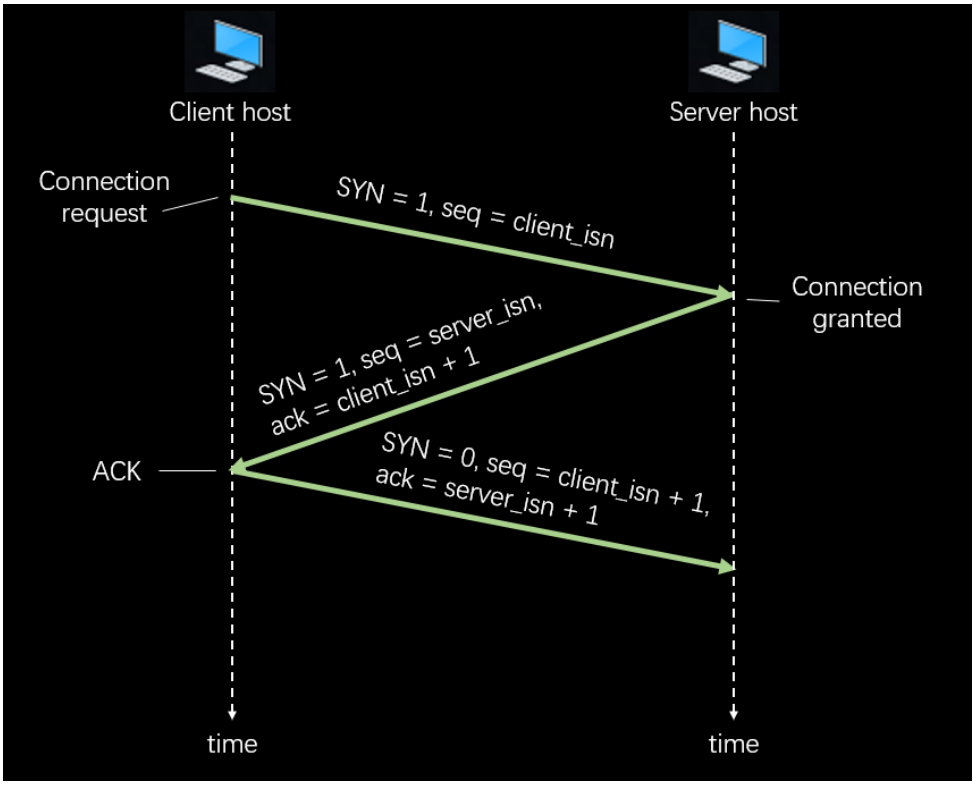
Q4

a. Many firewalls are configured to block (most types of) UDP traffic because there is no handshaking in UDP protocols. Another reason is that UDP has no congestion control which may make the internet enter a congested state.

