

Names: _____

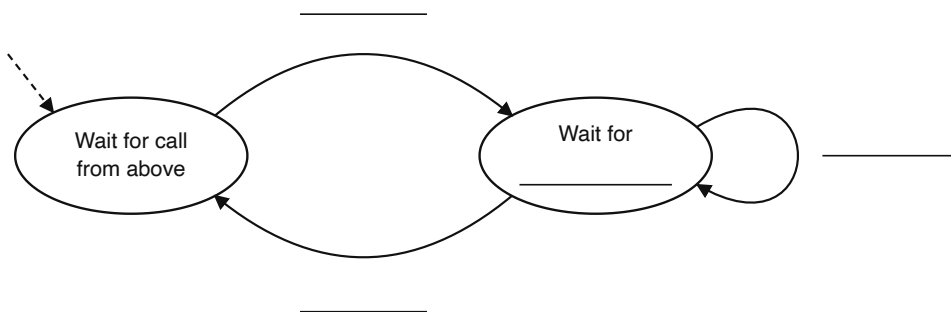
The channel under the transport layer can now **flip bits**, but it never loses or reorders a packet. A **checksum** lets whichever side receives a packet tell whether it arrived corrupted. rdt2.0's rules: the receiver answers every data packet with an **ACK** if it arrived intact or a **NAK** if it did not, and the sender retransmits whenever it gets a NAK. The sender is **stop-and-wait**: it sends one packet, then waits for the answer.

Each arrow in a state machine is labelled with the **event** that makes it fire (above the line) and every **action** the protocol takes when it does (below the line). `rdt_send()` is called by the application above; `rdt_rcv()` fires when a packet arrives from the channel below; `udt_send()` hands a packet to the unreliable channel; `deliver_data()` passes data up to the receiving application after a call to `extract()`.

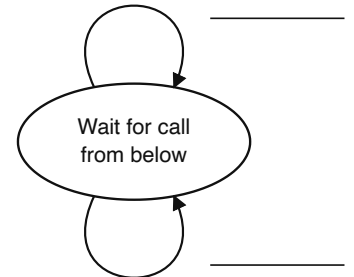
Events — each labels exactly one arrow	Actions — use as many as an arrow needs
A <code>rdt_rcv(rcvpkt) && corrupt(rcvpkt)</code>	1 <code>udt_send(sndpkt)</code>
B <code>rdt_send(data)</code>	2 <code>deliver_data(data)</code>
C <code>rdt_rcv(rcvpkt) && isNAK(rcvpkt)</code>	3 <code>udt_send(NAK)</code>
D <code>rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)</code>	4 <code>sndpkt = make_pkt(data, checksum)</code>
E <code>rdt_rcv(rcvpkt) && isACK(rcvpkt)</code>	5 <code>Λ // do nothing</code>
	6 <code>udt_send(ACK)</code>
	7 <code>extract(rcvpkt, data)</code>

1. Complete both machines: on each arrow write the event's letter above the line and its action numbers below, in the order they happen. Give the sender's second state a name that says what it is waiting for.

SENDER



RECEIVER



2. The sender needs two states; the receiver gets by with one. What is the sender remembering while it sits in its second state that the receiver never has to remember from one packet to the next?

Discuss with your partner

Your sender has no `rdt_send(data)` arrow out of its second state. What happens to an application that has more data ready while the sender is waiting? Would you want this sender on a satellite link where every ACK takes **500 ms** to come back?

Q2 Run rdt2.0, Then Break It Slides 51–53

Use your rdt2.0 machines from Q1. The application calls `rdt_send()` twice — first with data `d1`, then with `d2` — and both machines start in their *Wait for call* states. Each row of a trace is one packet crossing the channel, in the order it is sent. Row 1 is filled in as an example.

1. The channel corrupts the **first copy of `d1`** and nothing else. Trace the exchange until `d2` has been acknowledged. What does the receiving application end up with?

#	Direction	Packet	Arrives intact?	Actions at the side it reaches	Sender's state afterward
1	$S \rightarrow R$	data <code>d1</code>	no	<code>udt_send(NAK)</code>	Wait for ACK or NAK
2					
3					
4					
5					
6					
7					

The receiving application gets: _____

2. Start over. This time the first copy of `d1` arrives **intact**, but the receiver's response is corrupted on its way back. Which arrow in your sender machine does that garbled packet take? What does the receiver believe has happened to `d1`?

3. The sender cannot read the garbled response, so it has to guess what it was. For each case, what does the receiving application end up with once `d2` has gone through?

	It really was an ACK	It really was a NAK
Sender guesses ACK and moves on to <code>d2</code>		
Sender guesses NAK and resends <code>d1</code>		

Discuss with your partner

Slide 53's fix is to **always** resend when the response is garbled — it treats it like a NAK. Why that one and not the other? What would the receiver need in order to clean up afterwards? Have you seen this problem before?