CIS 457 - Data Communications

Nathan Bowman

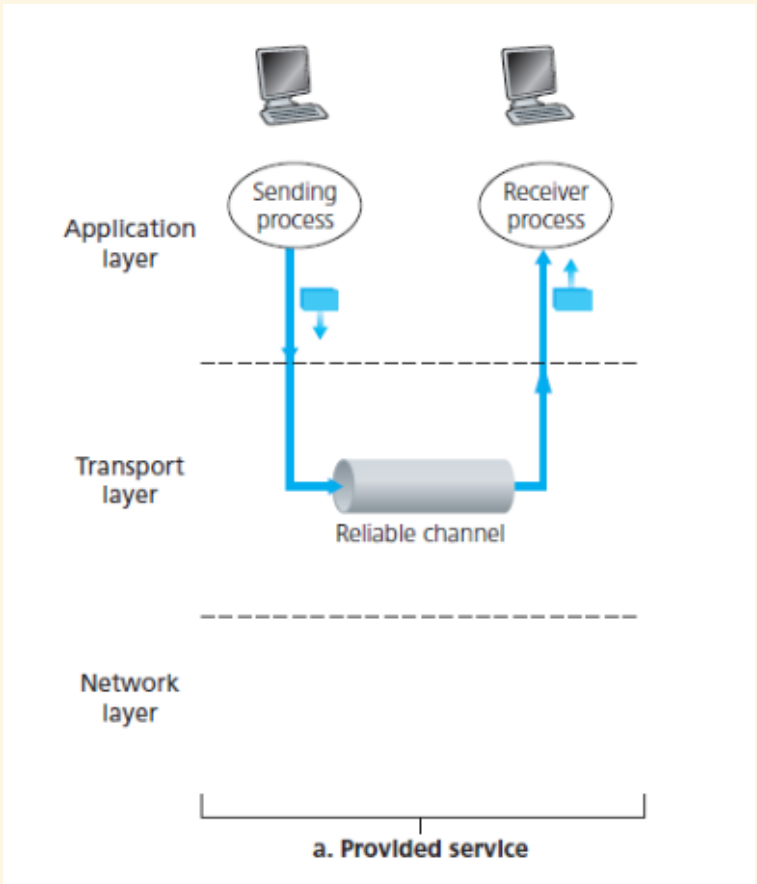
Images taken from Kurose and Ross book

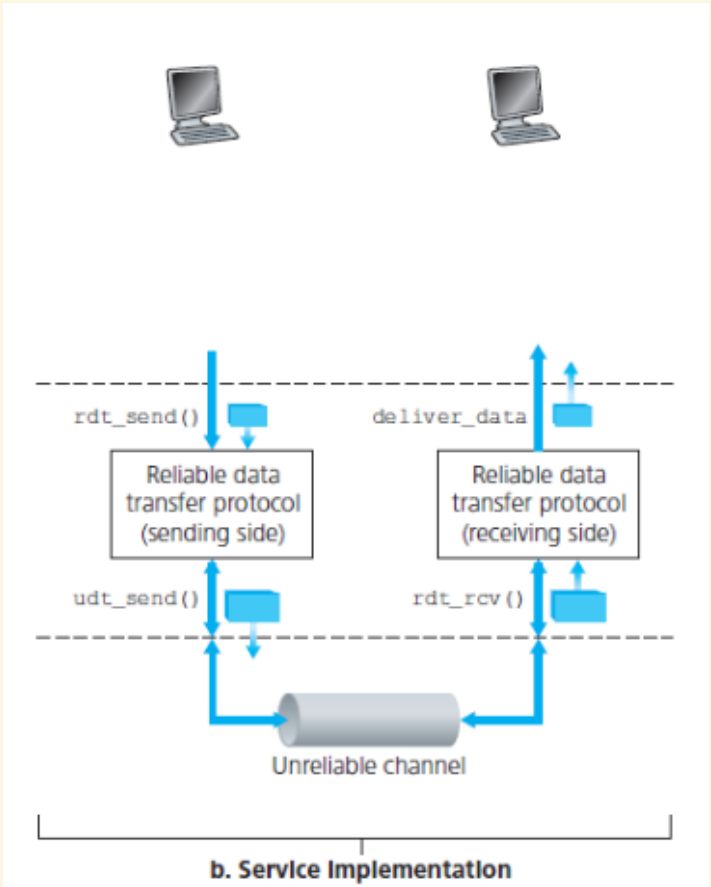
Reliable Data Transfer

Main advantage of TCP over UDP is reliable data transfer

Providing reliable data transfer over an unreliable infrastructure is possibly the most important aspect of networking

Before studying how TCP provides this, we will look at the problem in the abstract. Our analysis will also apply to other scenarios, such as sending over link layer