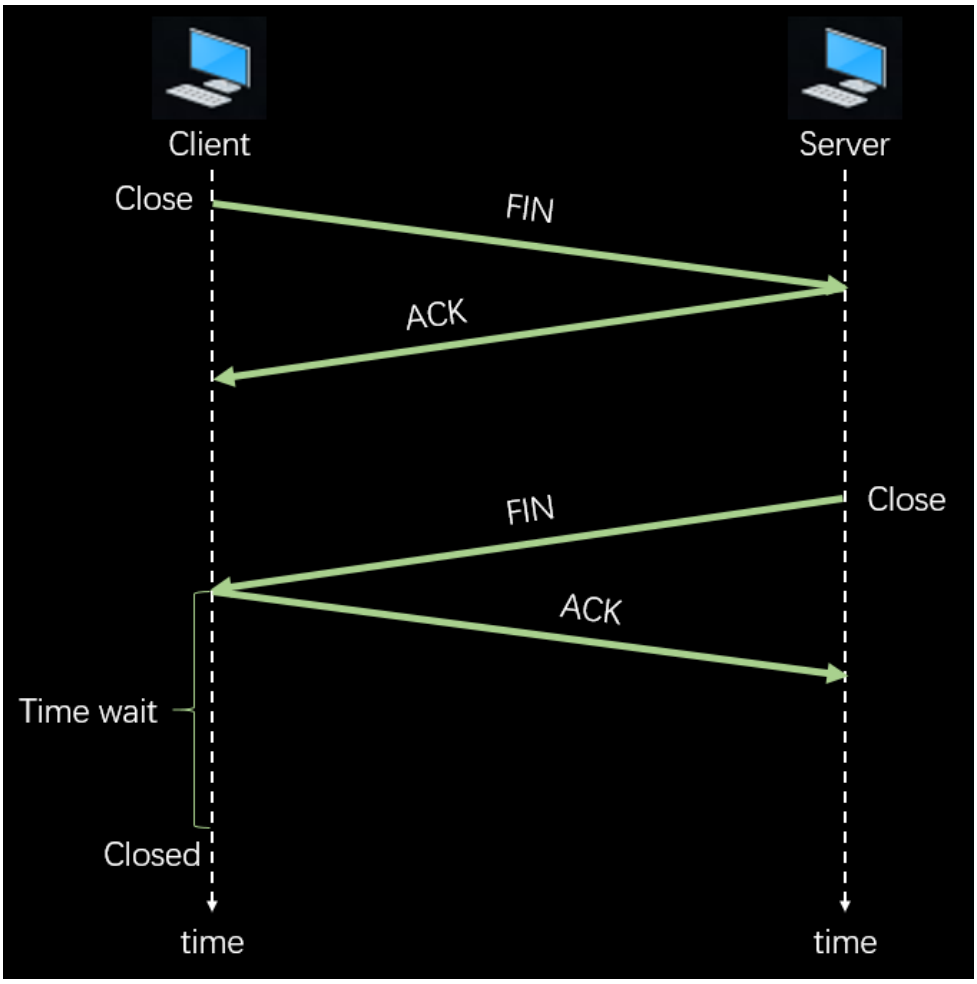
b. Yes. Developer can put reliable data transfer protocol in application layer.

Q5

a.

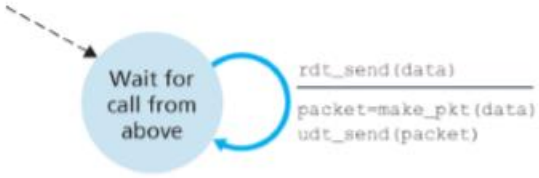

Client host
Server host
Connection request
SYN = 1, seq = client_isn
Connection granted
SYN = 1, seq = server_isn, ack = client_isn + 1
ACK
SYN = 0, seq = client_isn + 1, ack = server_isn + 1
time
time

b.

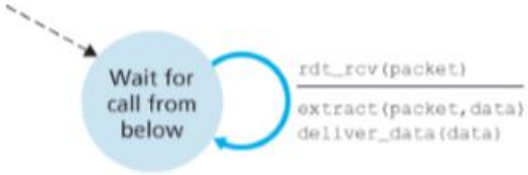

Client
Server
Close
FIN
ACK
FIN
Close
ACK
Time wait
Closed
time
time

Q6

rdt1.0: Both sending side and receiving side has only one state. The underlying channel is supposed to be completely reliable. On the sending side, once it sends a packet, it returns the state of waiting for call from above. On the receiving side, once it receives a packet, it returns the state of wait for call from below.

