

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

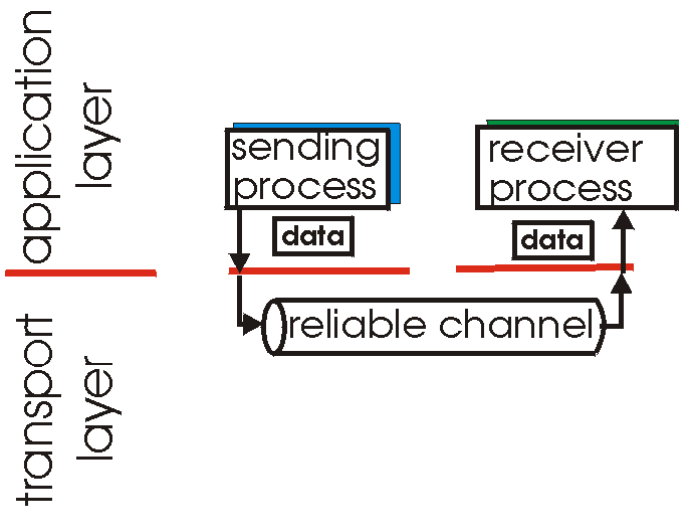
- segment structure
- reliable data transfer
- flow control
- connection management

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!

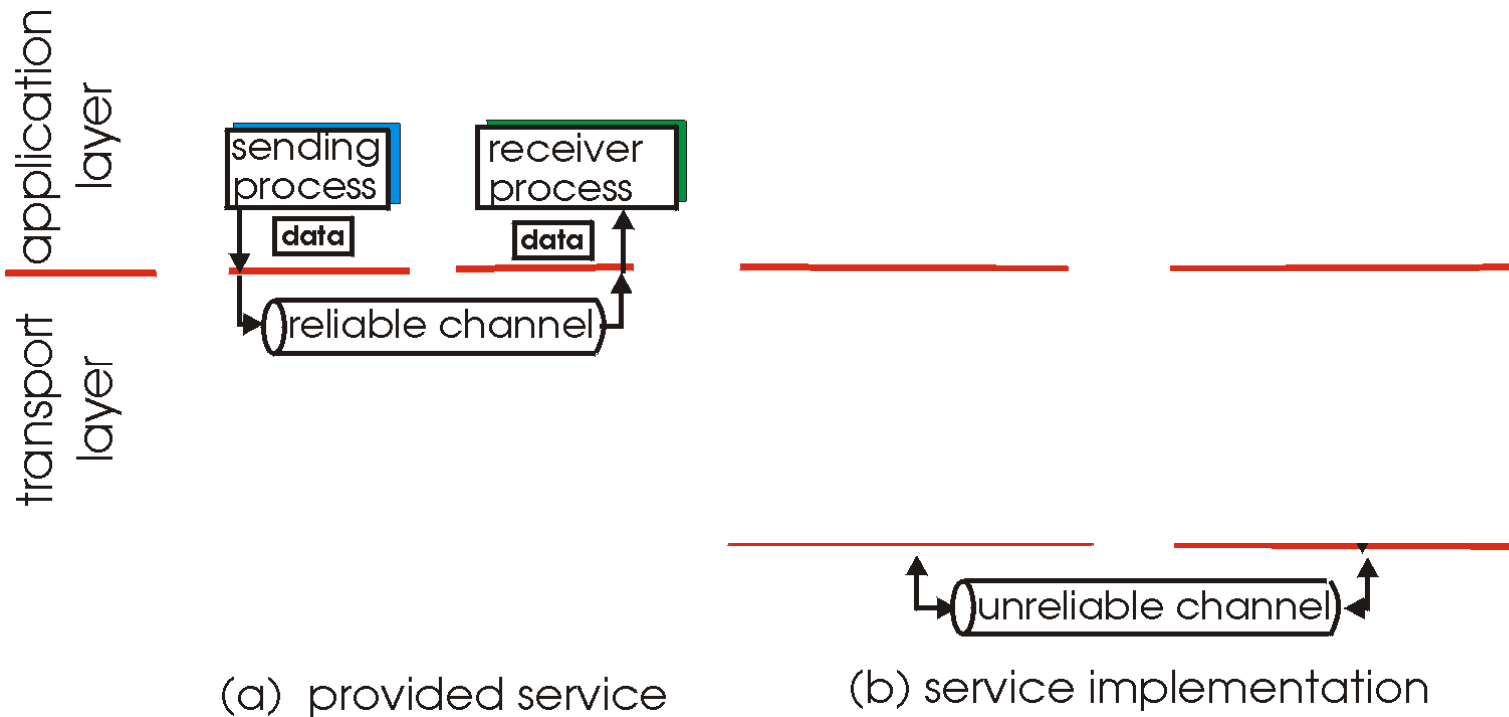


(a) provided service

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

Principles of reliable data transfer

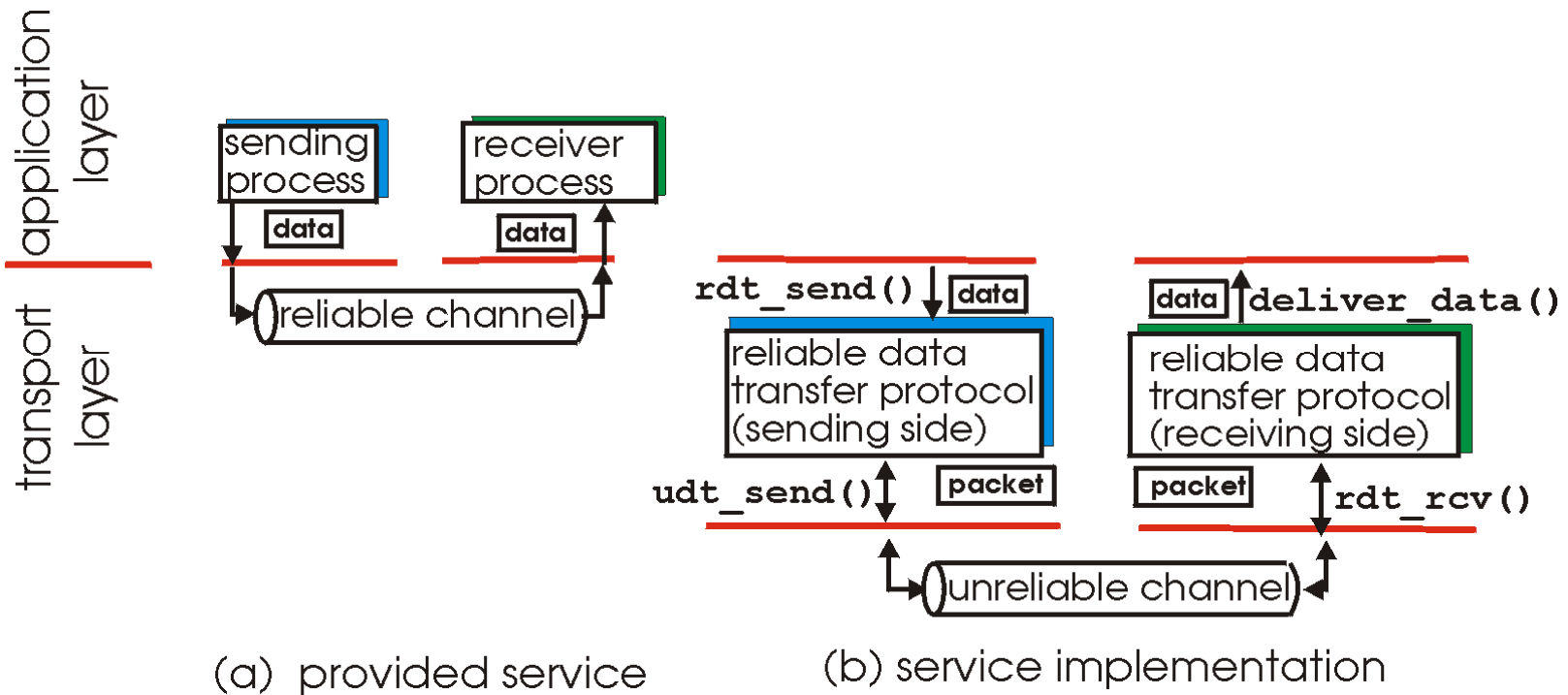
- important in application, transport, link layers
 - top-10 list of important networking topics!