Sending process
Receiver process
Application layer
Transport layer
Reliable channel
Network layer
a. Provided service



b. Service Implementation

It is sufficient to consider a point-to-point channel even though communication will occur over various links and switches

Begin with simple model, then build to more realistic assumptions

Use finite-state machines (FSMs) to describe protocols

One assumption we will keep throughout is that packets are never reordered in network -- they may be lost or corrupted, but cannot arrive in different order than sent

Also, we will describe one-way communication

Two-way communication is not much more difficult, but is more tedious to describe

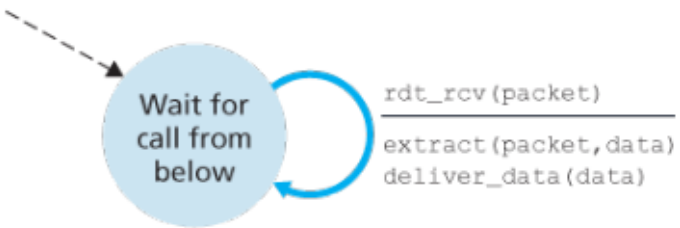
Although application messages will be sent only one way, control messages at lower layers may be sent in either direction

First, assume that network is perfectly reliable -- no lost or corrupt packets

Providing reliable delivery at transport layer is then trivial

We call resulting protocol Reliable Data Transfer 1.0 (rdt1.0)

a. rdt1.0: sending side



b. rdt1.0: receiving side

Next, assume channel does not drop packets, but may produce bit errors

That is, packets will always arrive, but they may be damaged

Imagine two people with bad telephone connection

If message is garbled and receiver cannot hear, they will ask sender to repeat message

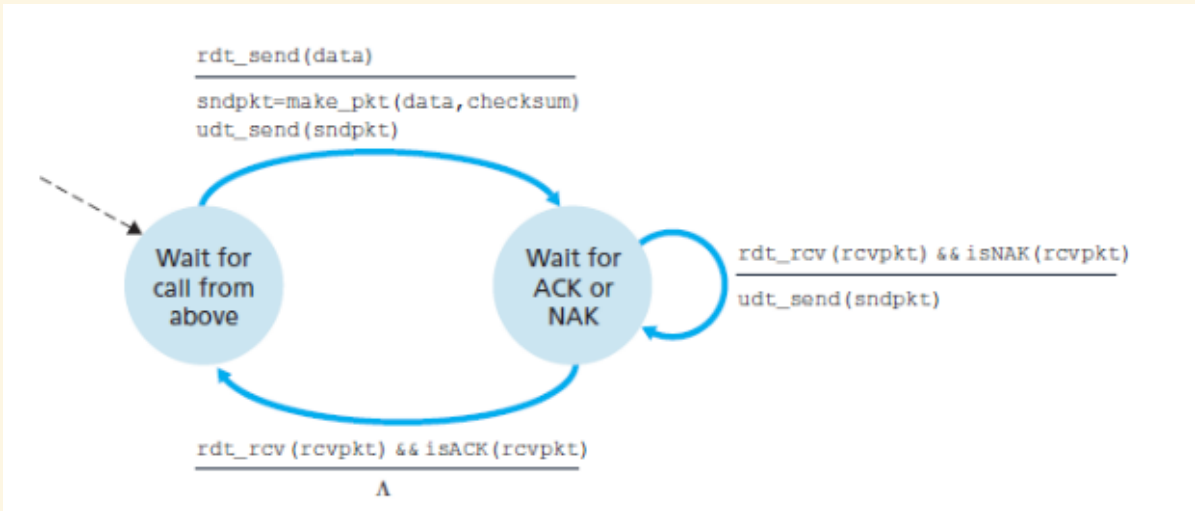
If message is understood, can reply "OK" to allow speaker to continue

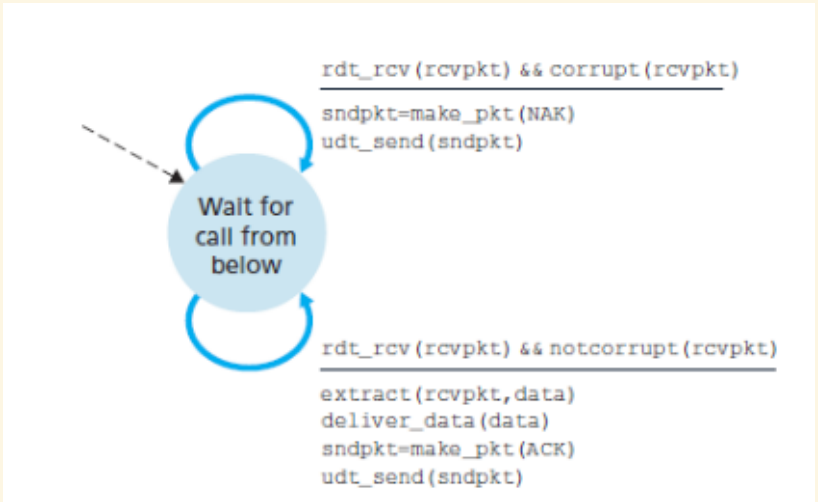
Same general idea applies to networks -- we call "OK" message **positive ACK** and "please repeat" message **negative ACK** (NAK)

