Transport Layer

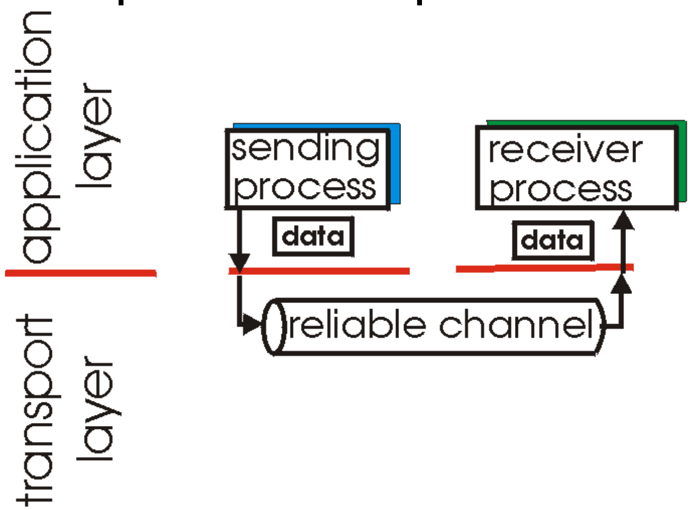
Instructor: C. Pu (Ph.D., Assistant Professor)

Lecture 09

puc@marshall.edu

Principles of Reliable Data Transfer

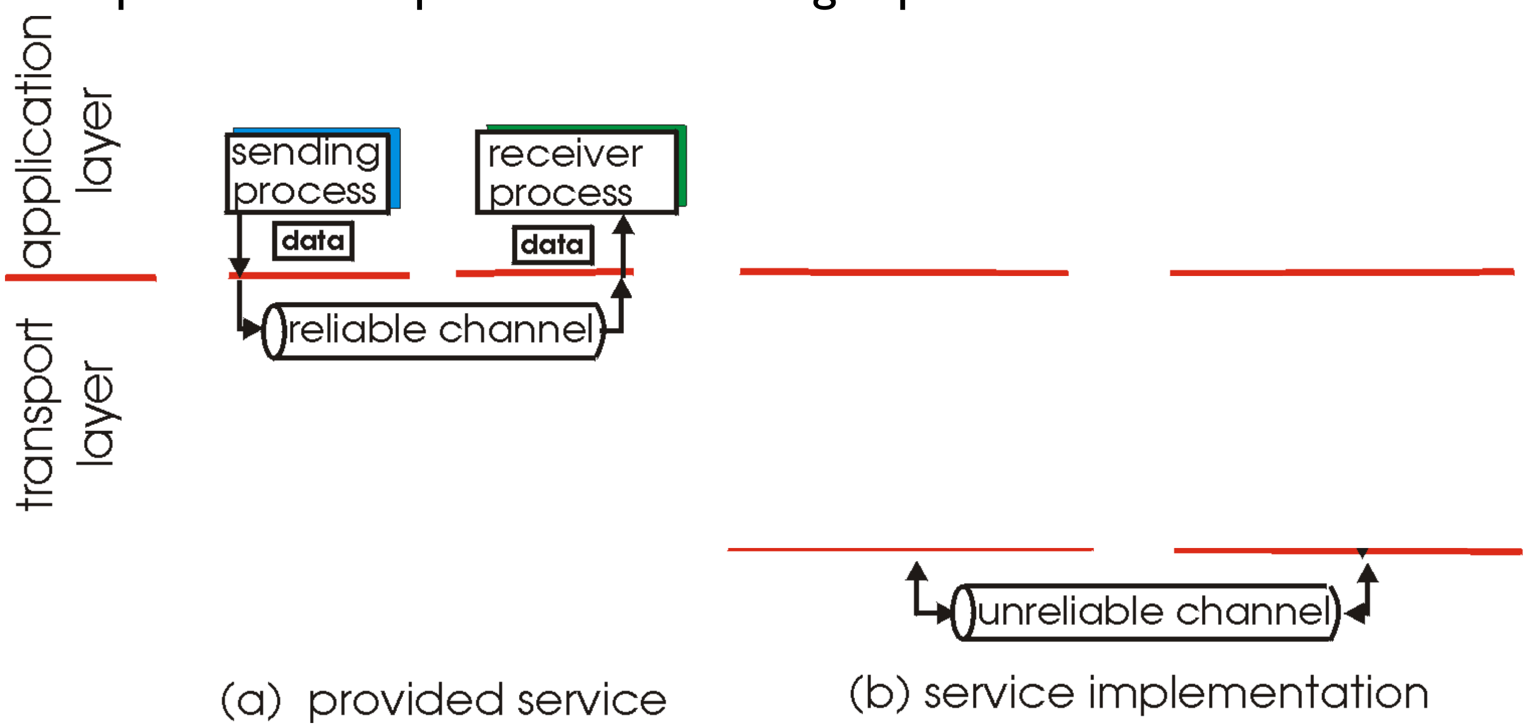
- important in app., transport, link layers
- top-10 list of important networking topics!



