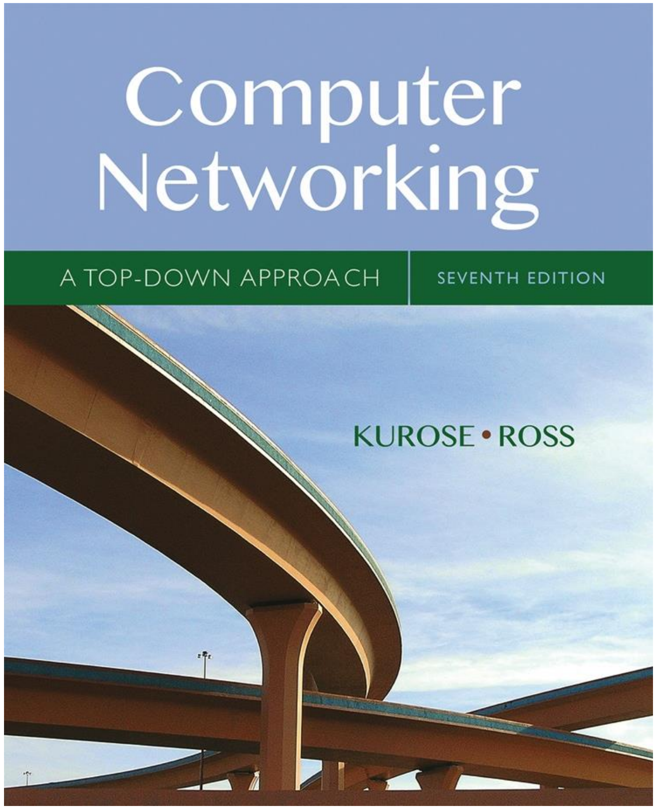
Chapter 3 Transport Layer Part 2



A note on the use of these Powerpoint slides:

We' re making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we' d like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

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

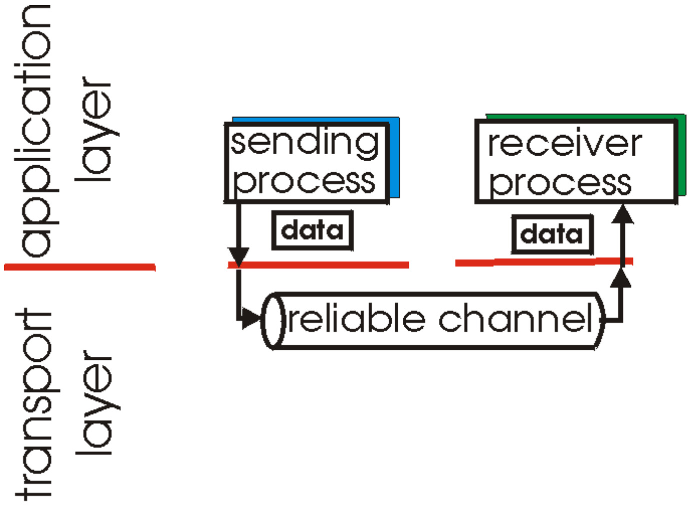
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

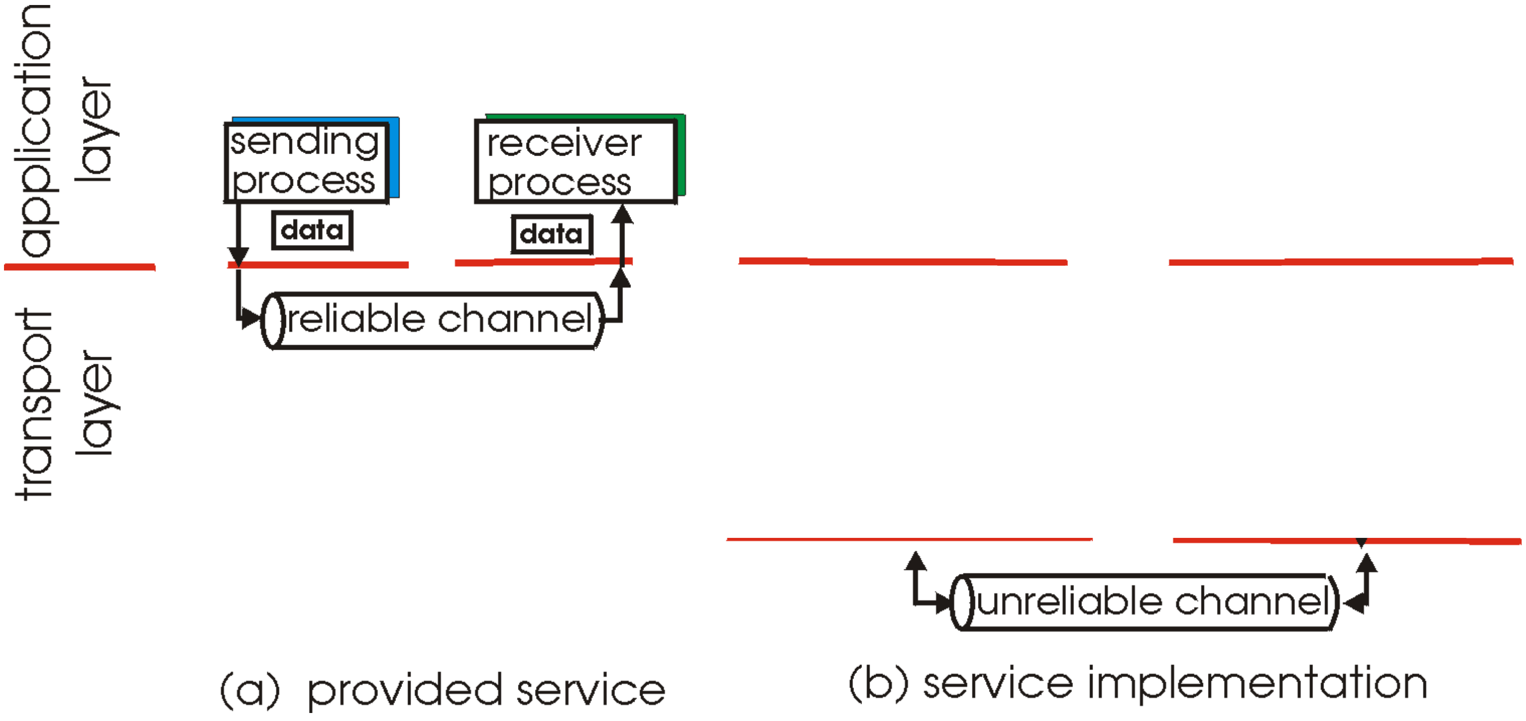


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

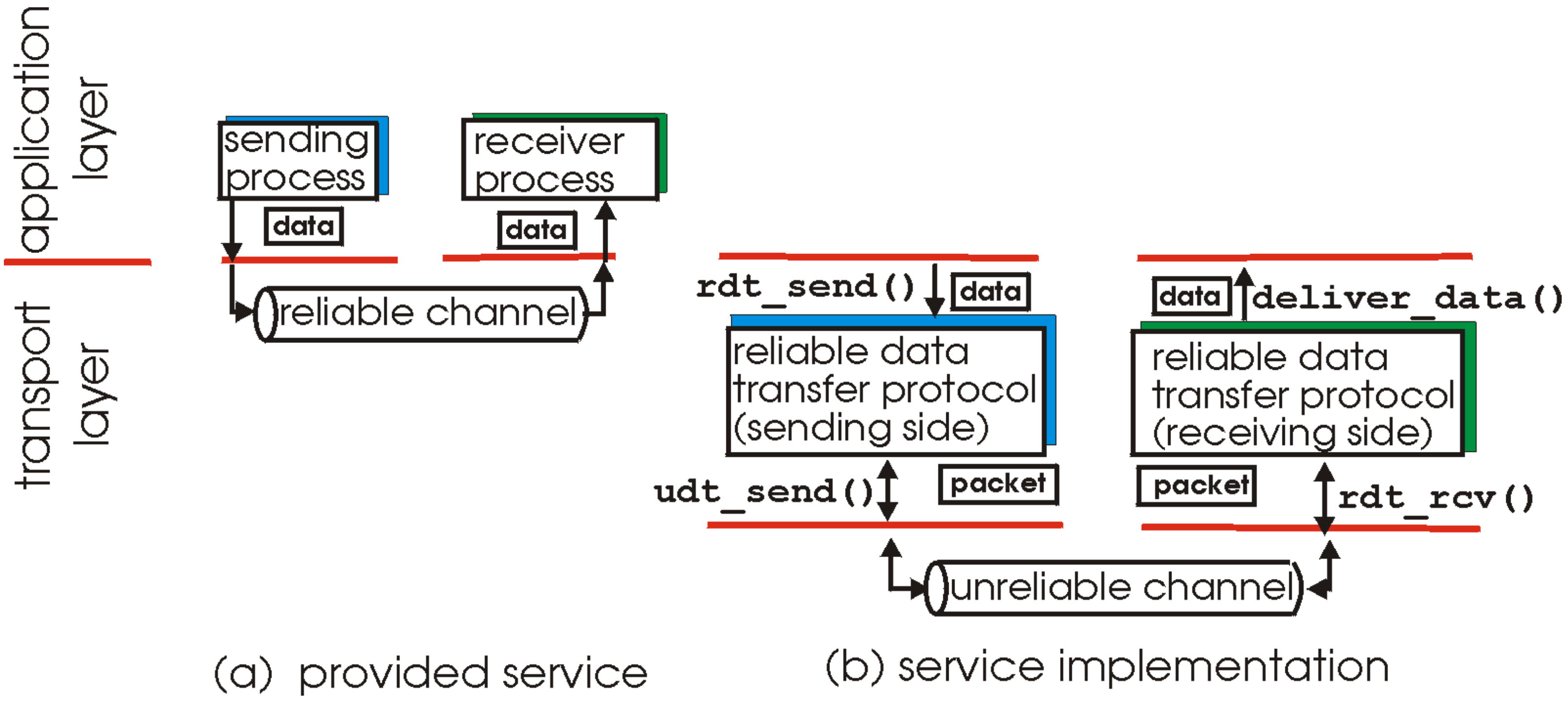
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