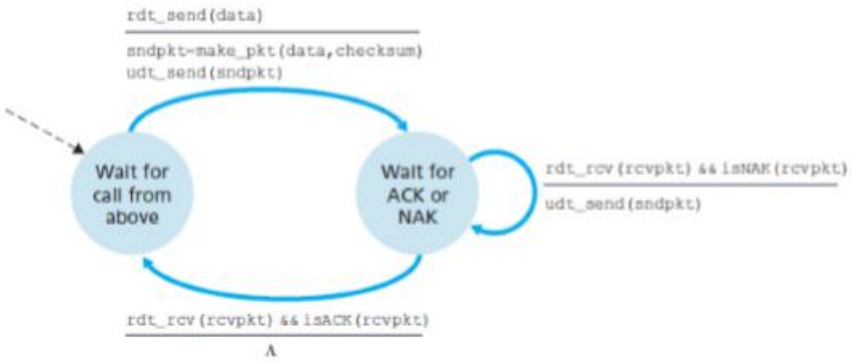


a. rdt1.0: sending side

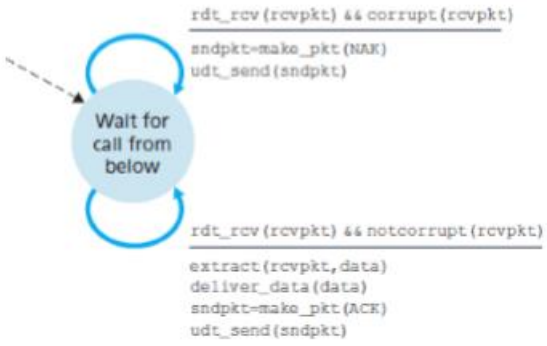


b. rdt1.0: receiving side

rdt2.0: In rdt2.0, an error detection is added into the system. On the sending side, there are two states which are “wait for call from above” and “wait for ACK or NAK”. On the receiving side, there is still one state which is “wait for call from below”. Once the sending side sends a packet to the receiving side, it switches to the “wait for ACK or NAK” state. When the receiving side receives the packet from the sending side, it checks the packet whether it has a bit error through the checksum in the packet. If there is a bit error, it returns a “NAK” to the sending side and the sending side will resend the packet and stays in the “wait for ACK or NAK” state. If there is no bit error, it returns an “ACK” to the sending side and the sending side switches to the “wait for call from above” state.



a. rdt2.0: sending side



b. rdt2.0: receiving side

rdt2.1: There is still a problem in the rdt2.0 which is the bit error of the "ACK" or "NAK" packet sent from receiving side. So rdt2.1 adds two states in sender and one state in receiver. The packets that are sent have a sequence number of 0 or 1. Once the sender receives a call 0 from above, it sends the packet to the receiver which has a sequence number of 0 and switch to the state of "wait for ACK or NAK 0". Once the receiver receives the packet, if the packet doesn't have a bit error, it returns an "ACK 0" packet to the receiver and switches to the "wait for 1 from below" state. If the packet has a bit error, it returns a "NAK 0" packet to the receiver and stays in the "wait for 0 from below" state. Once the receiver receives an "ACK 0" packet, it switches to "wait for call 1 from above" state. Once the receiver receives an "NAK 0" packet, it sends the packet 0 again and stays in the "wait for ACK or NAK 0" state. Once the receiver receives a packet with bit error, it sends the packet 0 again and stays in the "wait for ACK or NAK 0" state. When the receiver receives a packet 0 again in the "wait for 1 from below" state, it returns the "ACK 0" again to the sender and stays in the state.