(a) provided service

Principles of Reliable Data Transfer (cont.)

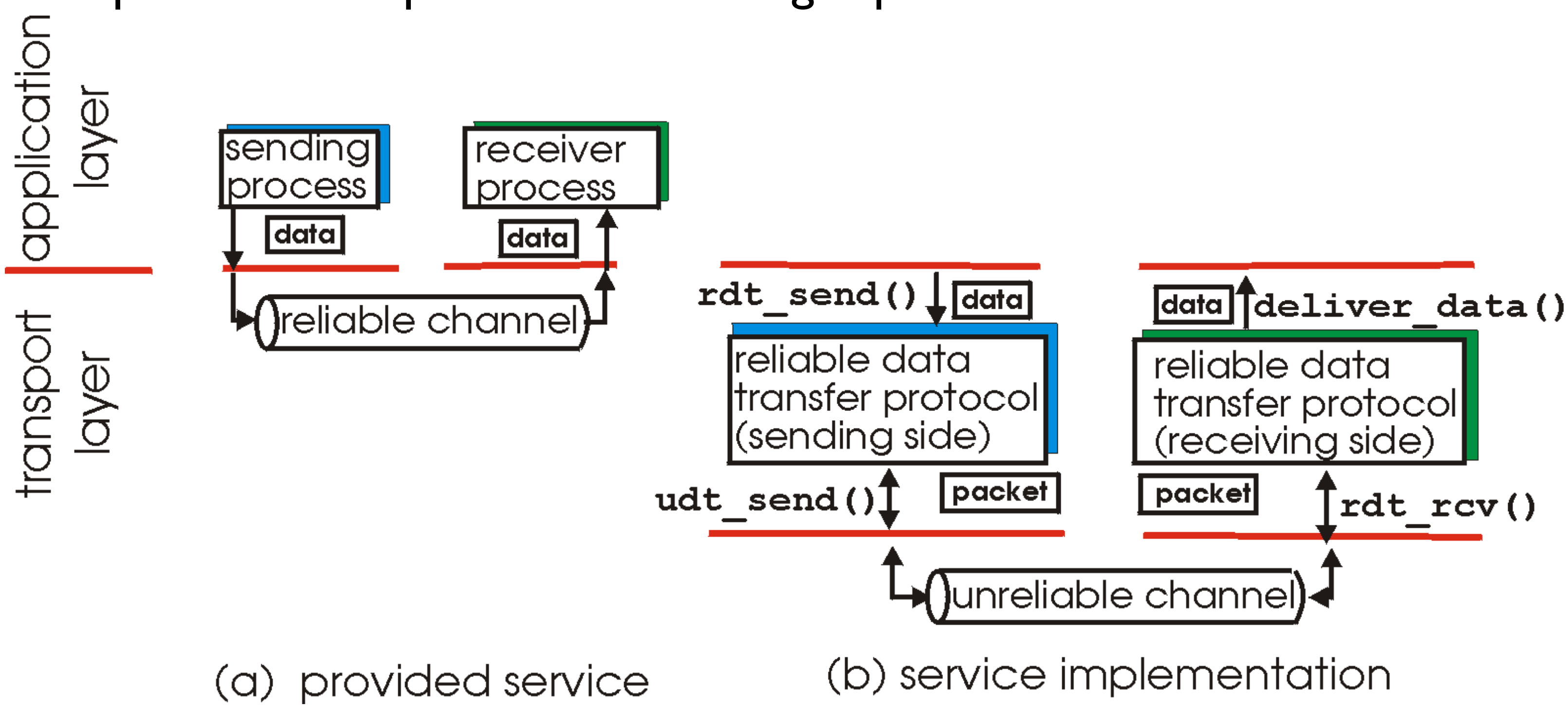
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable Data Transfer (cont.)