Handling noisy channel will require three new capabilities:

- error detection (checksum)
- receiver feedback (ACK)
- retransmission of failed message

Bear in mind that sender and receiver do not know same information as one another -- for example, sender knows message was successful only after receiving ACK


rdt_send(data)
sndpkt=make_pkt(data,checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ


rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK)
udt_send(sndpkt)
Wait for call from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK)
udt_send(sndpkt)

Refer to new protocol as rdt2.0

Important to note that while sender is waiting for ACK, it cannot accept another message from application layer to send

Protocols like this referred to as **stop-and-wait** protocols

We will see later that these can have major impact on efficiency

More important issue: this protocol does not work

Dealt with bit errors from sender to receiver, but communication breaks down if ACK message has bit error

How could we fix this?

- ACK with bit error could be replied to with request to resend ACK. But, that reply could get corrupted as well, and we keep digging deeper
- Could add additional checksum bits to perform error recovery. However, this will not help later when we introduce packet loss to channel

Solution will be to resend original packet if corrupt ACK message is received

Essentially, treating corrupt ACK as NAK because sender cannot be sure message was received

This raises problem of its own: if original packet was not corrupted, sender just sent duplicate packet into channel

Receiver has no way of knowing whether received packet is actually duplicate or is new

Need to add sequence numbers to packets to allow receiver to detect duplicates

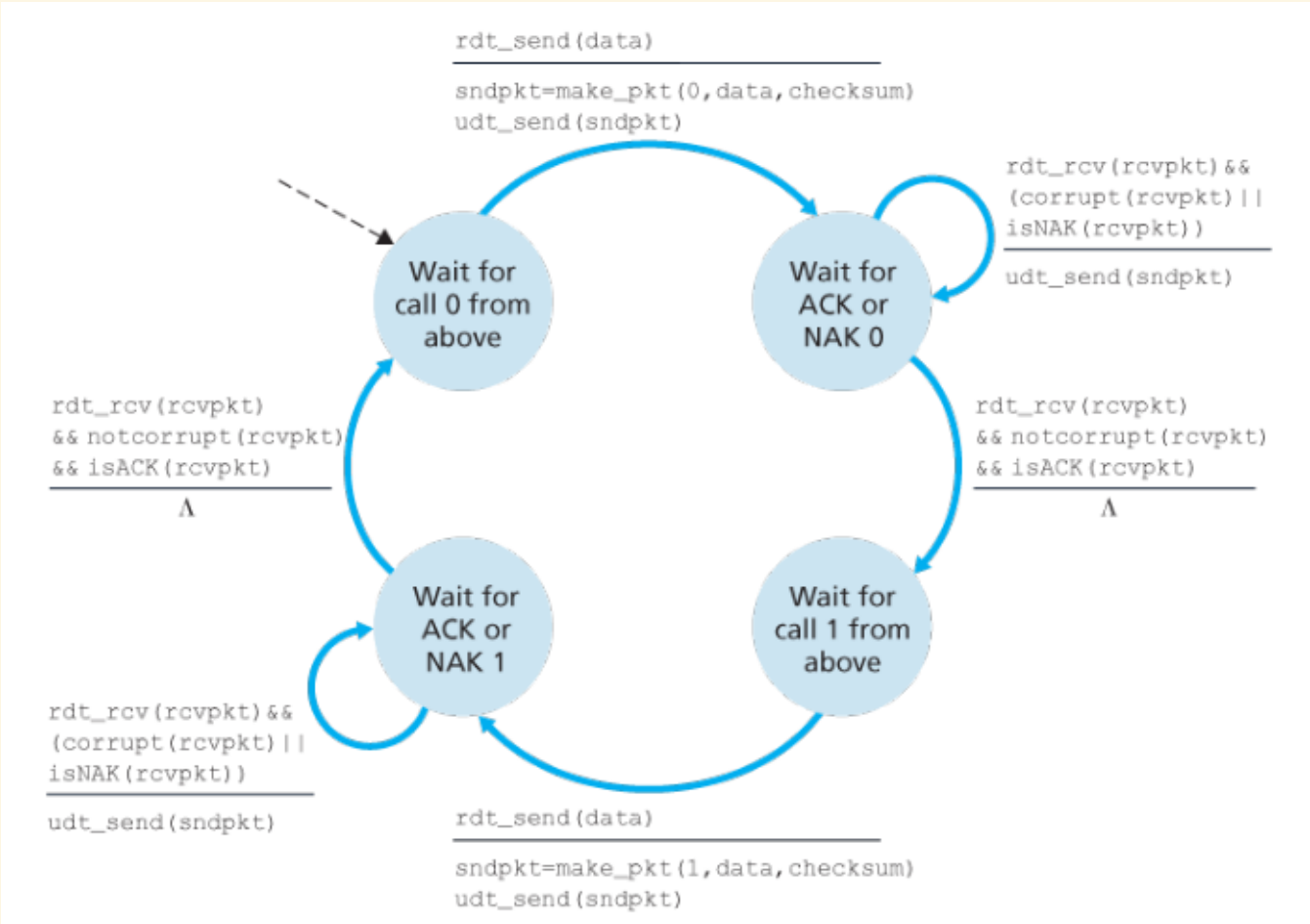
When using stop-and-wait protocol, sequence numbers need only be 0 or 1 to distinguish between duplicate and expected packets