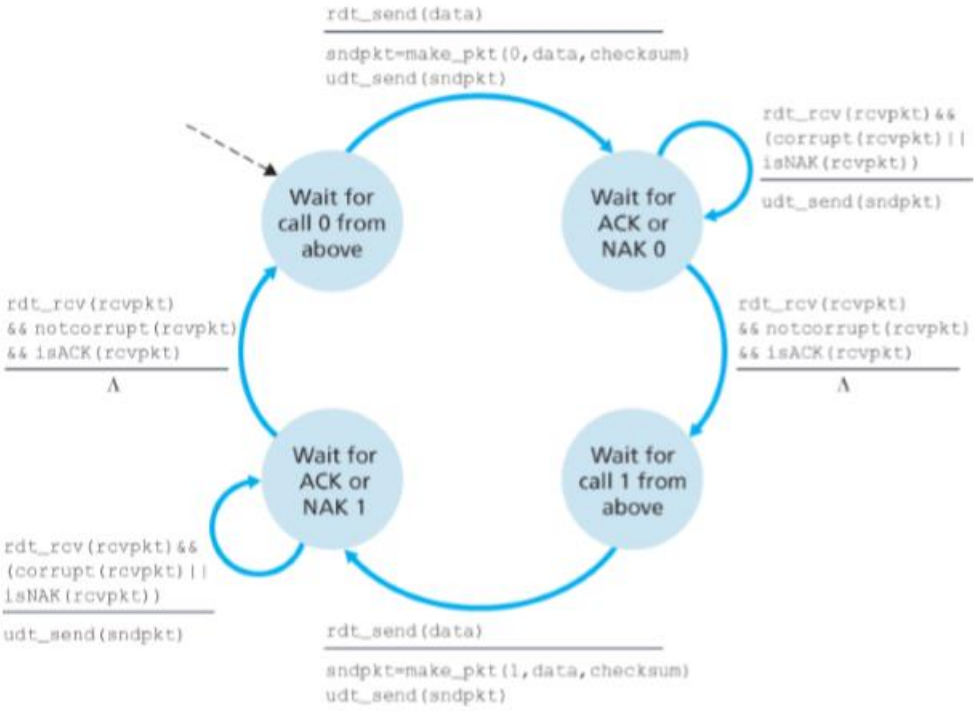


Figure 3.11 *rdt2.1* **sender**

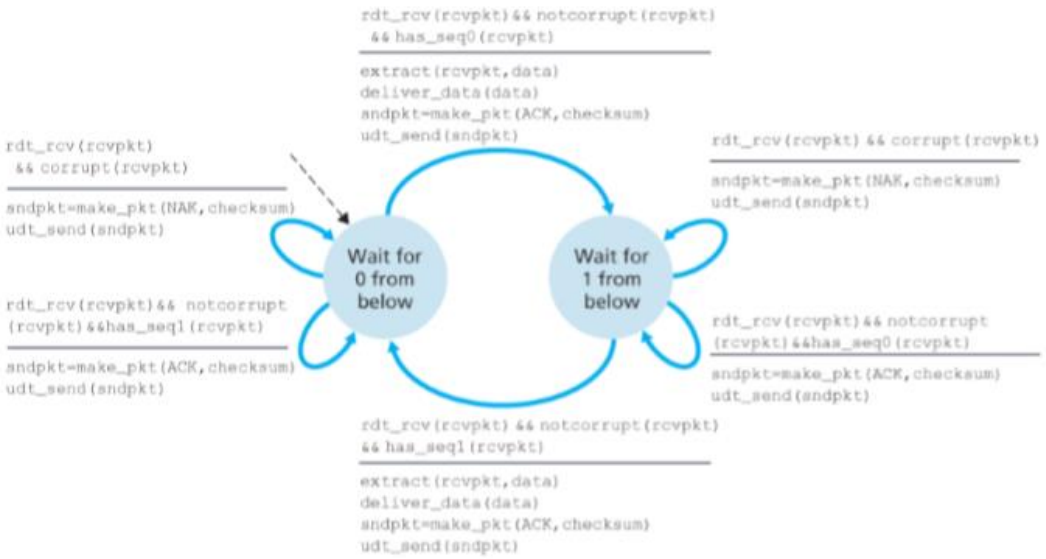


Figure 3.12 *rdt2.1* **receiver**

rdt2.2: To avoid the packet losing problem, a timer should be started once the packet is sent. However, the packet may not be lost but only been delayed., then the duplicate packet problem will occur. rdt2.2 changes the "NAK" packet in rdt2.1 to giving a sequence number of "ACK" packet to solve the duplicate packet problem. When the receiver receives a packet 0 with error bit, it will return an "ACK 1" instead of "NAK" packet. So that whether the packet 1 is duplicated or the packet 0 has a bit error, the sender will send the packet 0 again.