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

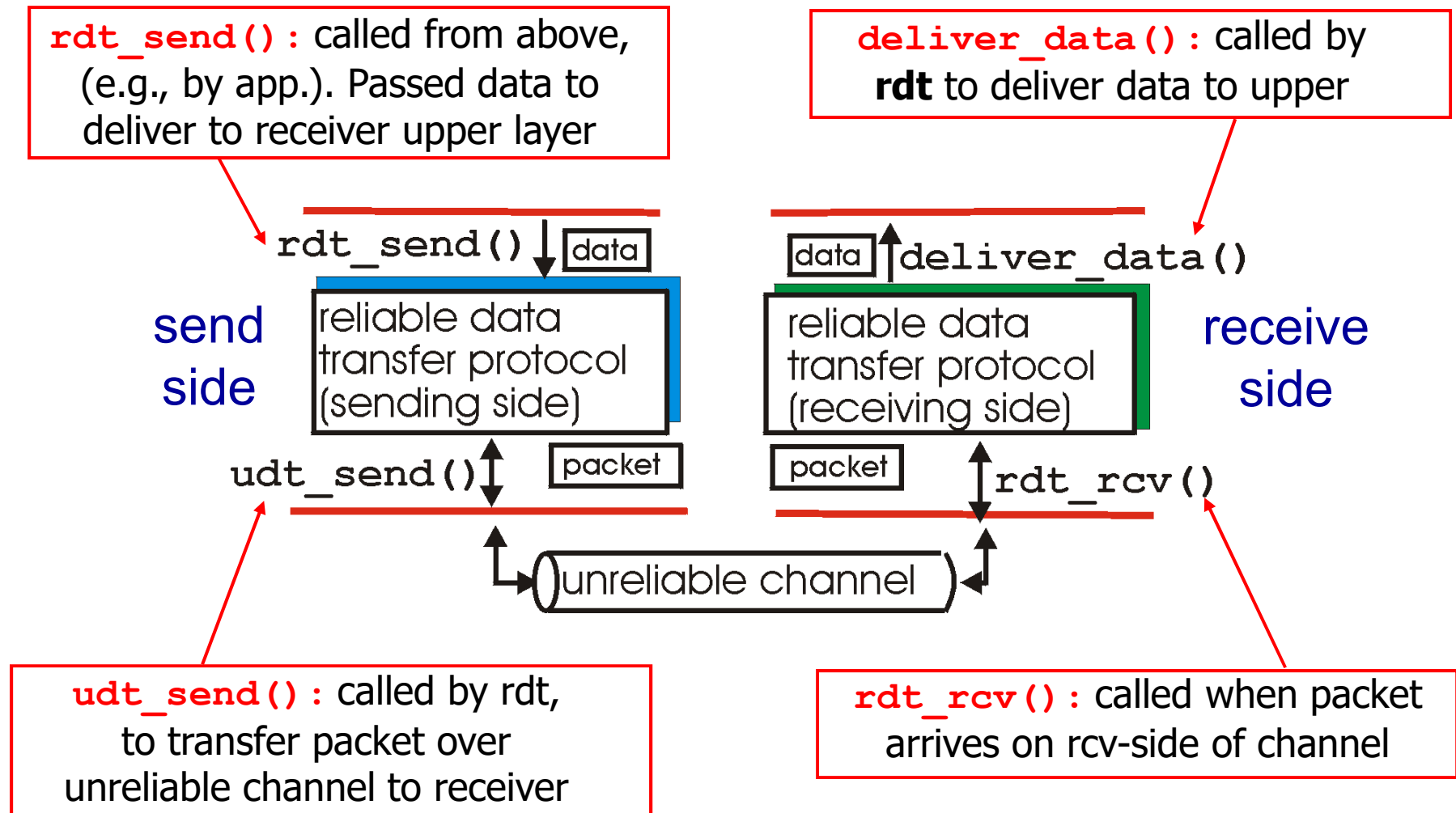
Principles of reliable data transfer

- 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)