ACK messages do not need sequence numbers in lossless channel because there is only one packet they could be acknowledging

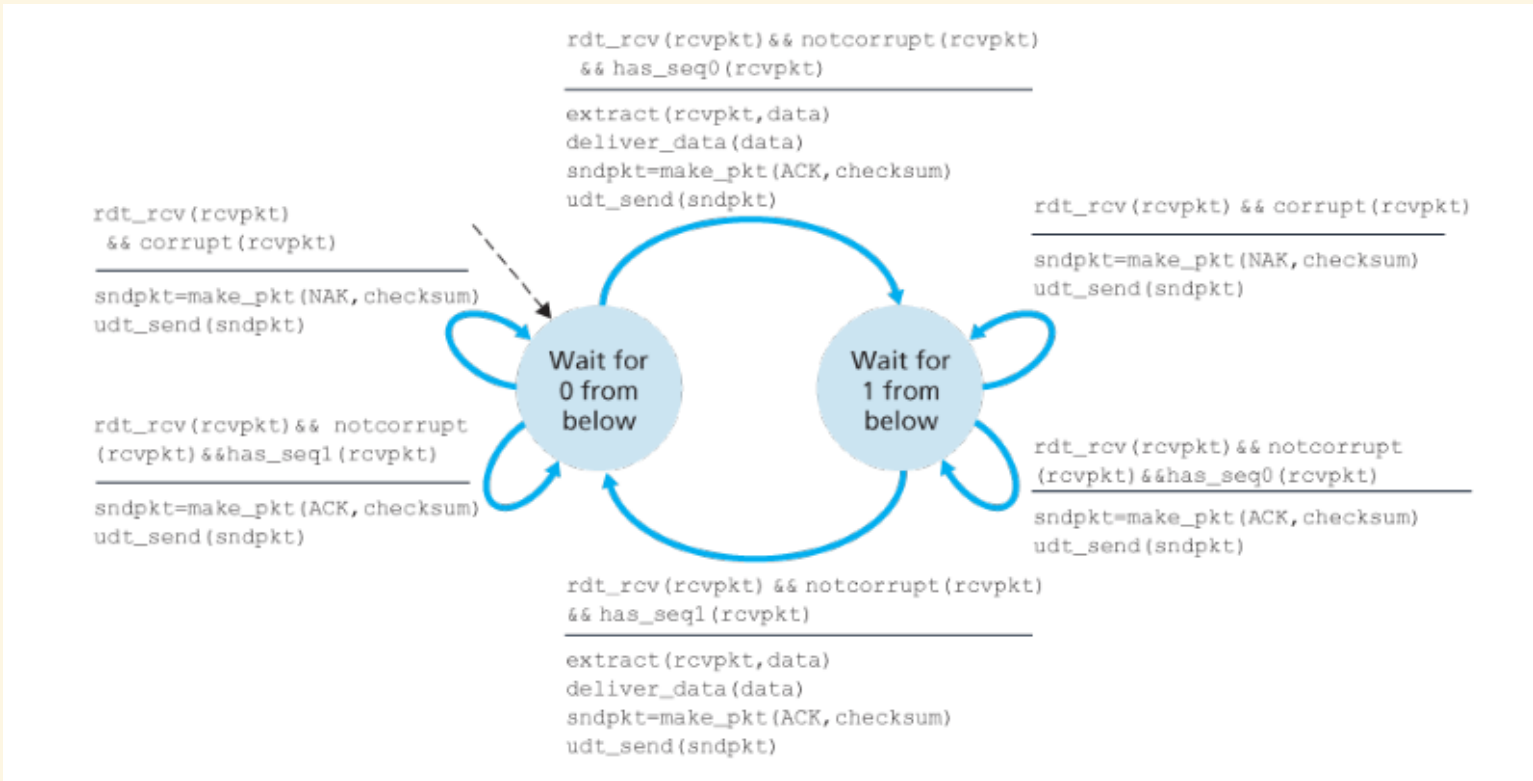
New protocol, rdt2.1, has more states to account for sequence numbers