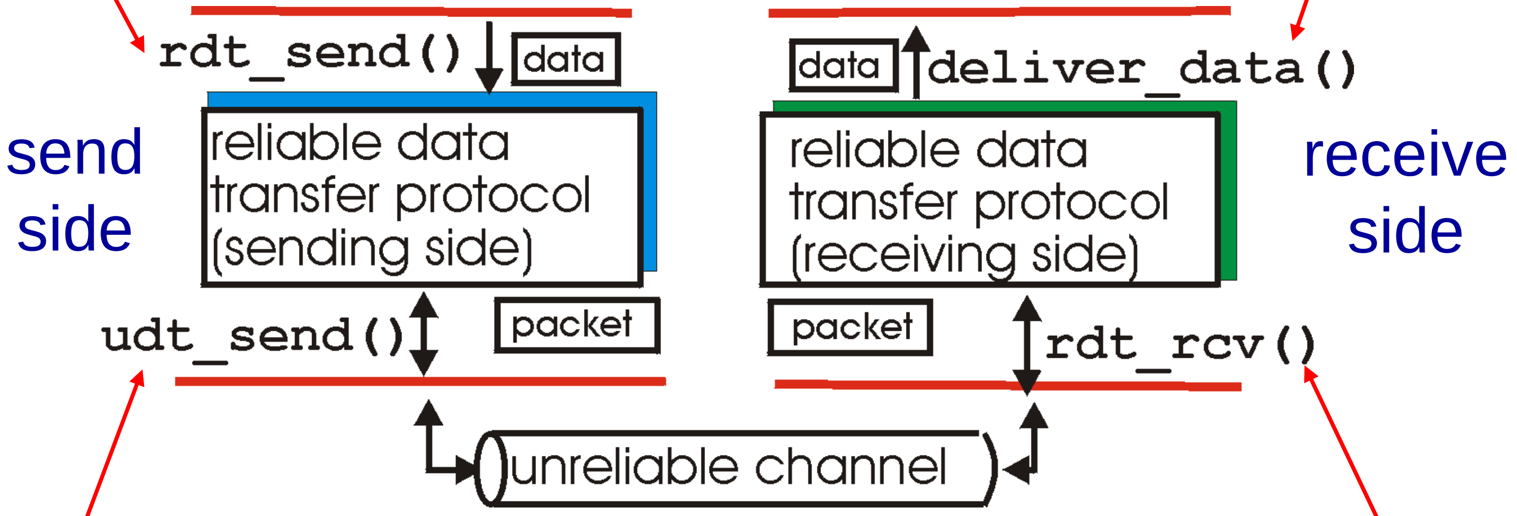


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

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



rdt_send()

data

send side

reliable data transfer protocol (sending side)

udt_send()

packet

data

deliver_data()

reliable data transfer protocol (receiving side)

receive side

packet

rdt_rcv()

unreliable channel

udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

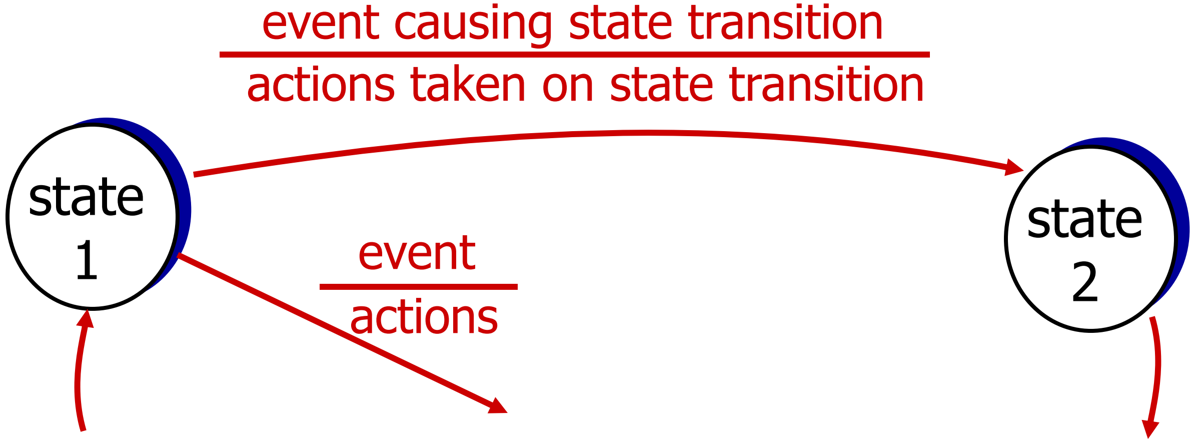
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

we' ll:

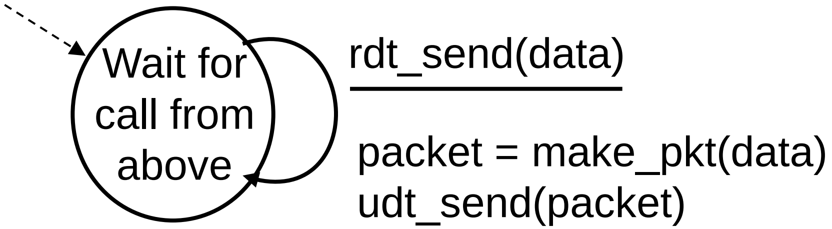
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but ***control info*** will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event


event causing state transition
actions taken on state transition
state 1
event
actions
state 2

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 reads data from underlying channel

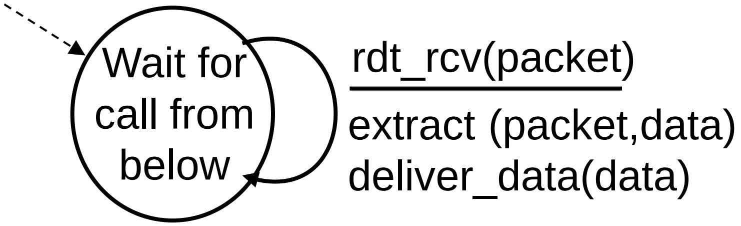


Wait for call from above

rdt_send(data)

packet = make_pkt(data)
udt_send(packet)