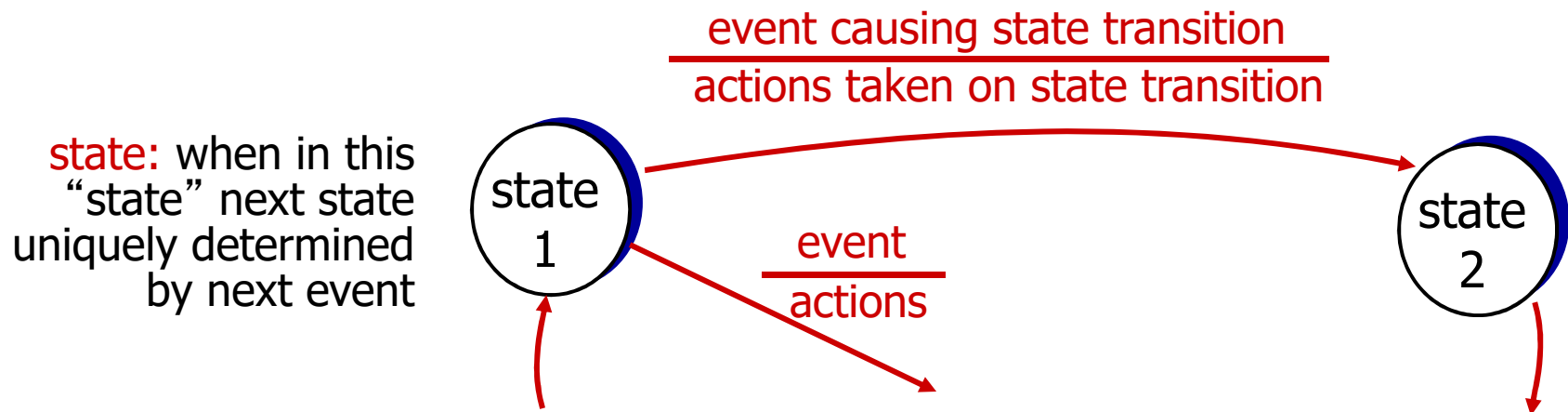
Reliable data transfer: getting started



Reliable data transfer: getting started

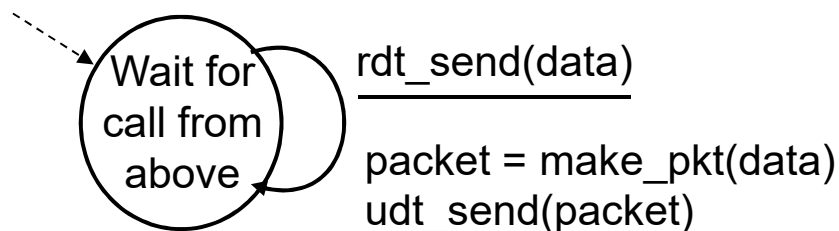
we' ll:

- 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

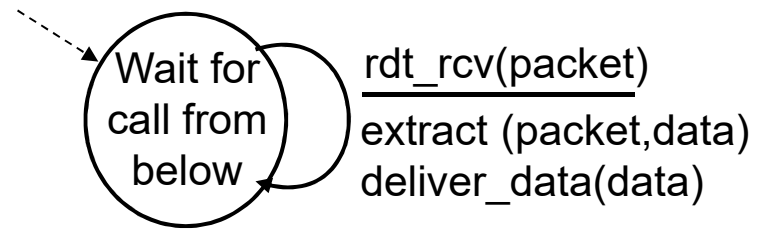


rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
 - no bit errors
 - no loss of packets
- separate FSMs for sender, receiver:
 - sender sends data into underlying channel
 - receiver reads data from underlying channel



sender



receiver

rdt2.0: channel with bit errors

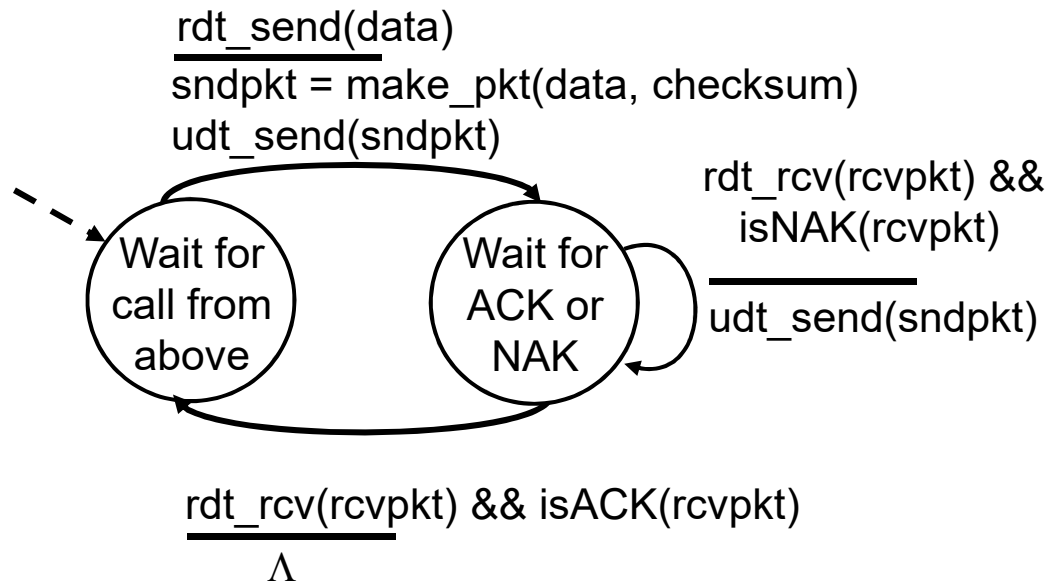
- underlying channel may flip bits in packet
 - checksum to detect bit errors
- *the question*: how to recover from errors:

*How do humans recover from “errors”
during conversation?*

rdt2.0: channel with bit errors

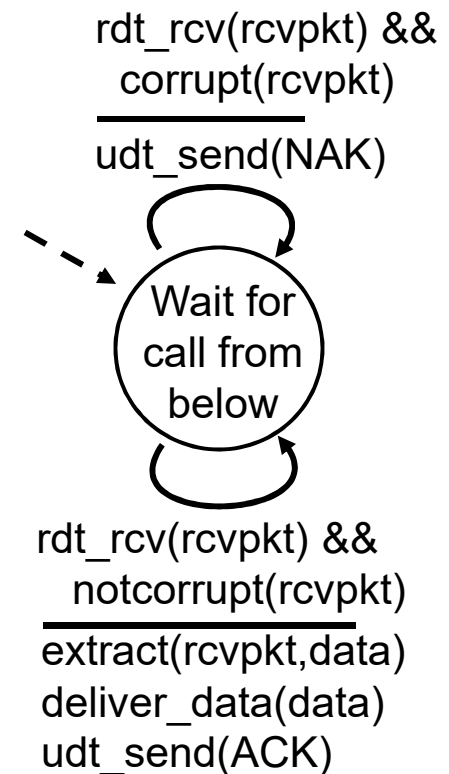
- underlying channel may flip bits in packet
 - checksum to detect bit errors
- *the question: how to recover from errors:*
 - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
 - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
 - 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: FSM specification