However, pairs of states are similar but with sequence numbers flipped

Sender


rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt)&&
(corrupt(rcvpkt)||
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt)&&
(corrupt(rcvpkt)||
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)

Receiver

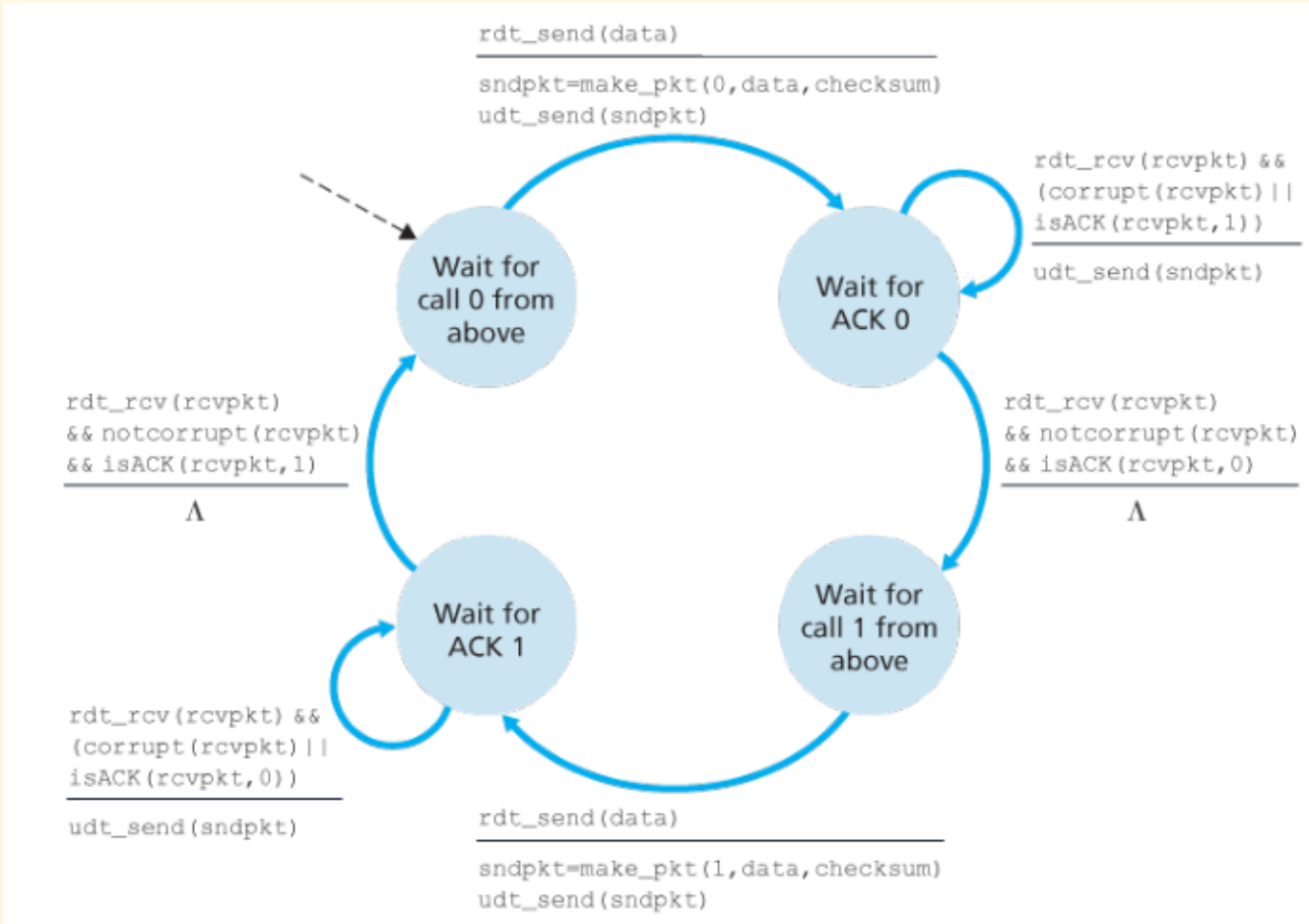

rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)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