sender



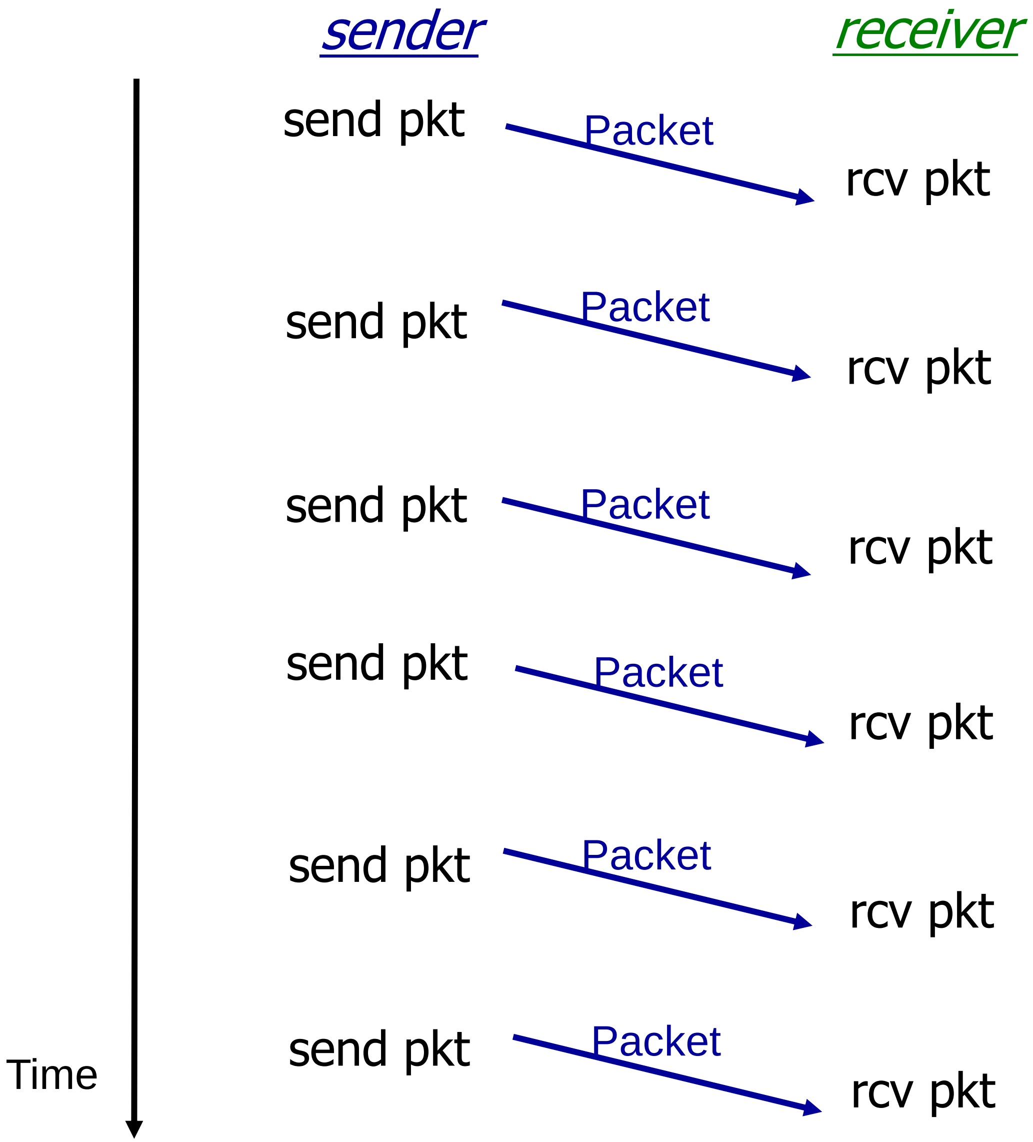
Wait for call from below

rdt_rcv(packet)

extract (packet,data)
deliver_data(data)

receiver

rdt1.0: reliable transfer over a reliable channel



sender

receiver

send pkt Packet rcv pkt

send pkt Packet rcv pkt

send pkt Packet rcv pkt

send pkt Packet rcv pkt

send pkt Packet rcv pkt

send pkt Packet rcv pkt

Time

rdt2.0: channel with bit errors

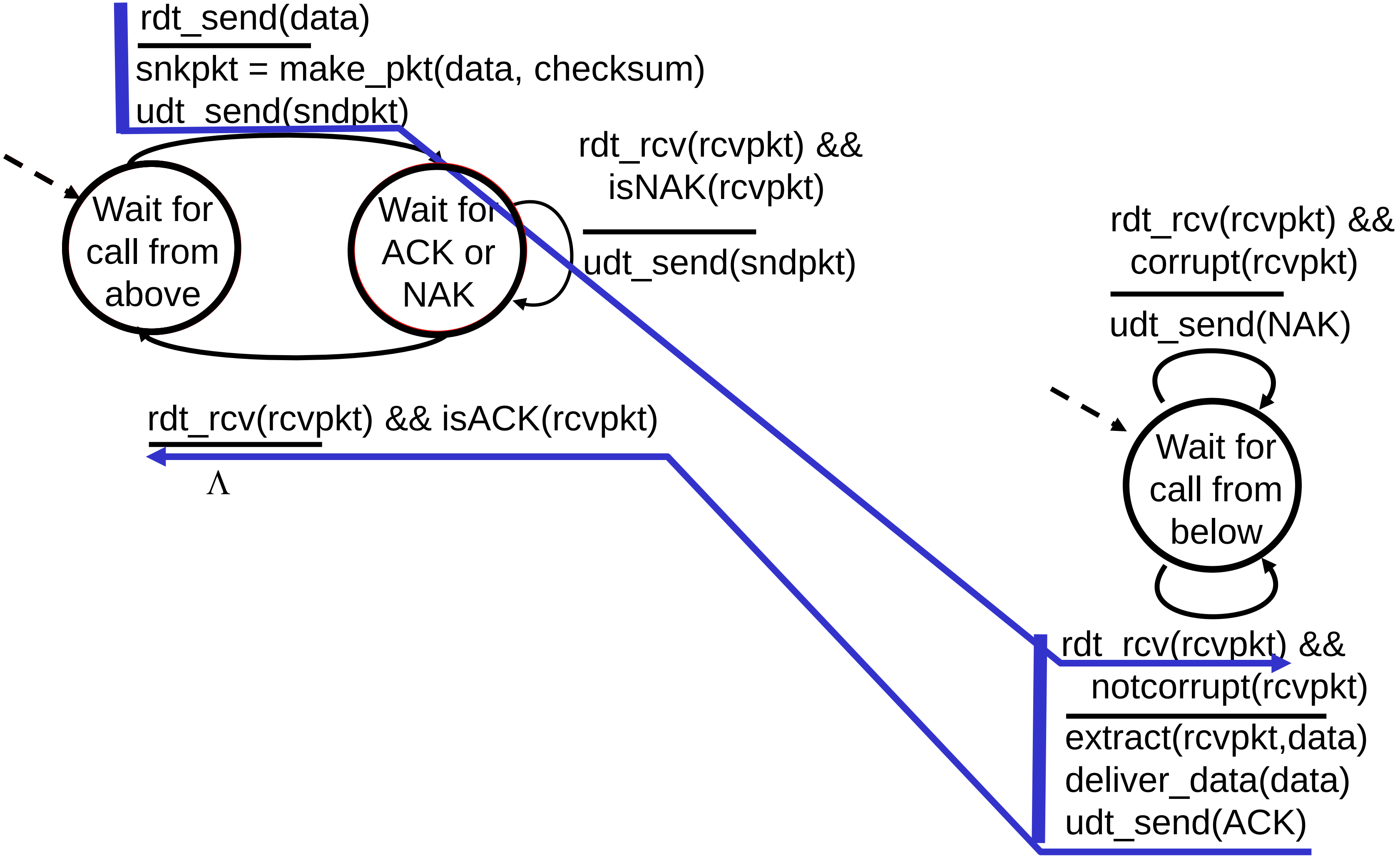
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - 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
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

rdt2.0: operation with no errors in data



rdt_send(data)

snkpkt = 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) && corrupt(rcvpkt)

udt_send(NAK)

Wait for call from below

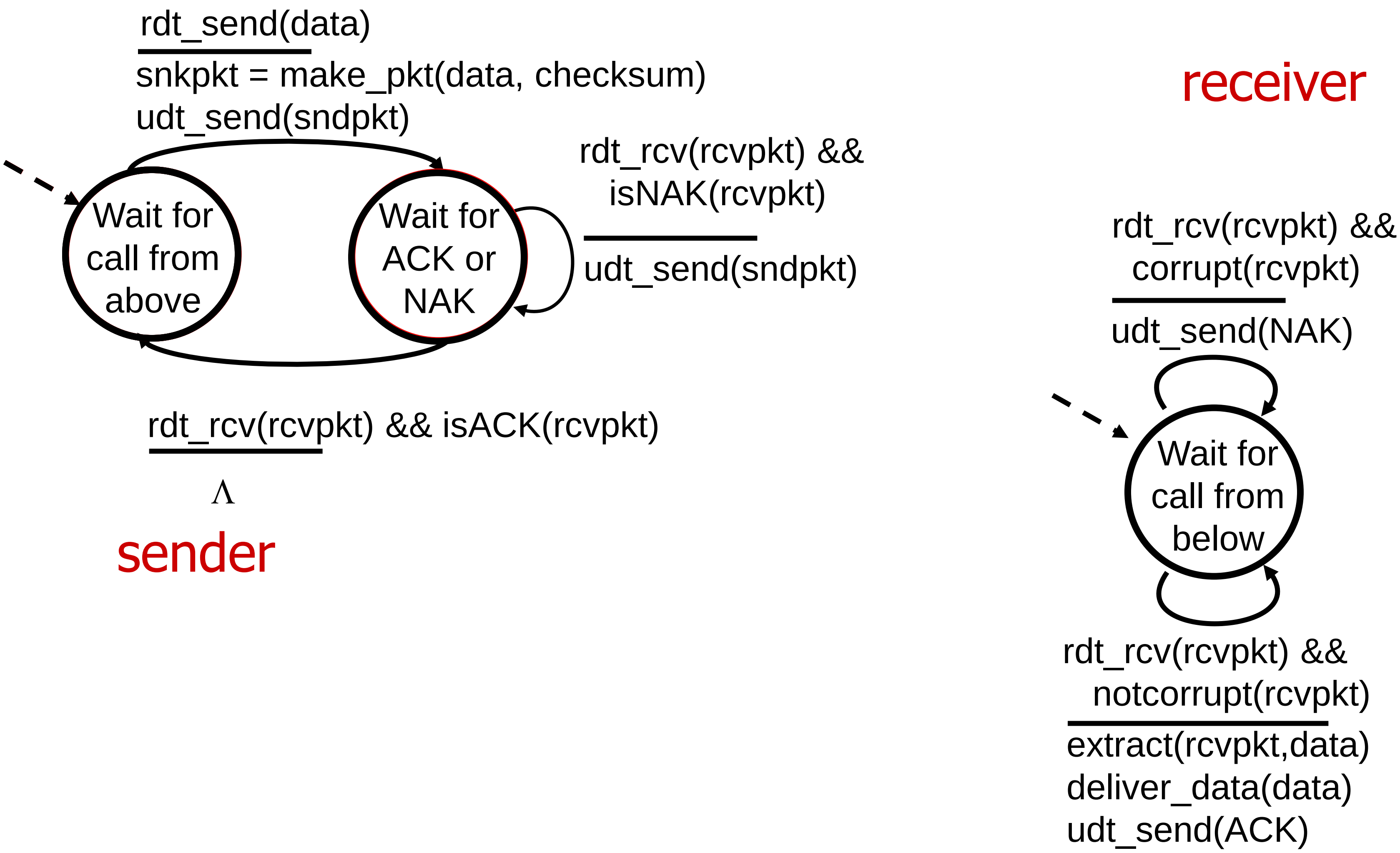
rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: error in data and no error scenarios



rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

receiver

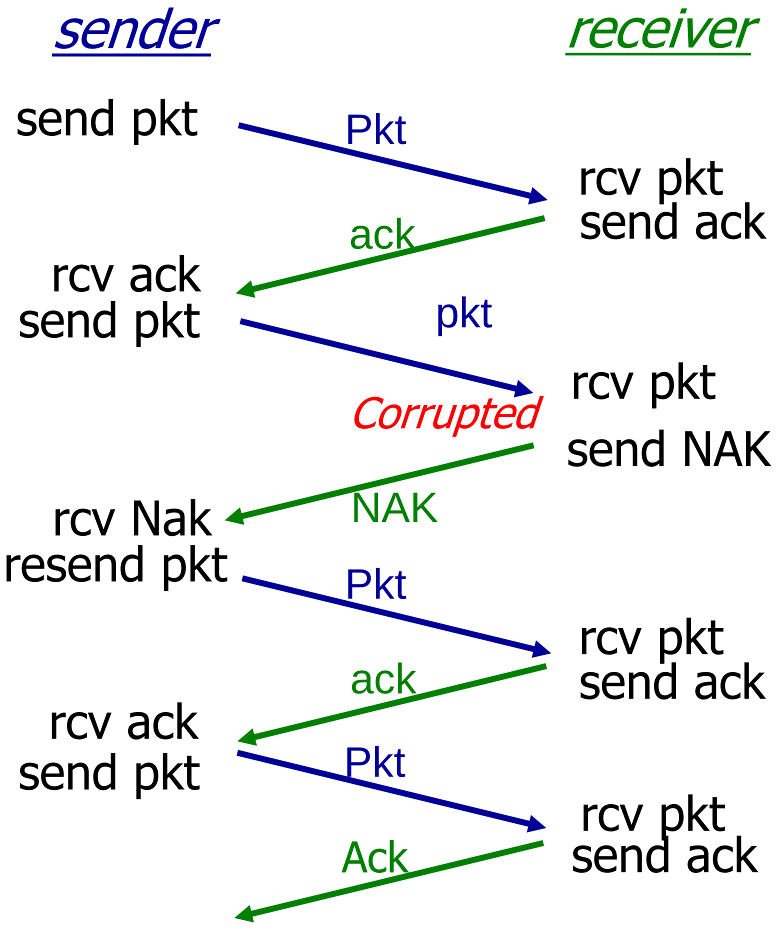
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0 in action (Assuming ACK/NAK Not Corrupted)



Only Data Packet corrupted

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

Solution for handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each data pkt
- receiver discards (doesn't deliver up) duplicate pkt

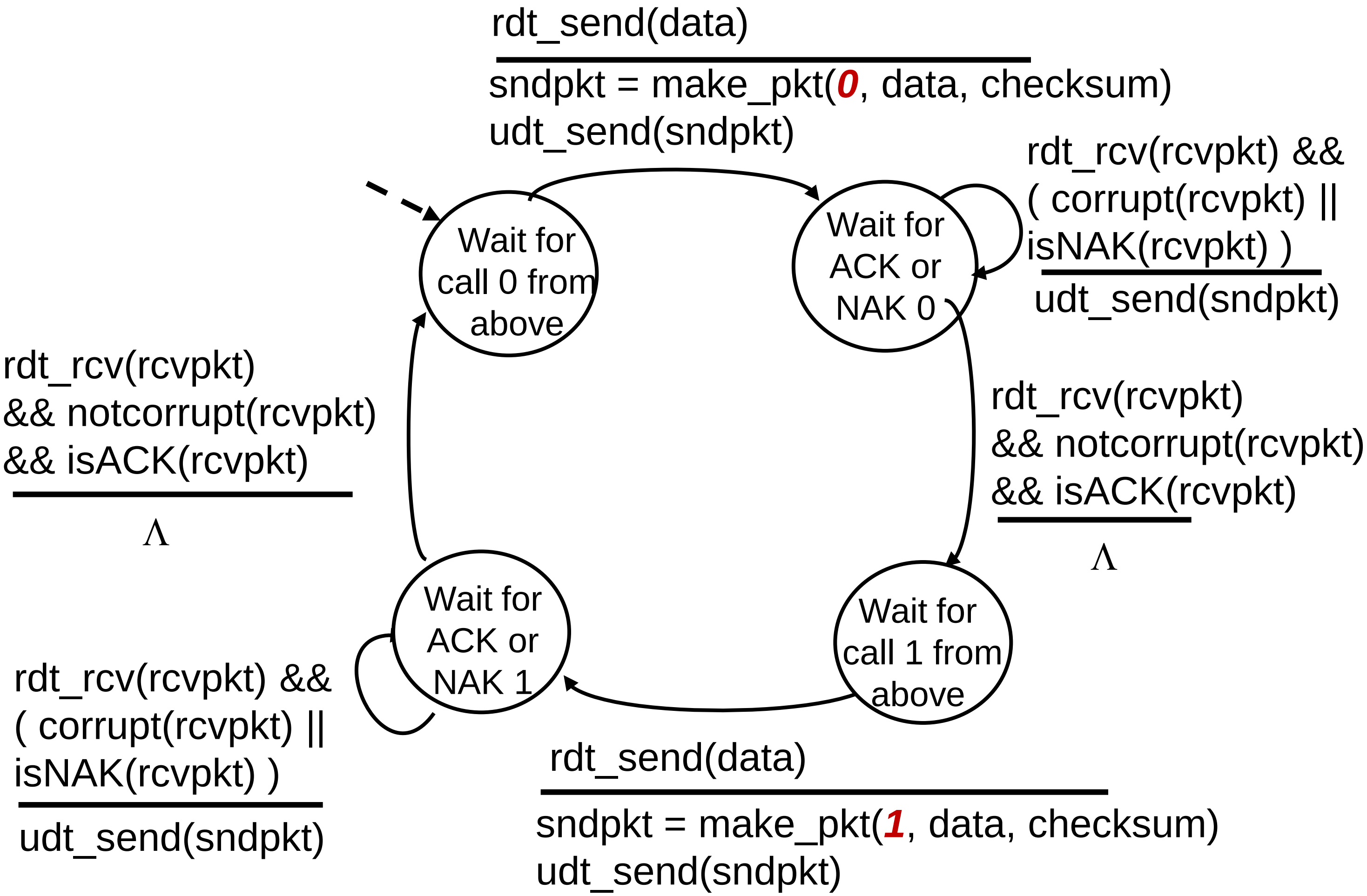
Problem: duplicate

Receiver doesn't know whether received pkt is a retransmit or a new pkt

stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_send(data)

sndpkt = make_pkt(***0***, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt_send(data)

sndpkt = make_pkt(***1***, data, checksum)
udt_send(sndpkt)