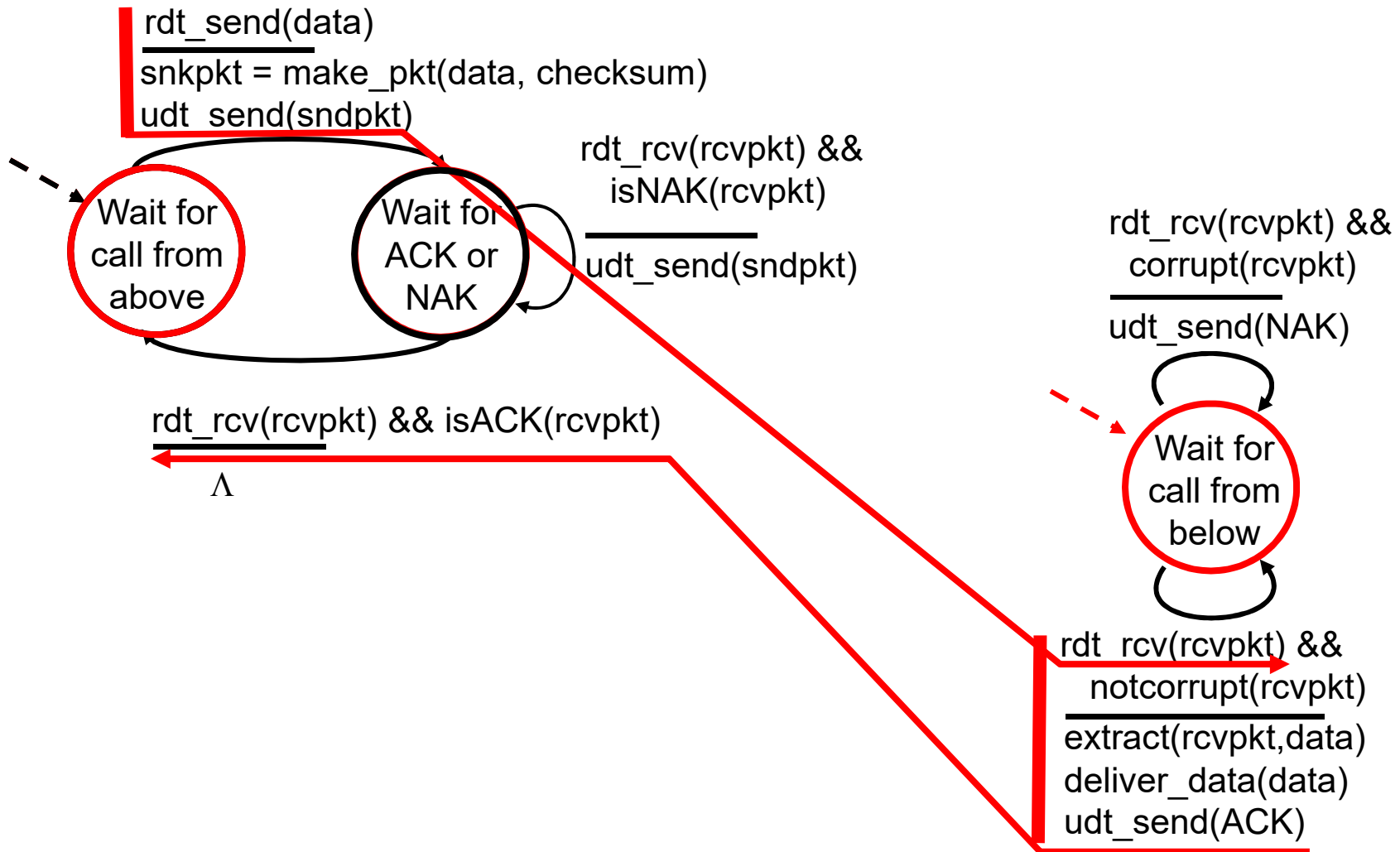


sender

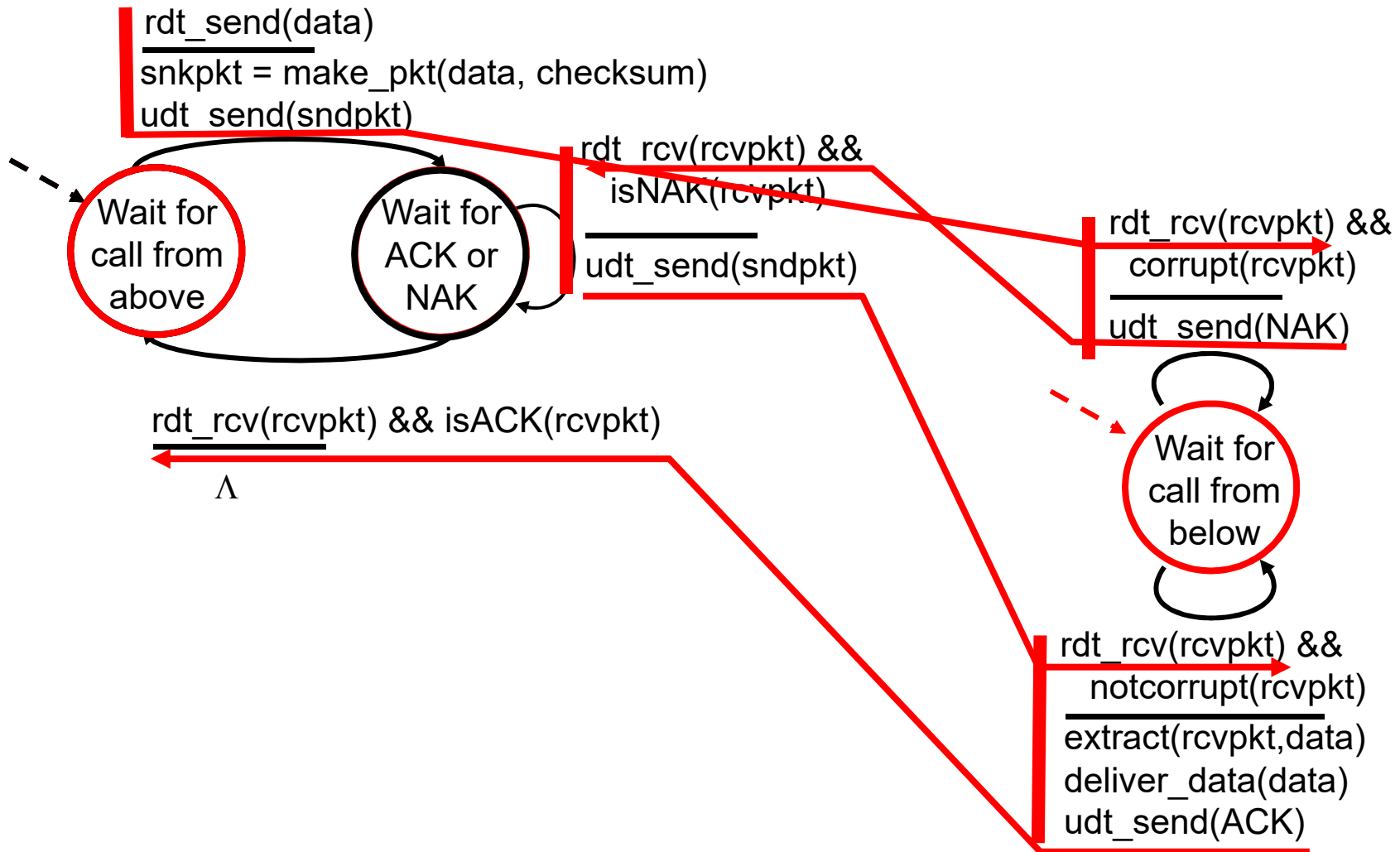
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario



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

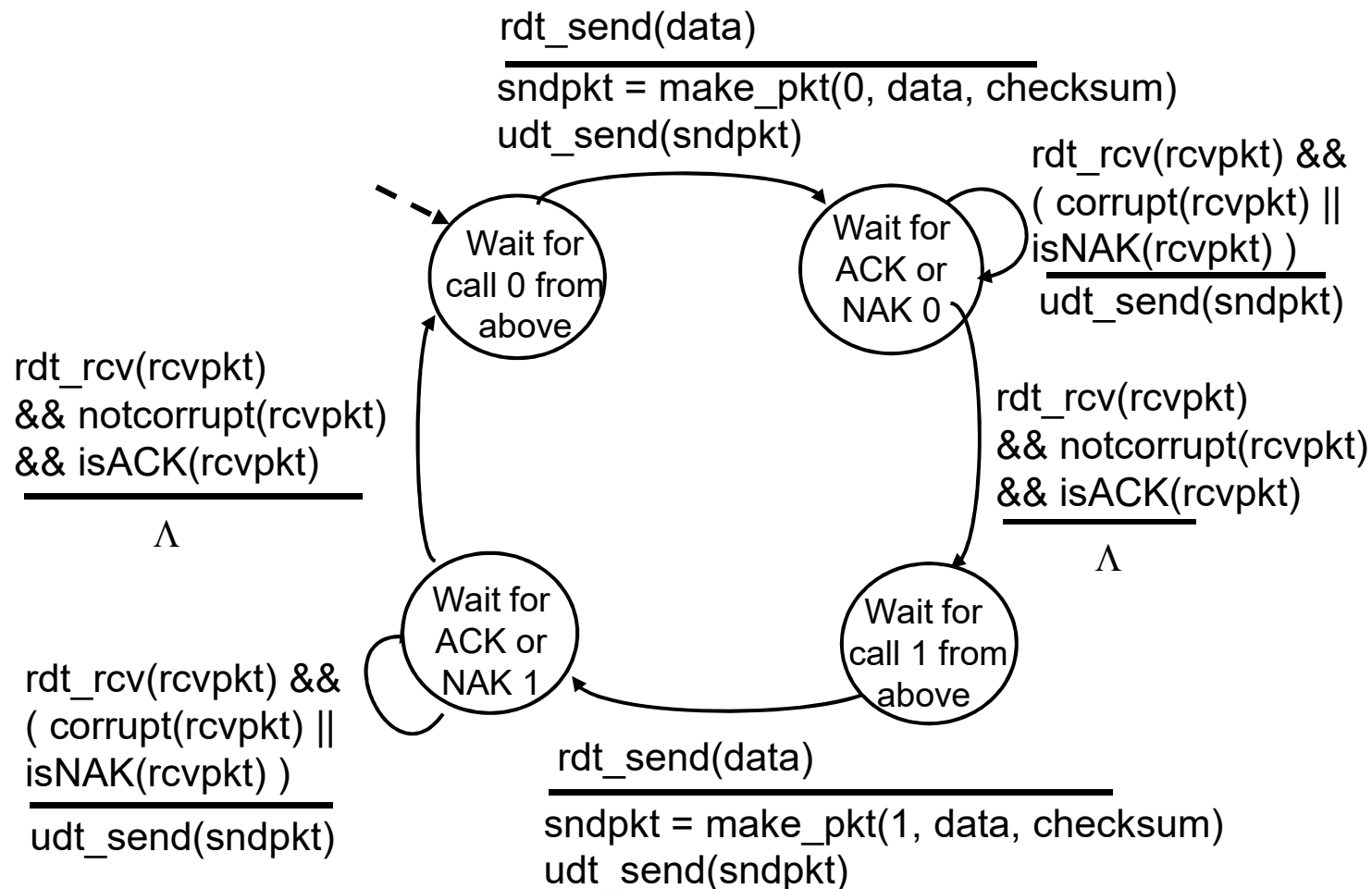
handling duplicates:

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

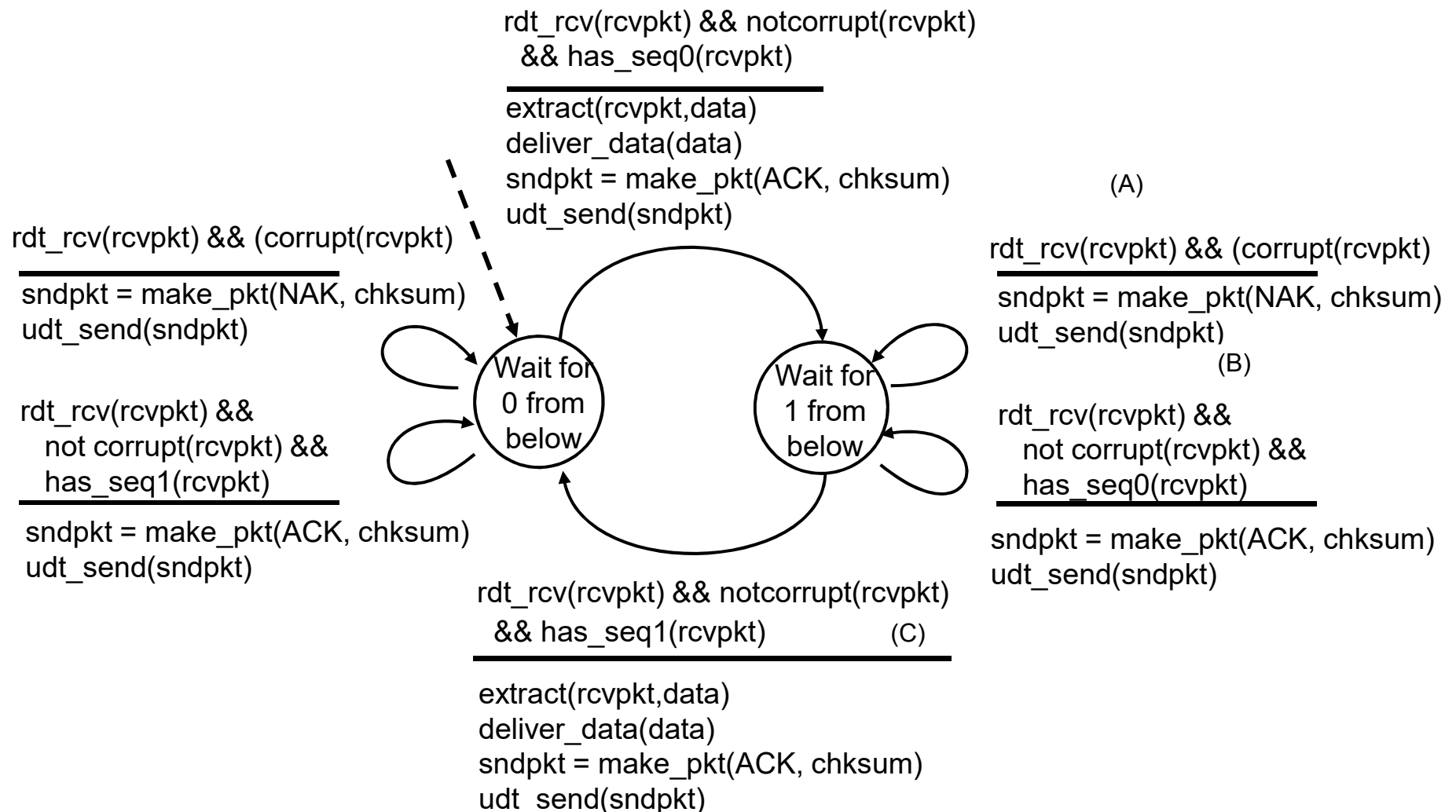
stop and wait

sender sends one packet,
then waits for receiver
response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