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

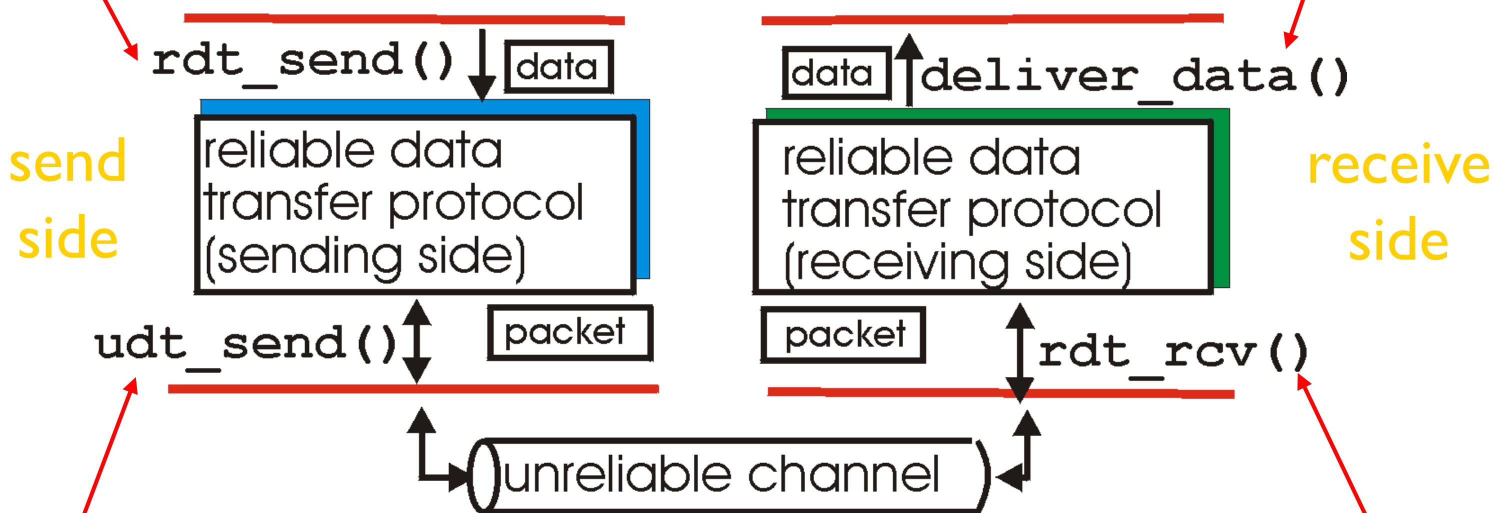


- characteristics of unreliable channel will determine complexity of reliable data transfer (rdt) protocol