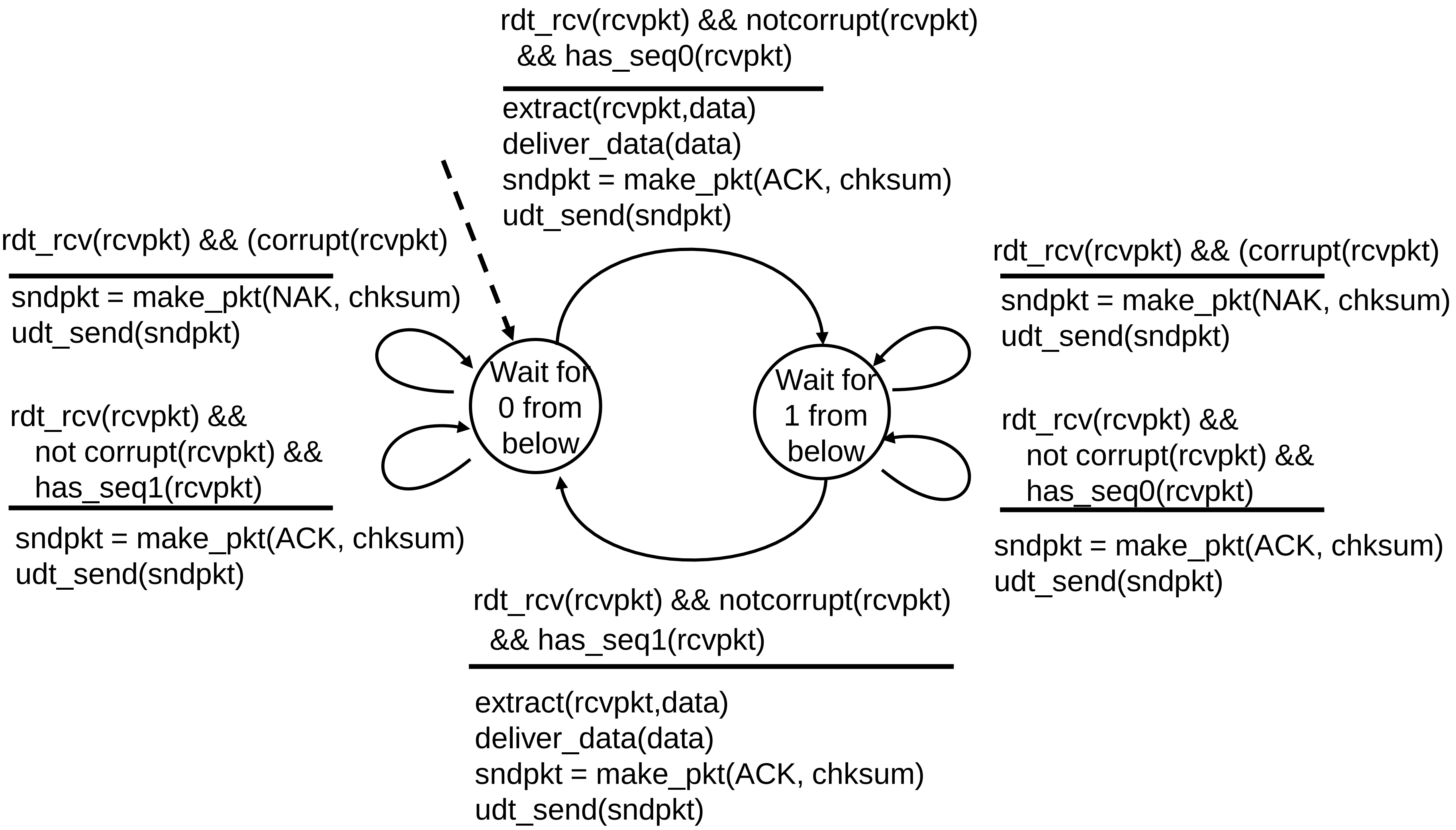
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt2.1: receiver, handles garbled ACK/NAKs



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Wait for 0 from below

Wait for 1 from below

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

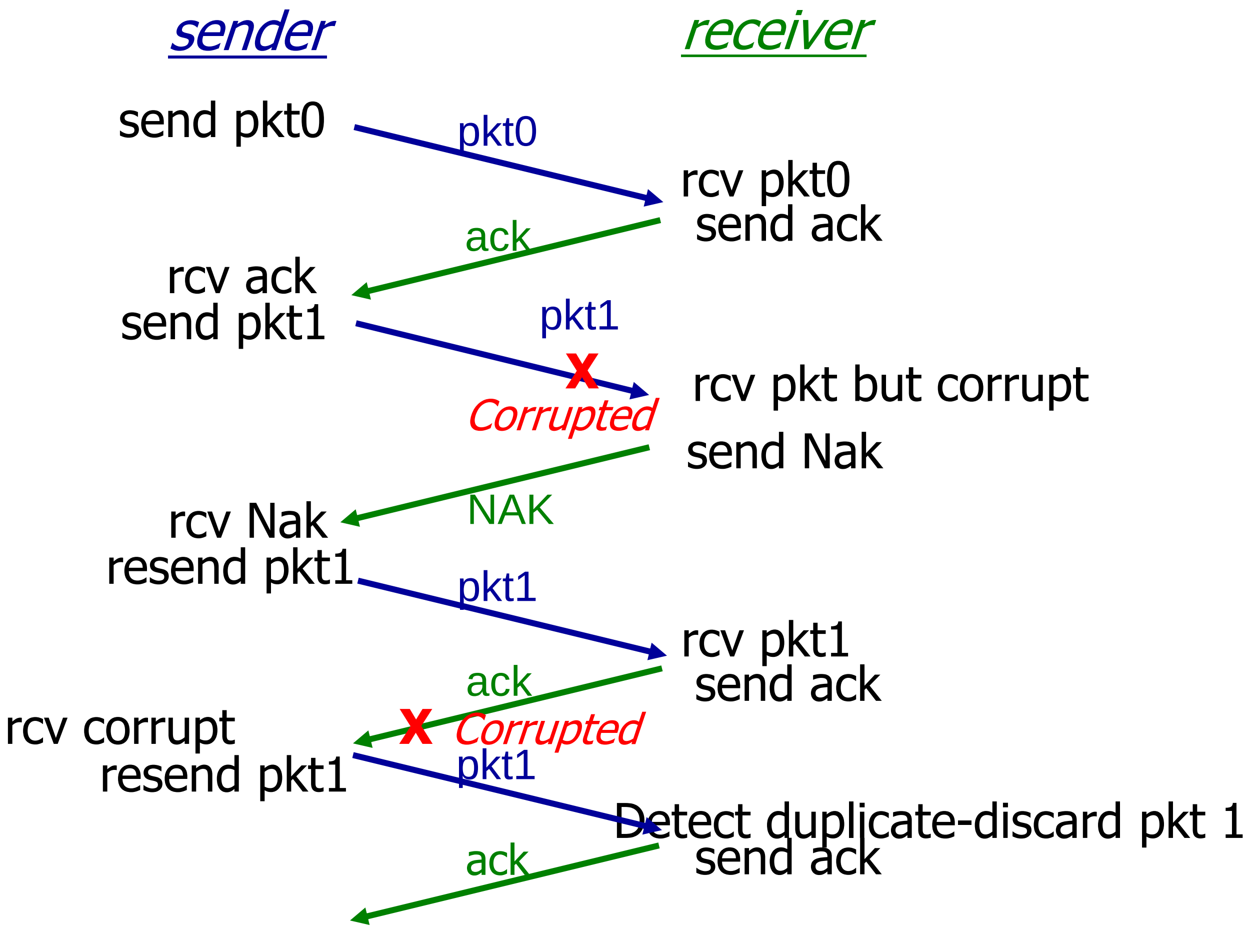
rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1 (Each DataPacket has a sequence number)



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack

ack

rcv ack
send pkt1

pkt1

X
Corrupted

rcv pkt but corrupt

send Nak

NAK

rcv Nak
resend pkt1

pkt1

rcv pkt1
send ack

ack

X *Corrupted*

rcv corrupt
resend pkt1

pkt1

Detect duplicate-discard pkt 1
send ack

ack

Packet and ack/nak corruption

rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #' s (0,1) will suffice.
- must check if received ACK/NAK corrupted
- twice as many states

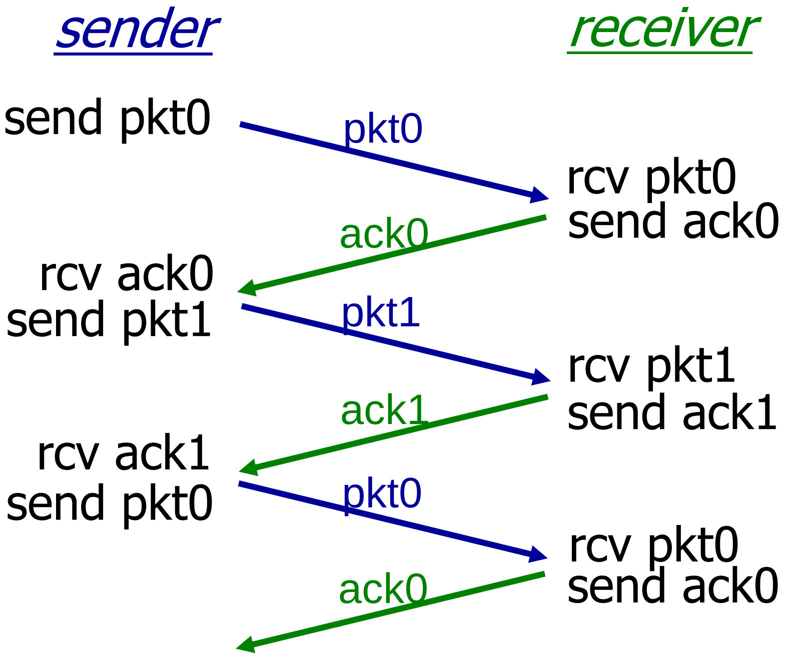
receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

rdt2.2: a NAK-free protocol

- No changes in channel characteristics:
  - Channel introduces errors to the data and control packets
- 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*
- NAK-free protocol puts foundation for more advanced 'reliable data transfer protocols' in the upcoming slides.

rdt2.2 in action (Add seq# to ACK/NAK too)