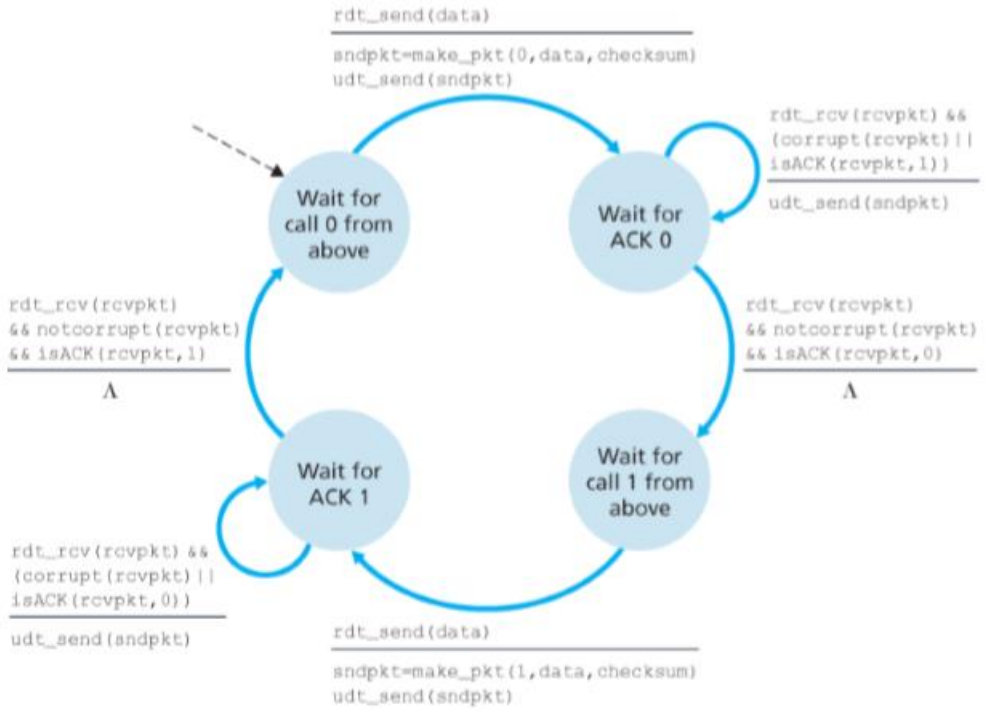


Figure 3.13 *rdt2.2* **sender**

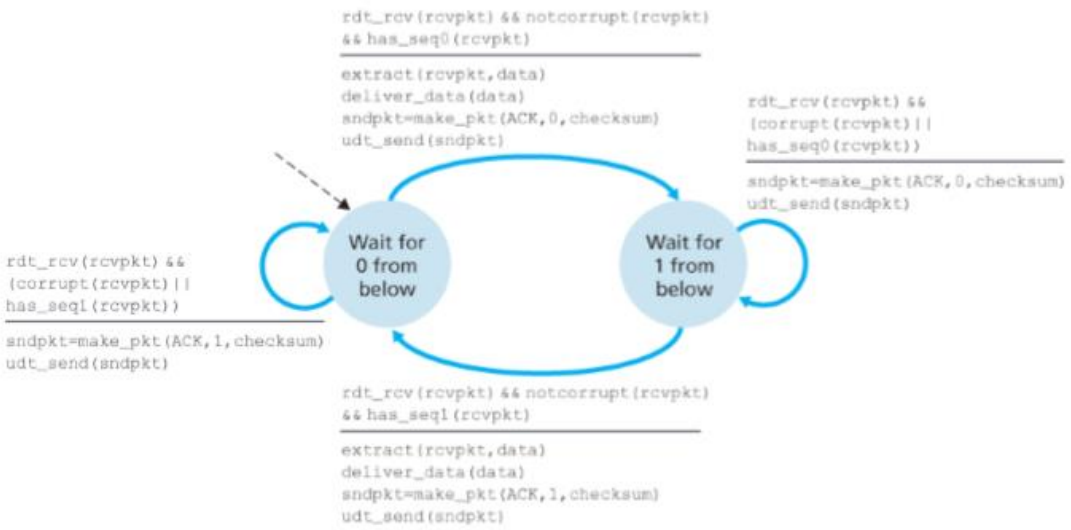


Figure 3.14 *rdt2.2* **receiver**

rdt3.0: rdt3.0 finally adds the timer to the system which solves the packet losing problem. Once a packet is sent from the sender, a timer will be started. Whether the packet is lost or the ACK from receiver is lost, the sender will send the packet again to the receiver. The receiver is the same as the receiver of rdt2.2.