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 “expected” pkt should have seq # of 0 or 1

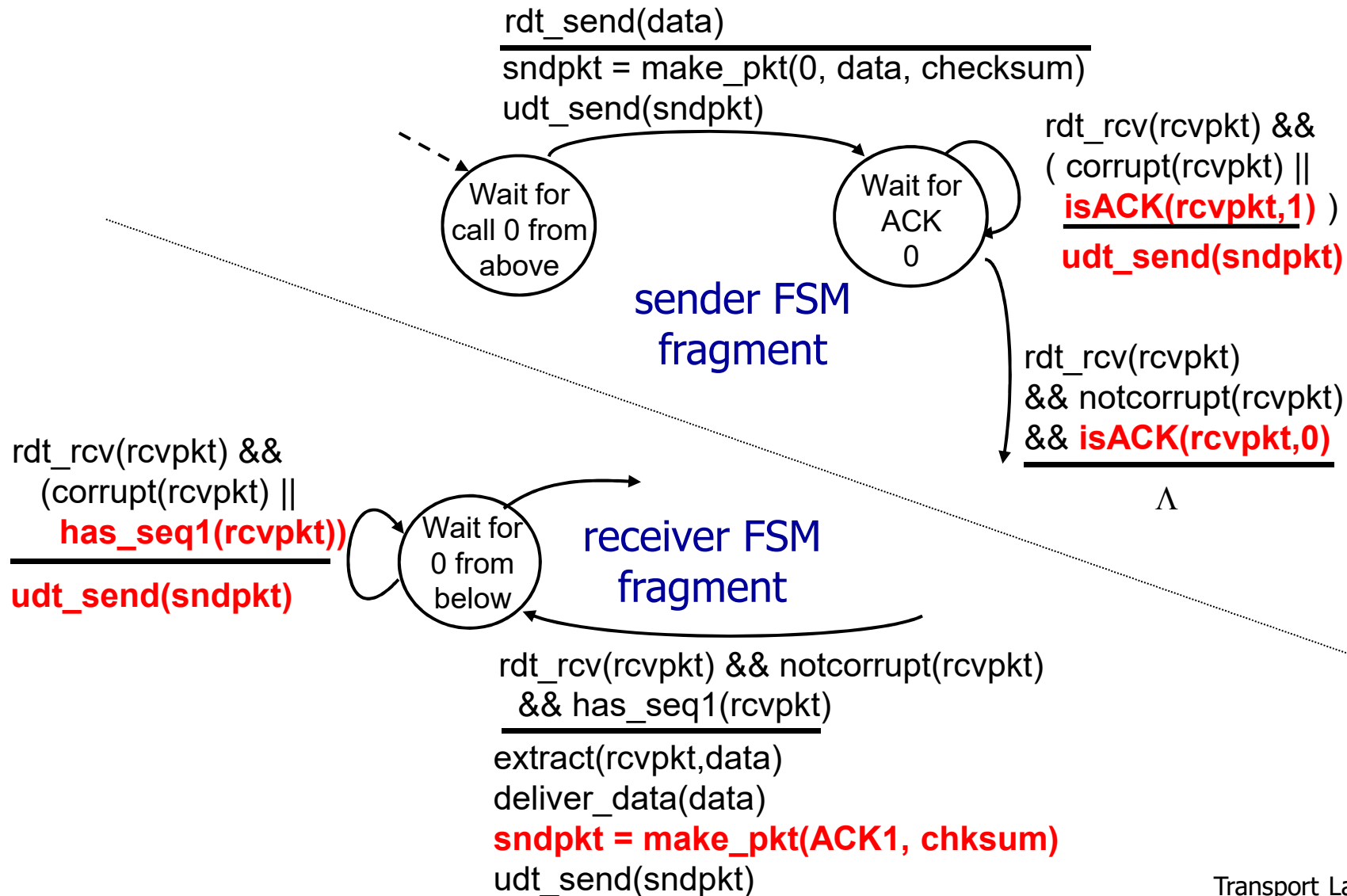
receiver:

- 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

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*

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors and loss

new assumption:

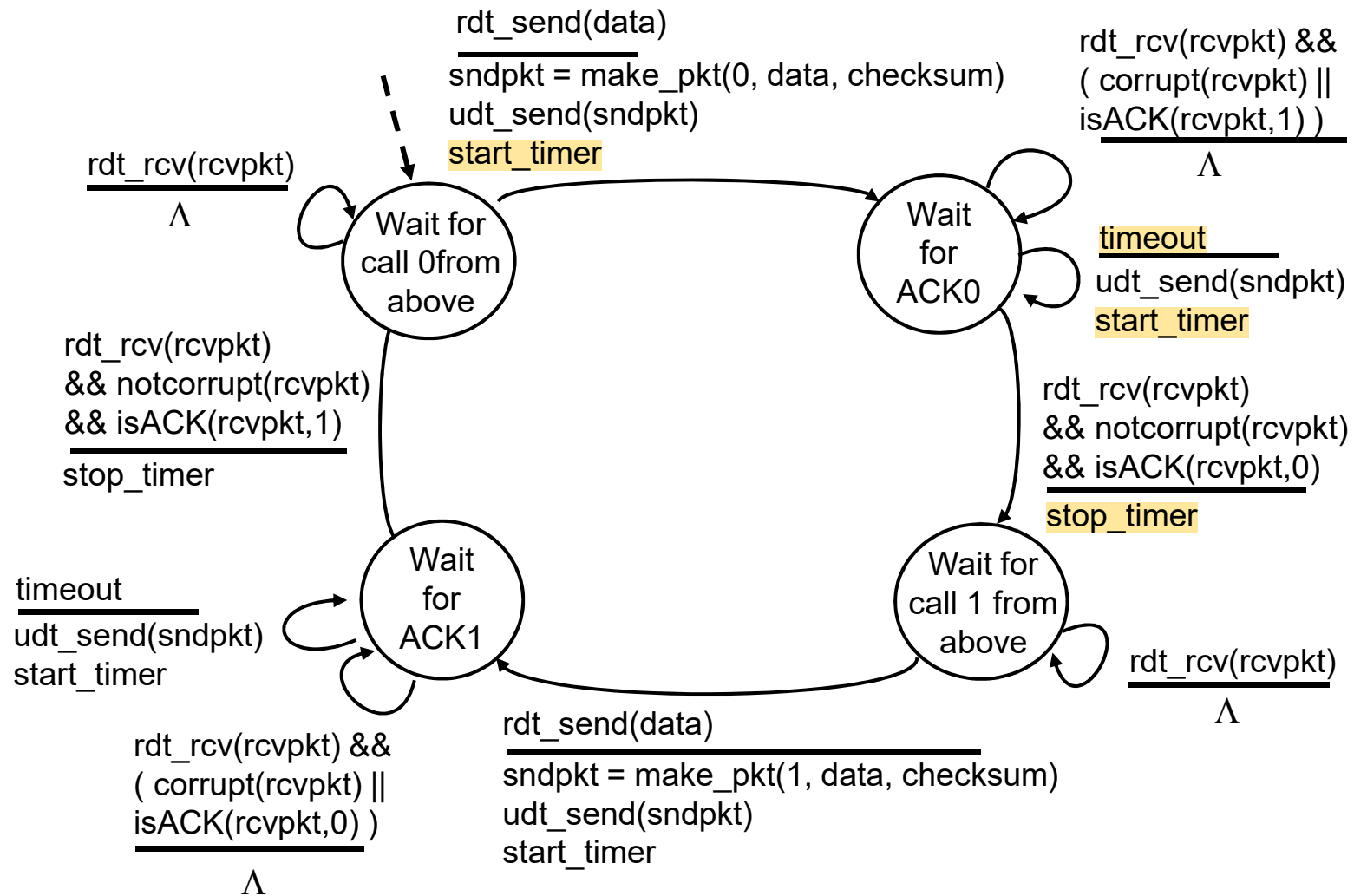
underlying channel can also lose packets (data, ACKs)