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

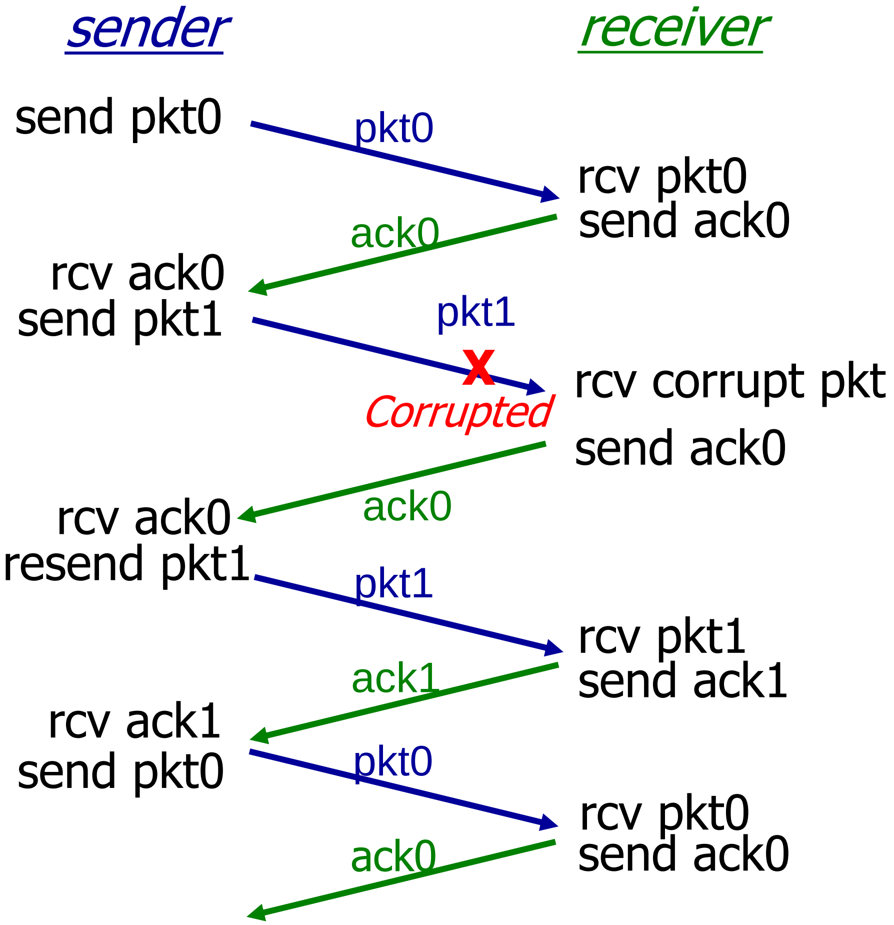
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(a) no corruption



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

x

Corrupted

rcv corrupt pkt

send ack0

ack0

rcv ack0
resend pkt1

pkt1

rcv pkt1
send ack1

ack1

rcv ack1
send pkt0

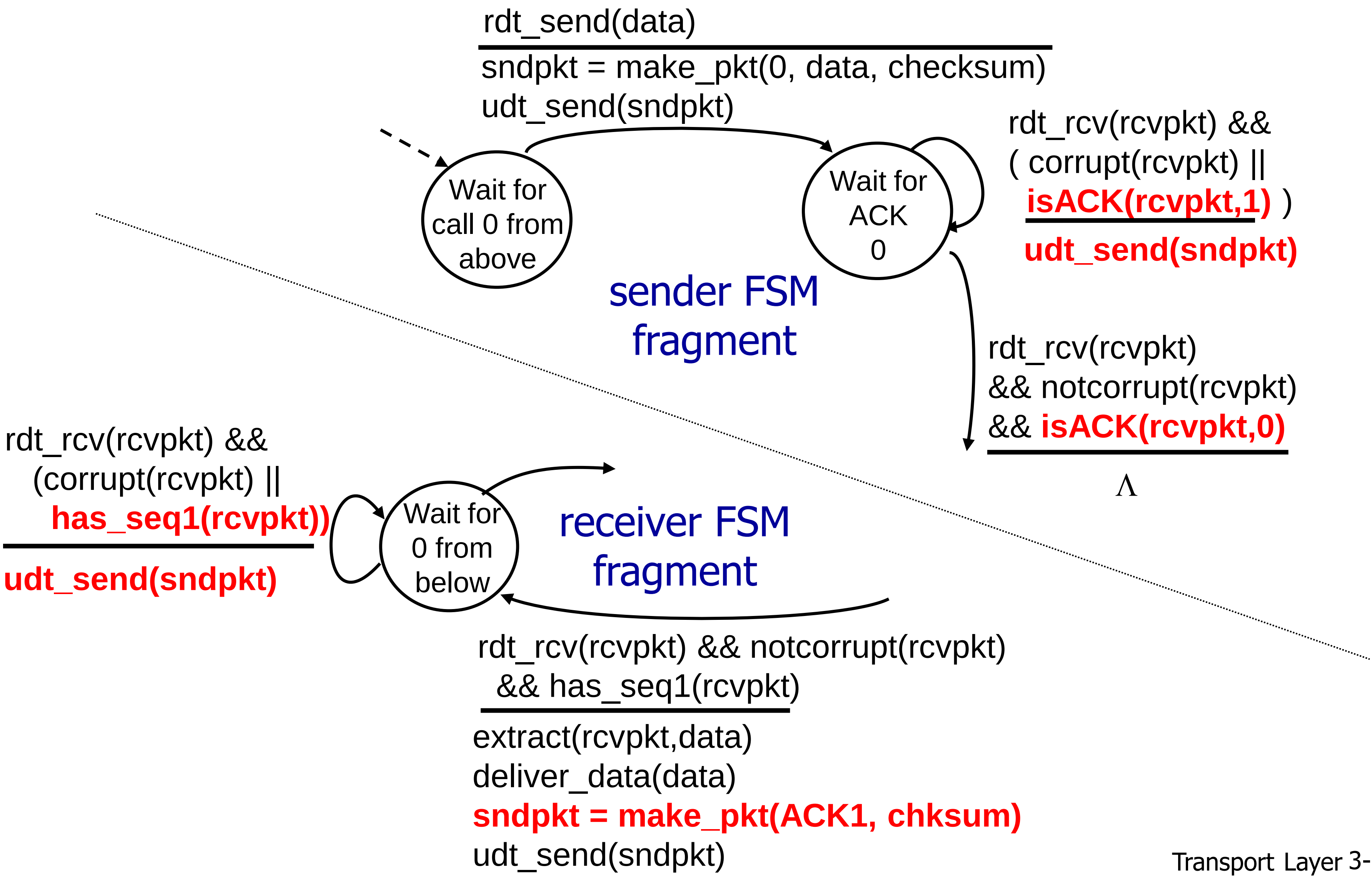
pkt0

rcv pkt0
send ack0

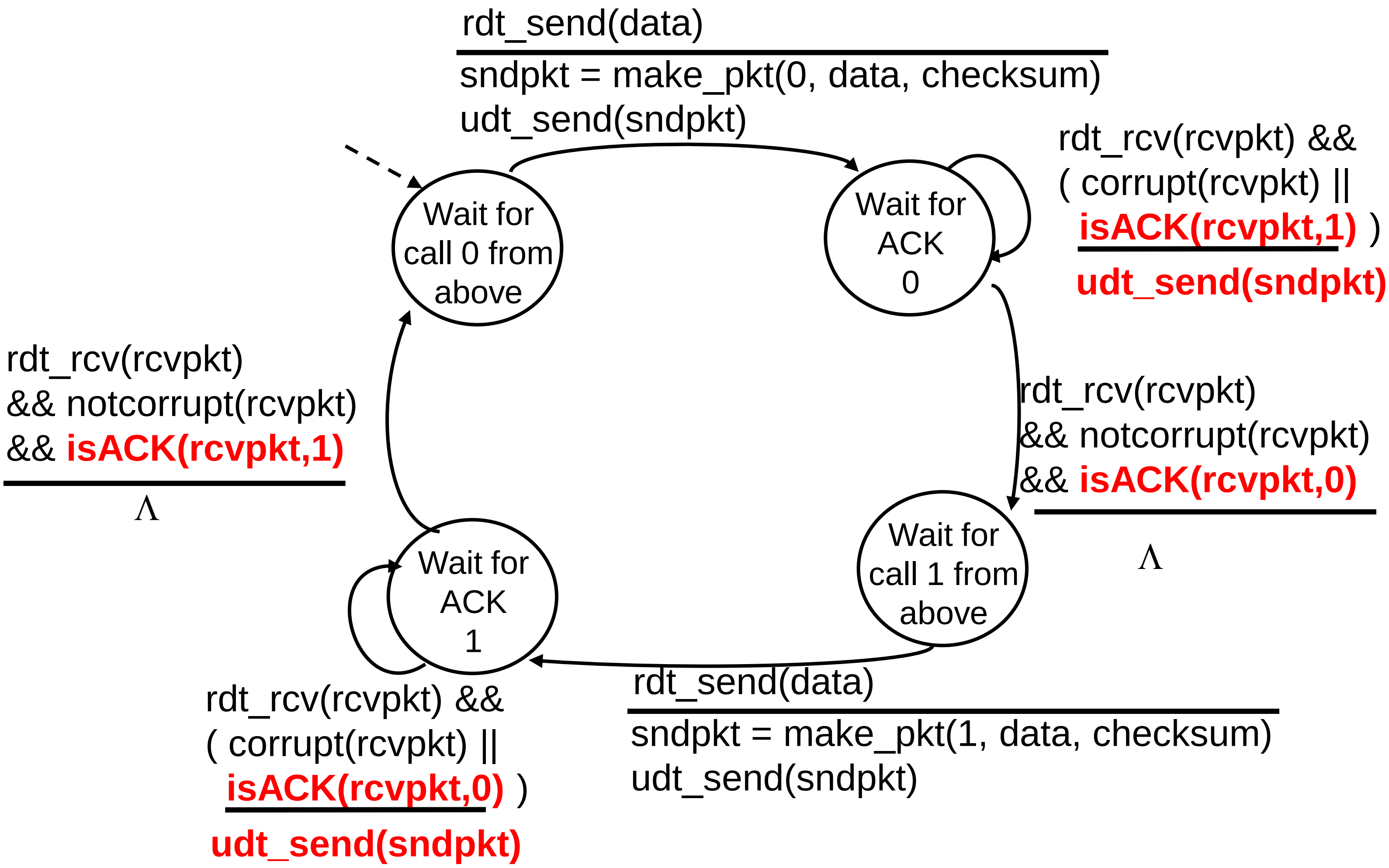
ack0

(b) Packet corruption

rdt2.2: sender, receiver fragments



rdt2.2: sender (full FSM)