Reliable Data Transfer: Getting Started

rdt_send(): called from above, (e.g., by app.). Pass data to be delivered to receiver upper layer

deliver_data(): called by rdt to deliver data to upper layer



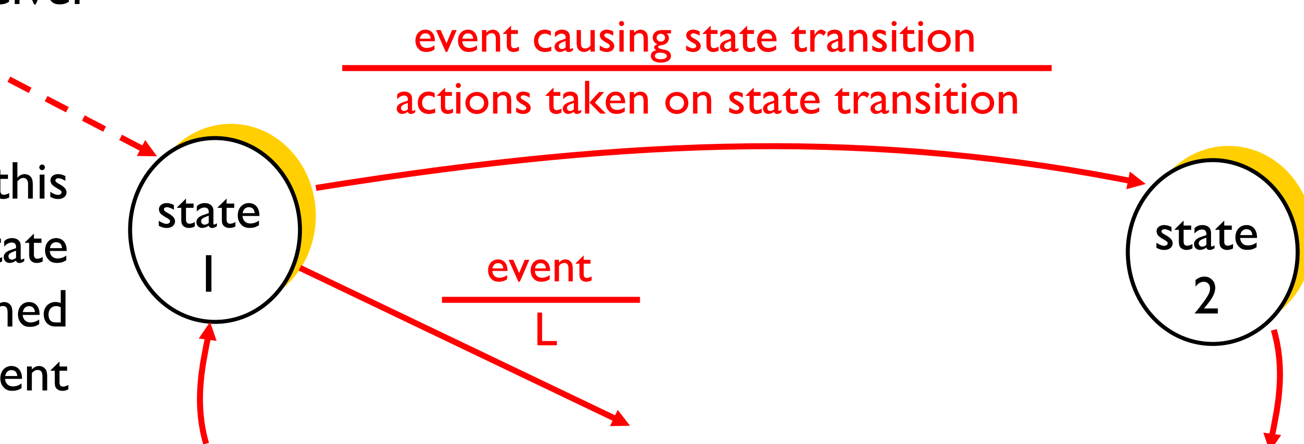
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable Data Transfer: Getting Started (cont.)

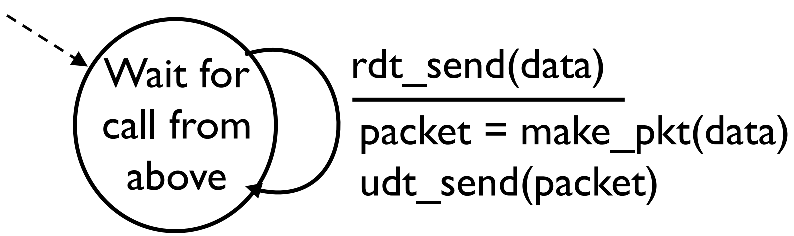
- We'll:
  - incrementally develop sender and receiver sides of ***reliable data transfer protocol*** (**rdt**)
  - consider only ***unidirectional data transfer***
    - but control info will flow on both directions!
  - use finite state machines (FSM) to specify the behaviors of sender and receiver

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

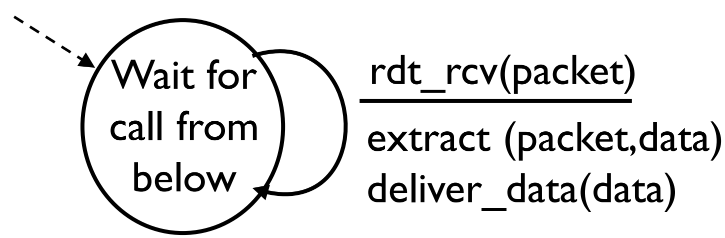


rdt1.0: Reliable Transfer over a Reliable Channel

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



sender



receiver

with a perfectly reliable channel, there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong!