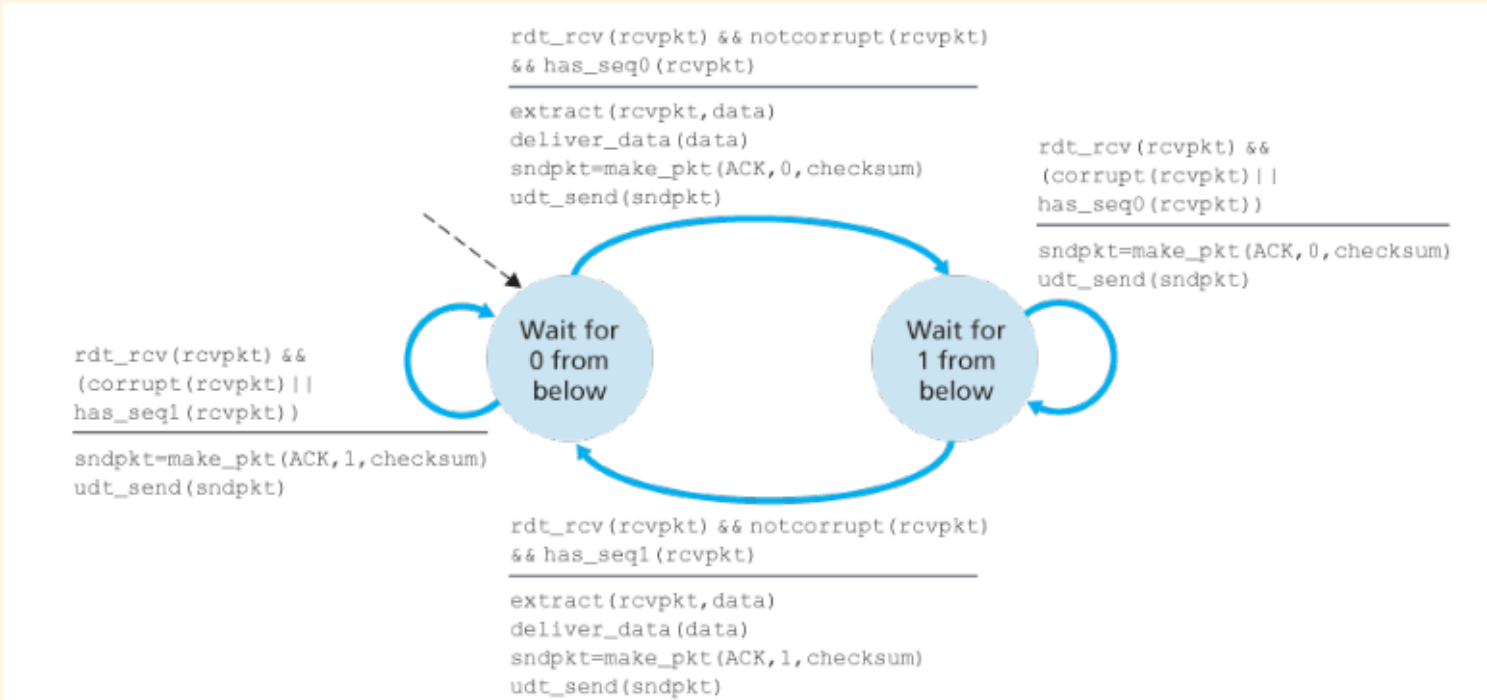
rdt2.1 is working protocol for this channel

We make one additional modification to simplify protocol: rather than send NAK when corrupt packet detected, send ACK for previous message

Has same effect (notifies sender of failure of most recent packet), but removes one type of message

For this to work, ACK now needs to have sequence number as well


rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
Wait for ACK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)


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)

Previous protocol referred to as rdt2.2

Final model we examine is channel that can produce bit errors and can lose packets

We already handle bit errors, so questions become

- how to detect lost packet?
- what to do about them?

Previous methods are enough to handle second question -- after all, lost packet is no worse than damaged packet

Detecting lost packets is simply a matter of waiting

Sender can approximate how long it would take for packet to be delivered and ACK to come back