Wait for call 0 from above

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for ACK 0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,0)**
Λ

Wait for call 1 from above

rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

Wait for ACK 1

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,1)**
Λ

rdt3.0: channels with errors *and* loss

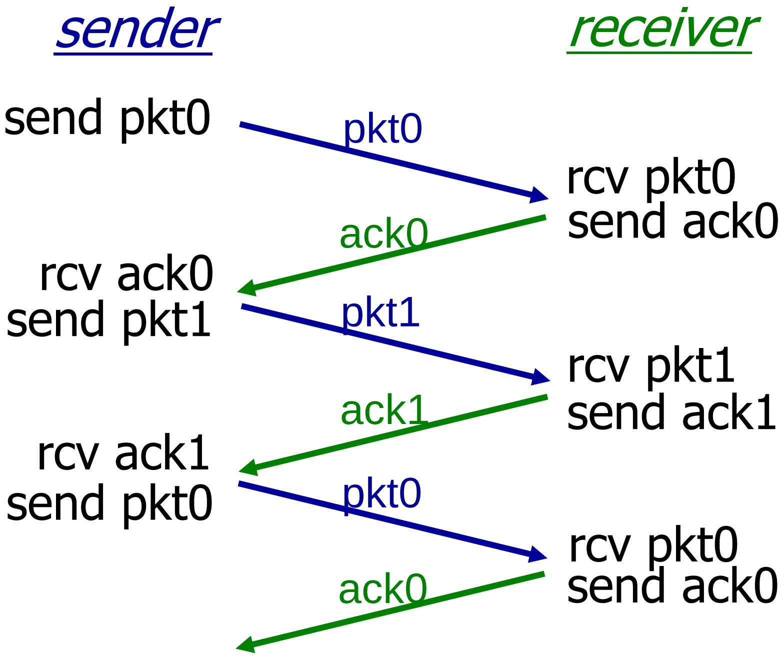
new assumption (or maybe relaxing one assumption of channel not being lossy anymore): underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

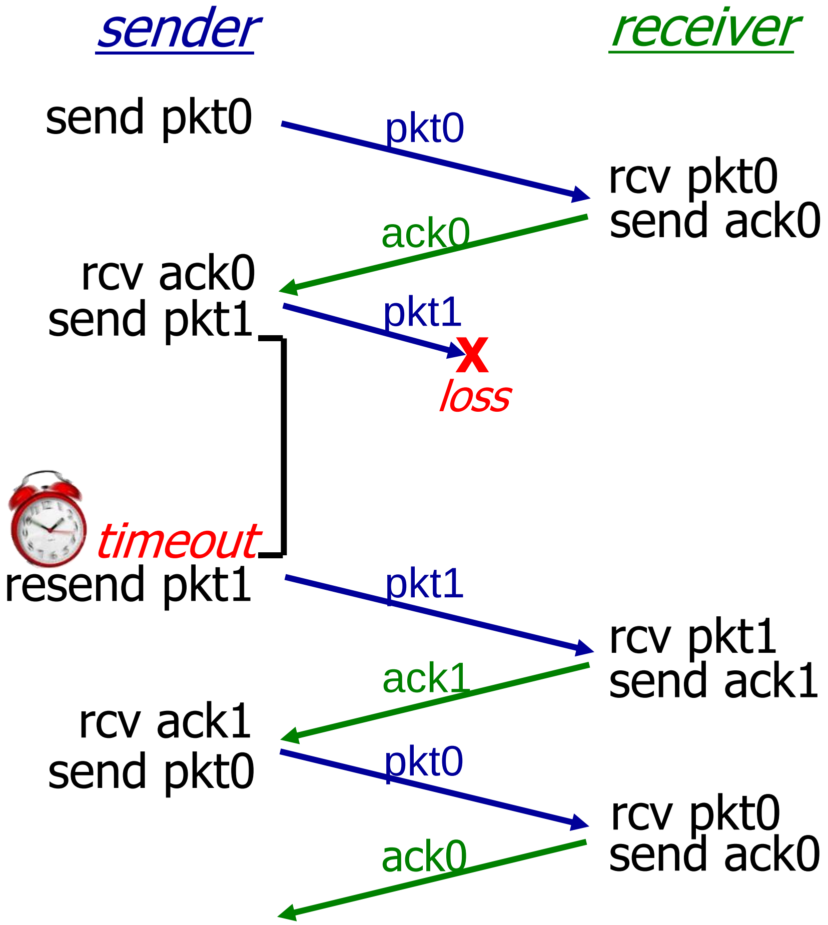
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 (no loss vs packet loss)