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

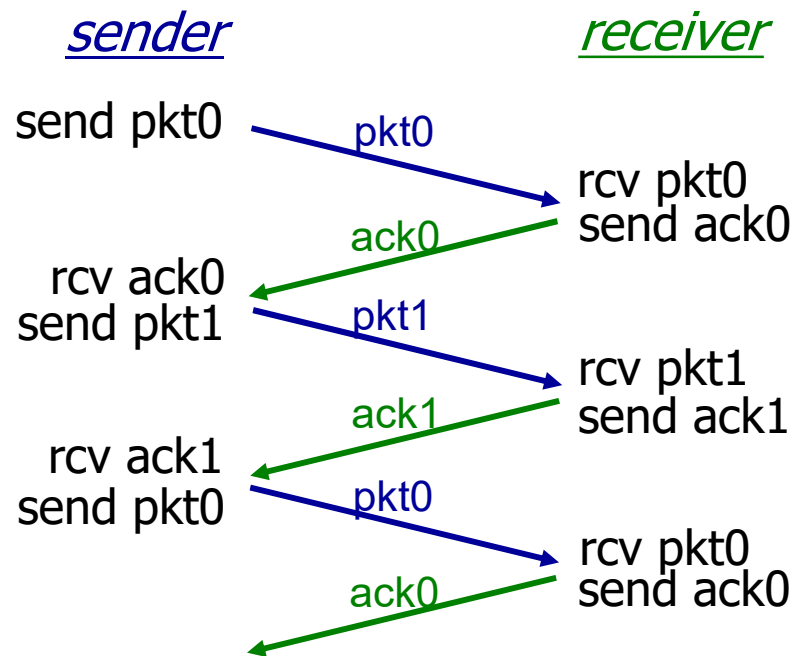
approach: sender waits “reasonable” amount of time for ACK