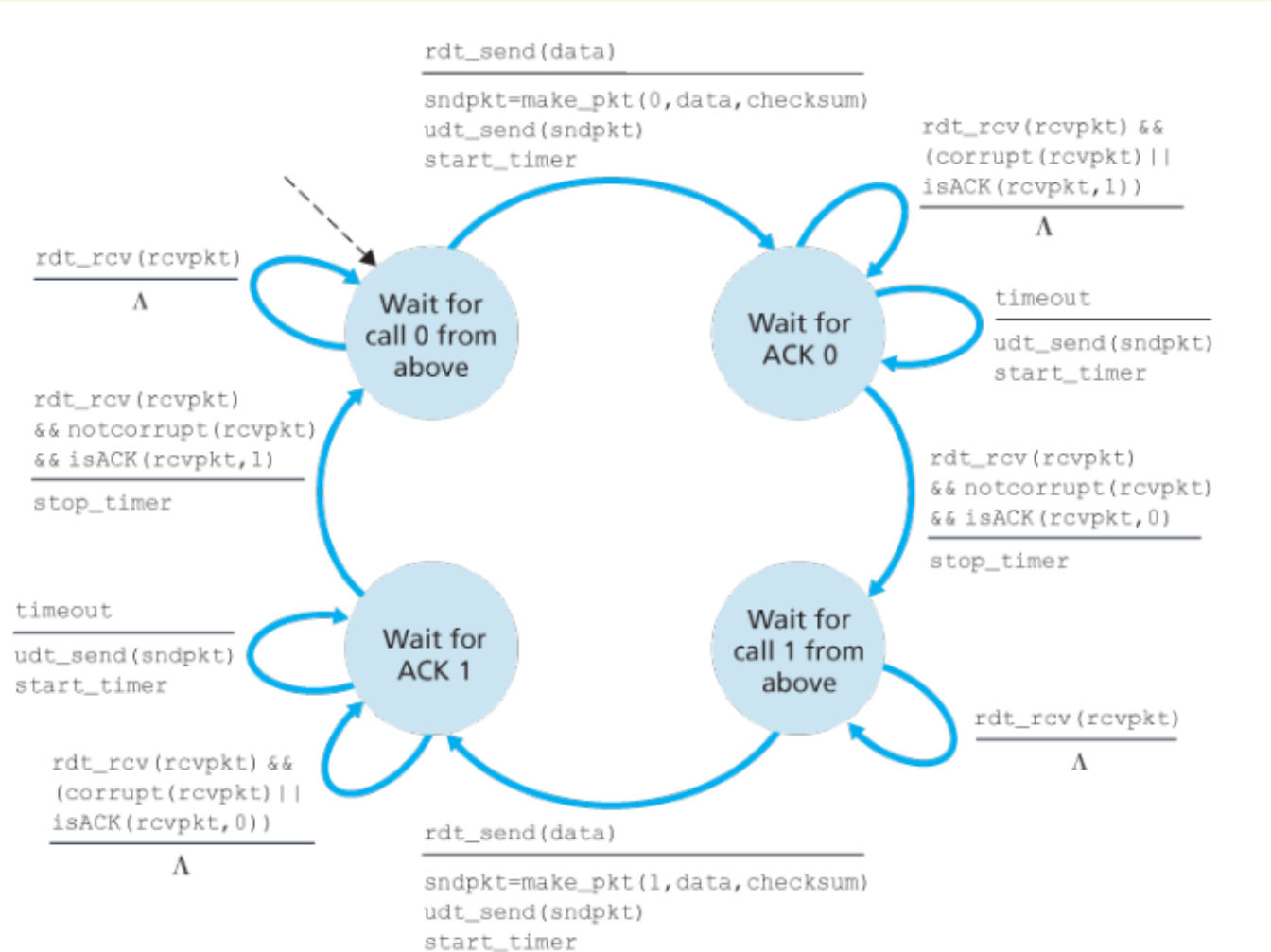
No need for sender to wait until it is "certain" packet was lost -- hard to know, and could spend excessive time waiting

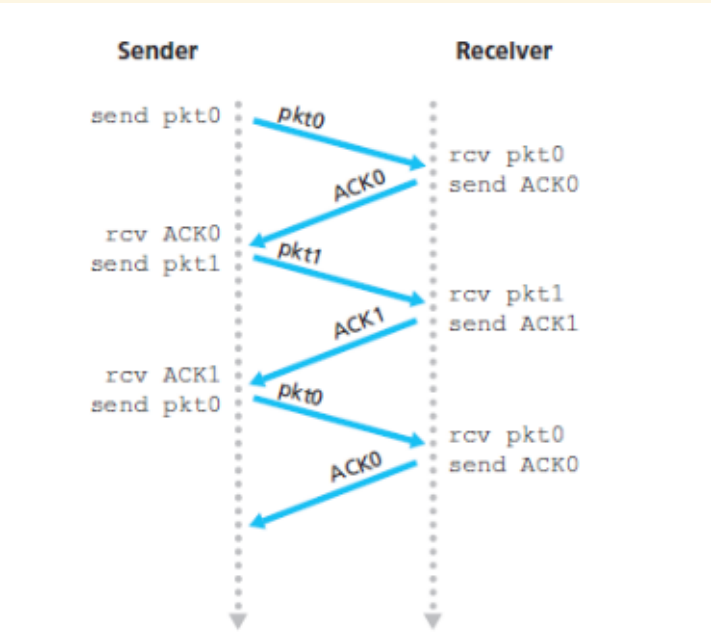
Sender simply waits until it believes packet was likely lost, then retransmits