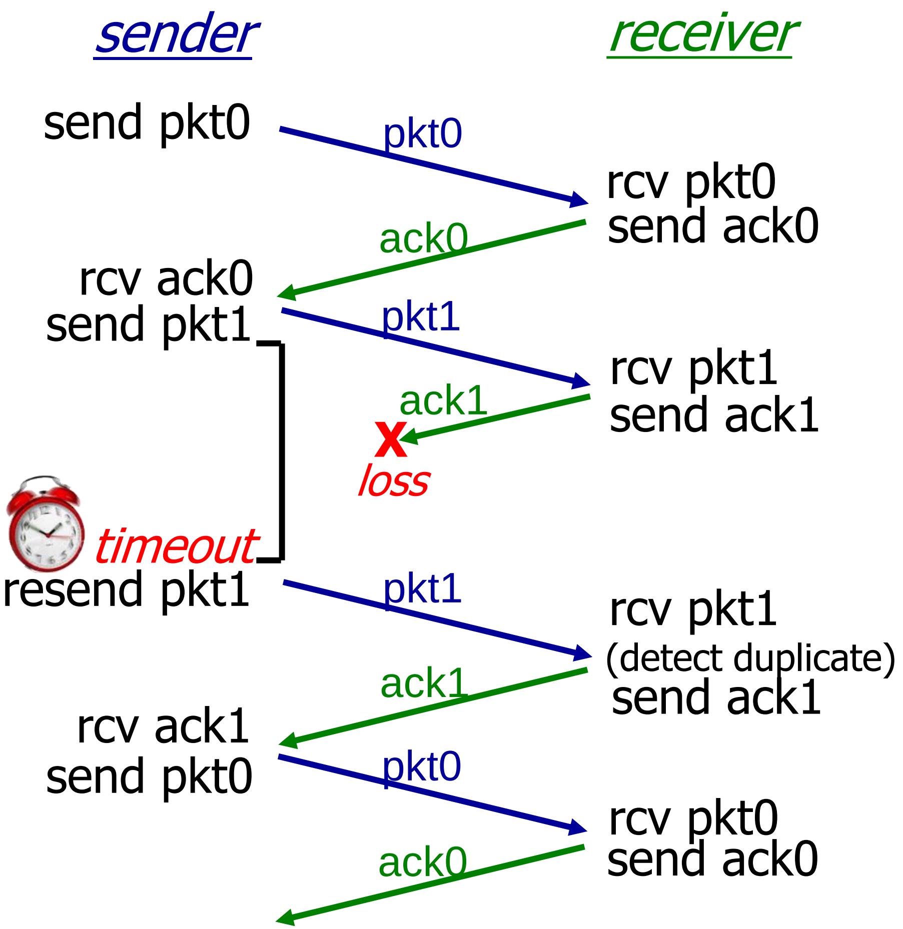


(a) no loss

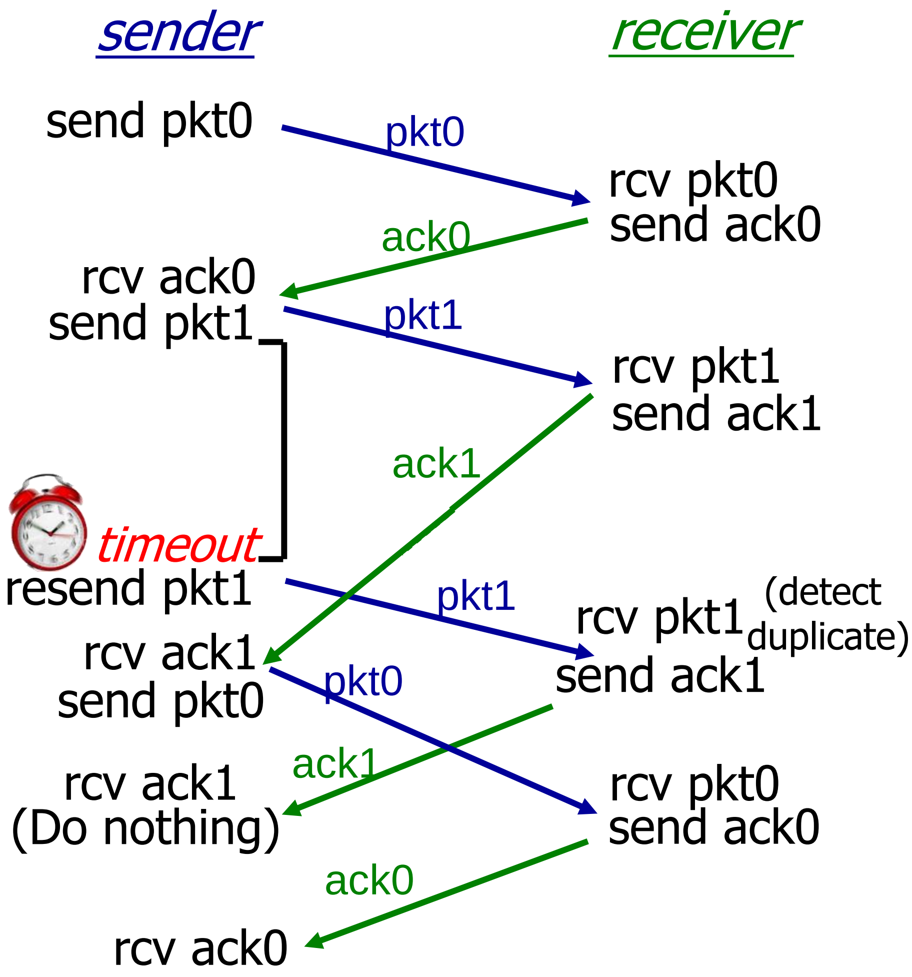


(b) packet loss

rdt3.0 (ACK loss vs ACK delayed)

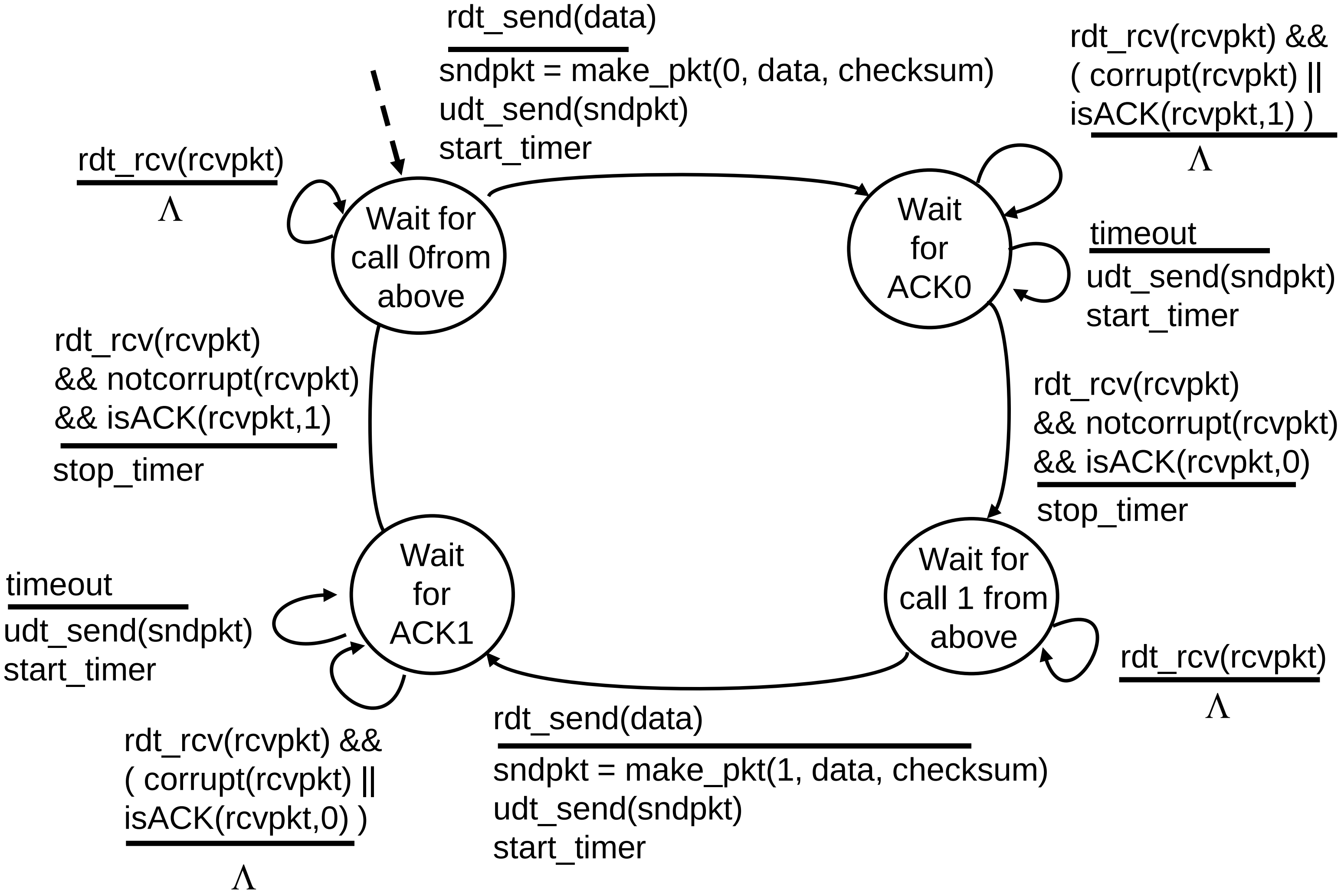


(c) ACK loss



(d) premature timeout/ delayed ACK

rdt3.0 sender



rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)

Λ

Wait for call 0from above

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

Λ

Wait for ACK0

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 ACK1

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

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}=0.008\text{msec}$$

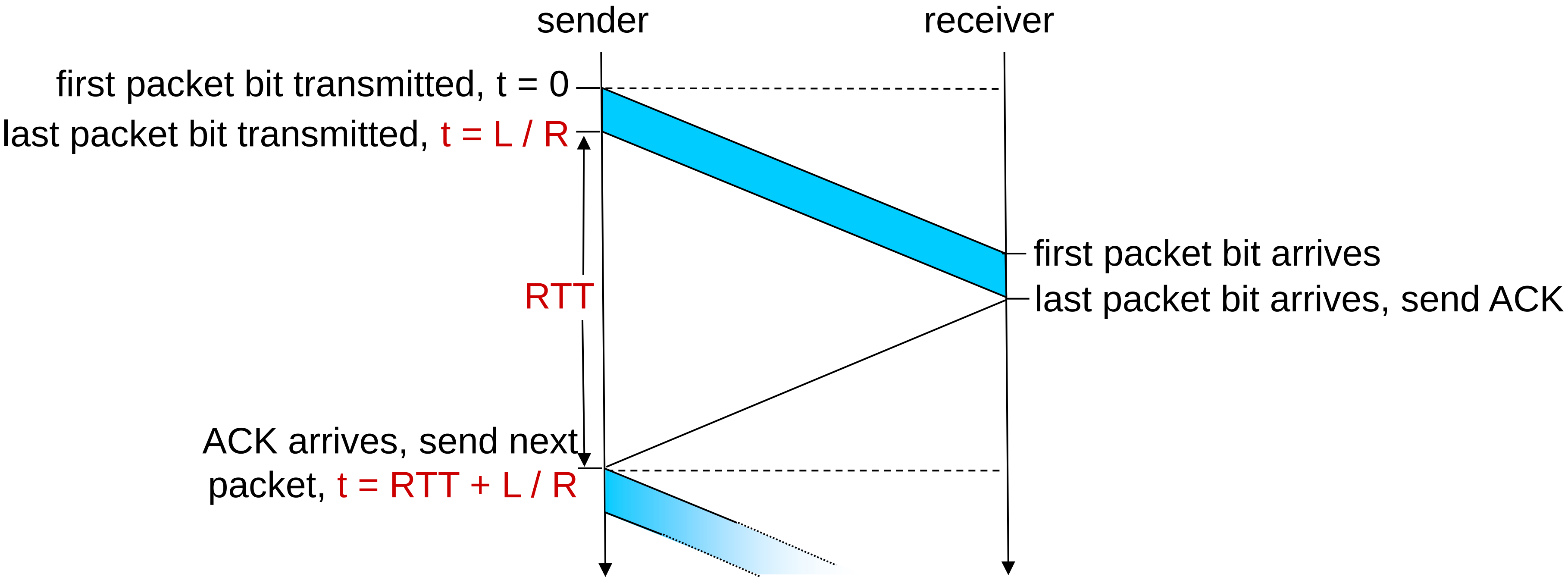
- Performance can be found using (1) utilization and (2) Throughput
  - U_{sender}: *utilization* is fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30+0.008} = 0.00027$$

- if RTT=30 msec, and 1kB pkt every 30 msec (0.03 seconds) is transmitted over 1 Gbps link, then throughput is 33kB/sec

- network protocol (rdt3.0) limits use of physical resources!

rdt3.0: stop-and-wait operation



sender

receiver

first packet bit transmitted, t = 0

last packet bit transmitted, t = L / R

first packet bit arrives

last packet bit arrives, send ACK

RTT

ACK arrives, send next packet, t = RTT + L / R

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