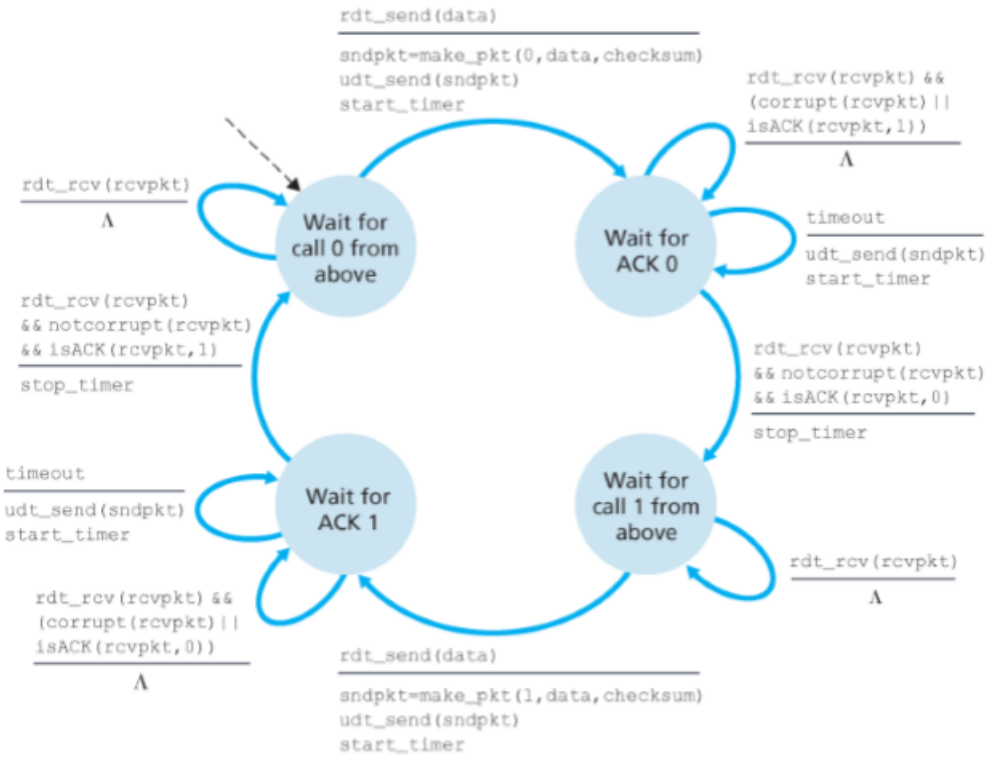
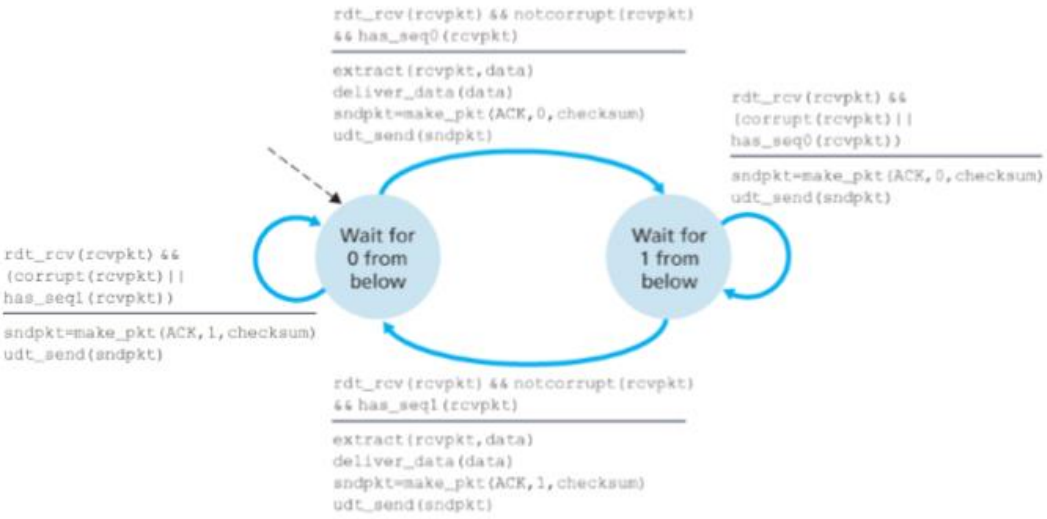


Figure 3.15 *rdt3.0* **sender**


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt)||
has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
Wait for
0 from
below
Wait for
1 from
below
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt)||
has_seq1(rcvpkt))
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)