- 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

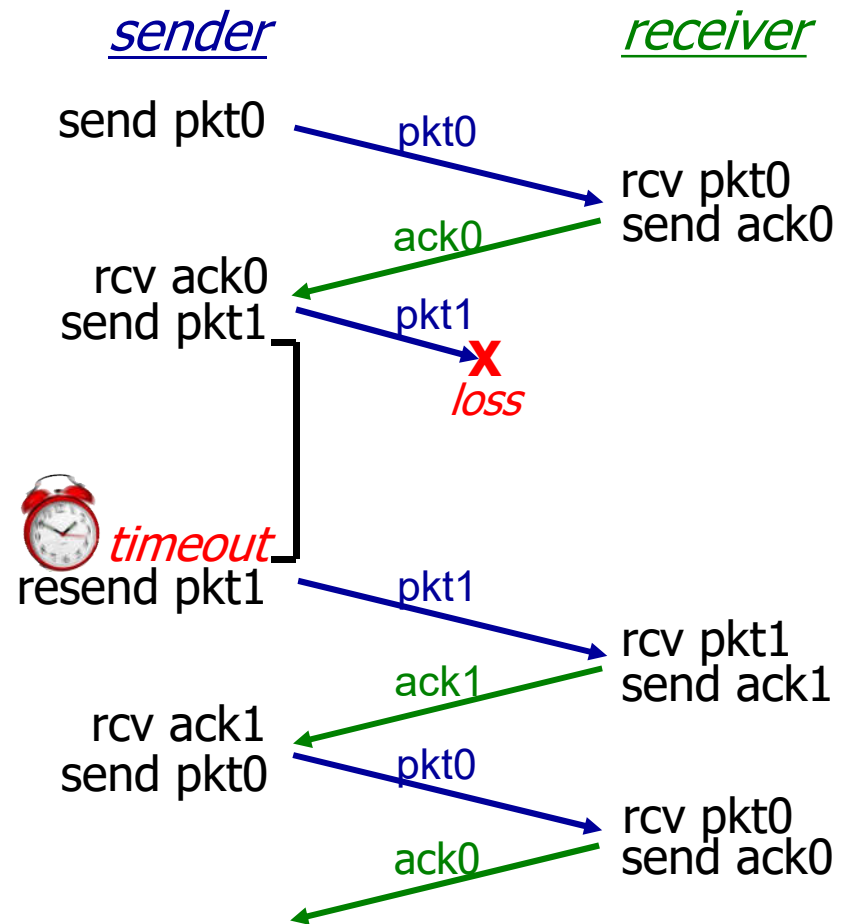
rdt3.0 sender



rdt3.0 in action

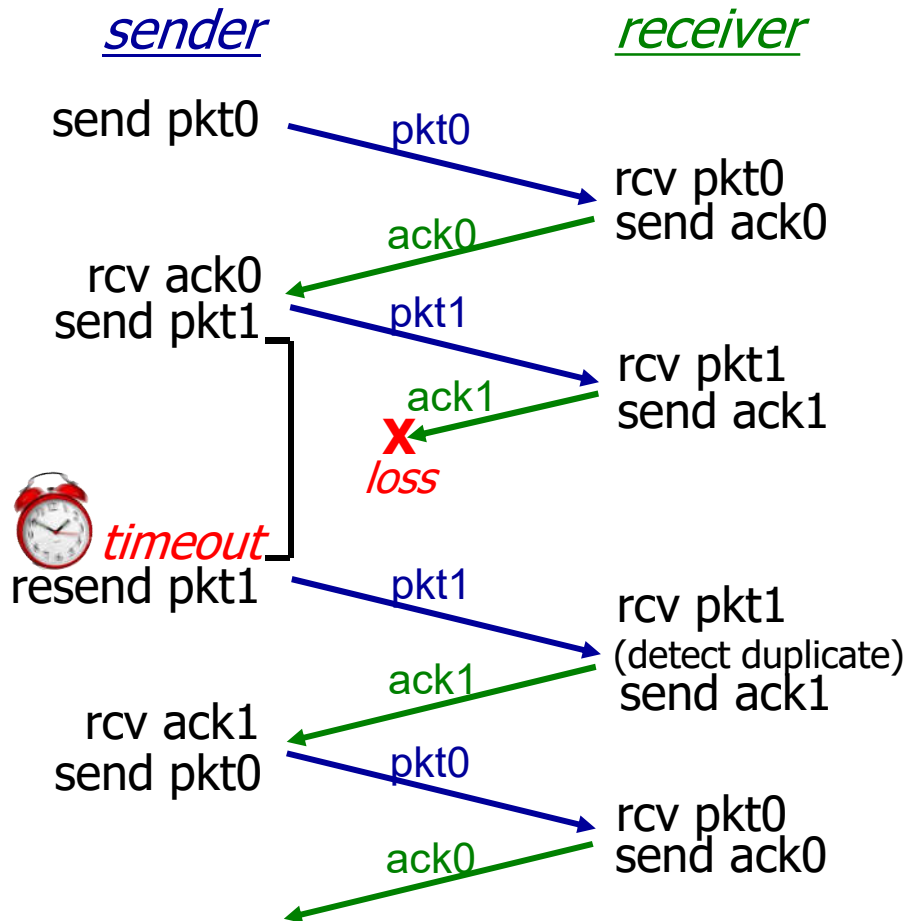


(a) no loss

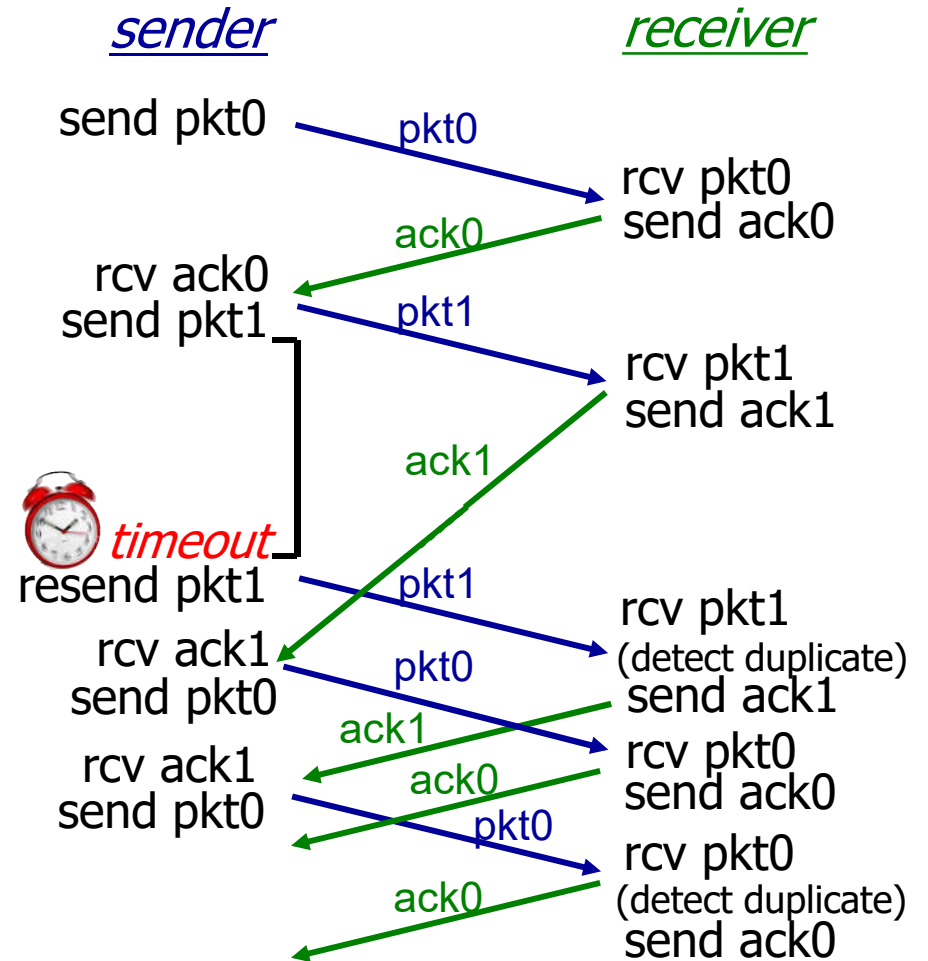


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

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

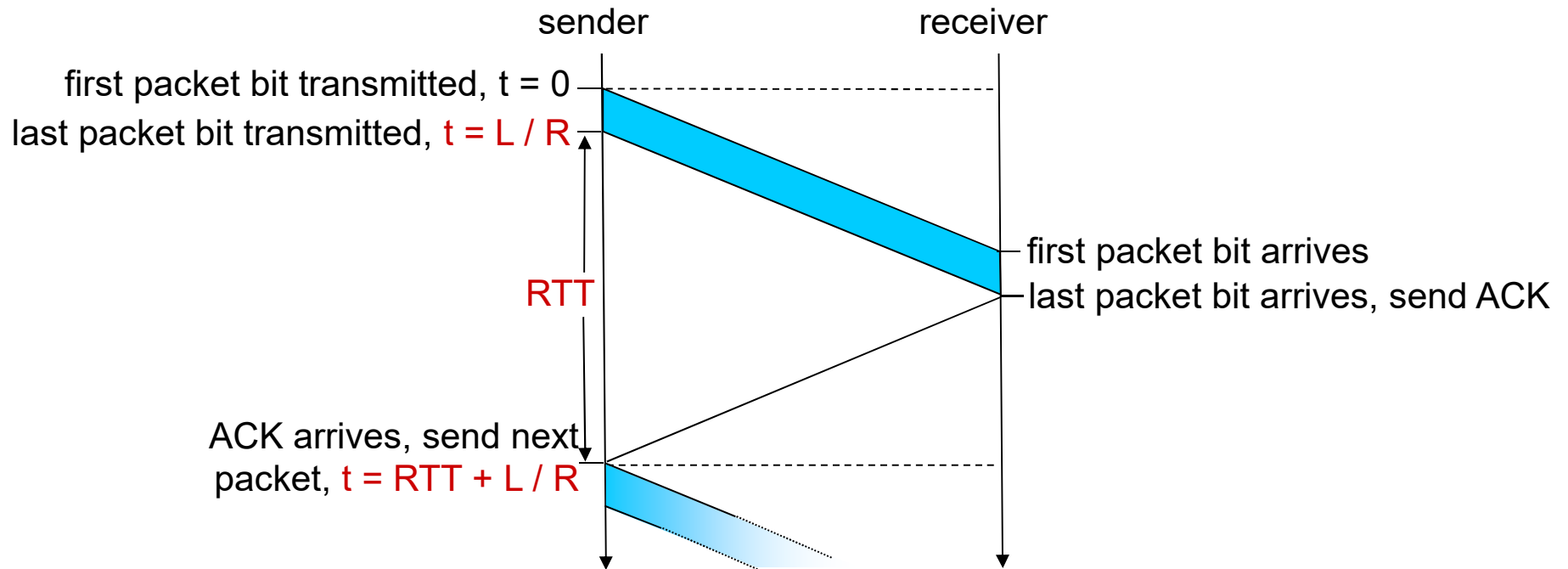
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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

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

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



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