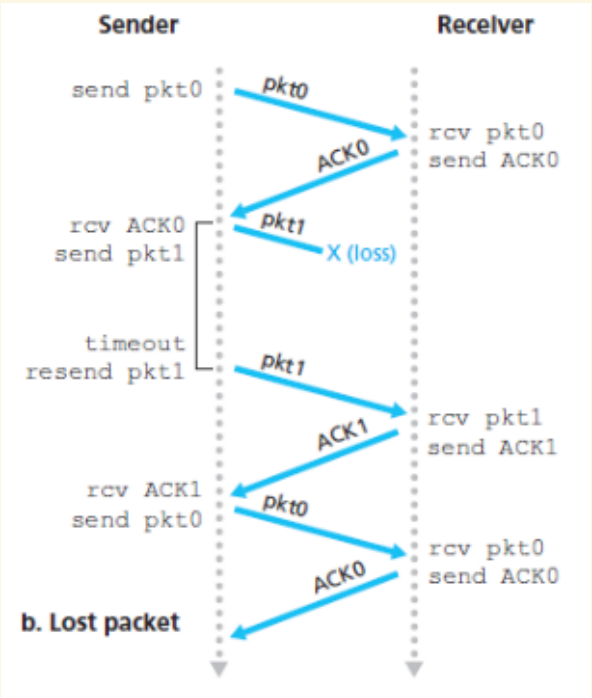
This may put duplicate packet into channel, but previous protocol already showed how to deal with that

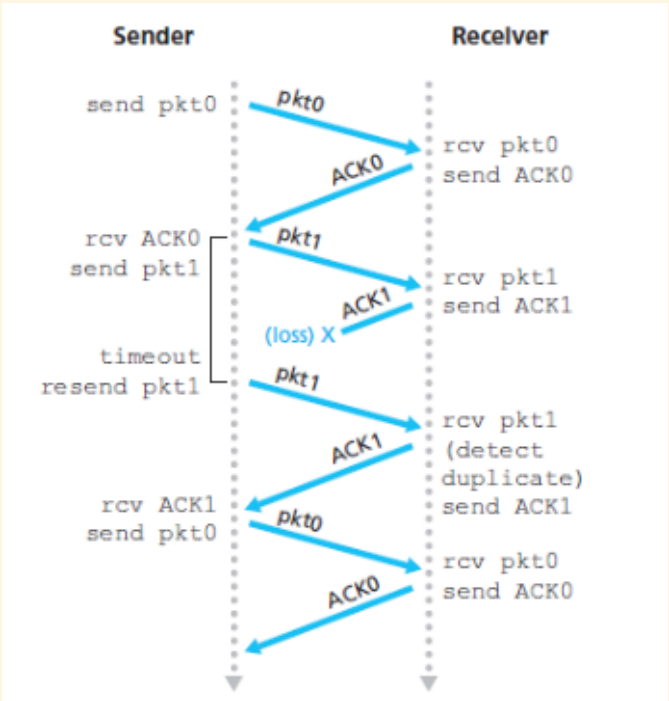
Requires sender to start countdown timer each time packet is sent

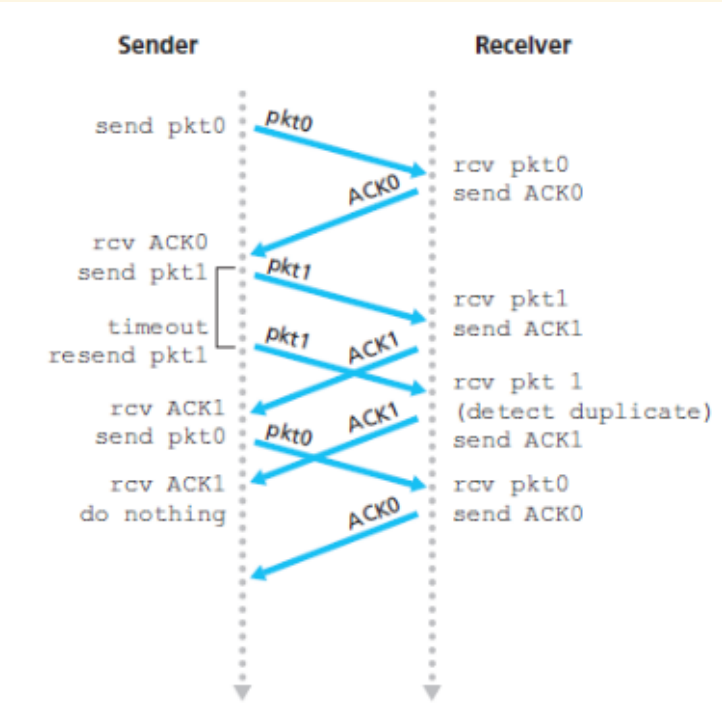
Modified protocol referred to as rdt3.0


rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for
call 0 from
above
Wait for
ACK 0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for
ACK 1
Wait for
call 1 from
above
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)
start_timer


Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0


Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
X (loss)
timeout
resend pkt1
pkt1
rcv pkt1
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
b. Lost packet


Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
ACK1
(loss) X
timeout
resend pkt1
pkt1
rcv pkt1
(detect
duplicate)
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0


Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
timeout
resend pkt1
pkt1
ACK1
rcv pkt 1
(detect duplicate)
send ACK1
rcv ACK1
send pkt0
pkt0
ACK1
rcv pkt0
send ACK0
rcv ACK1
do nothing
ACK0