rdt2.0: Channel with Bit Errors

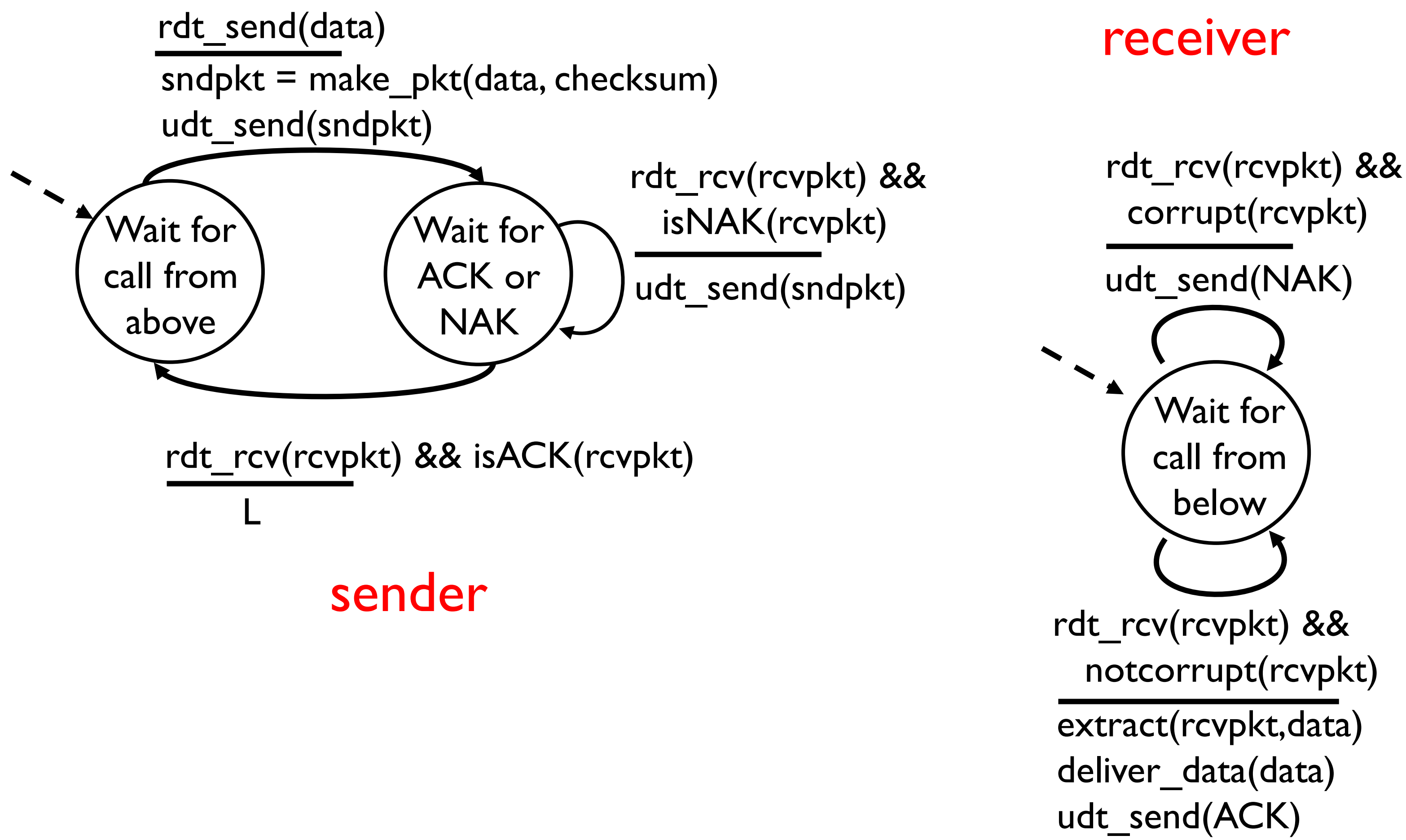
- consider a more realistic model
  - underlying channel may flip bits in packet
    - bit errors occur in the physical components of a network as a packet is transmitted, propagates, or is buffered
    - checksum to detect bit errors

- how people might deal with such a situation?
  - conversation over a phone
  - the message taker might say "OK" after each sentence has been heard, understood, and recorded
  - if the message taker hears garbled sentence, the message sender is asked to repeat

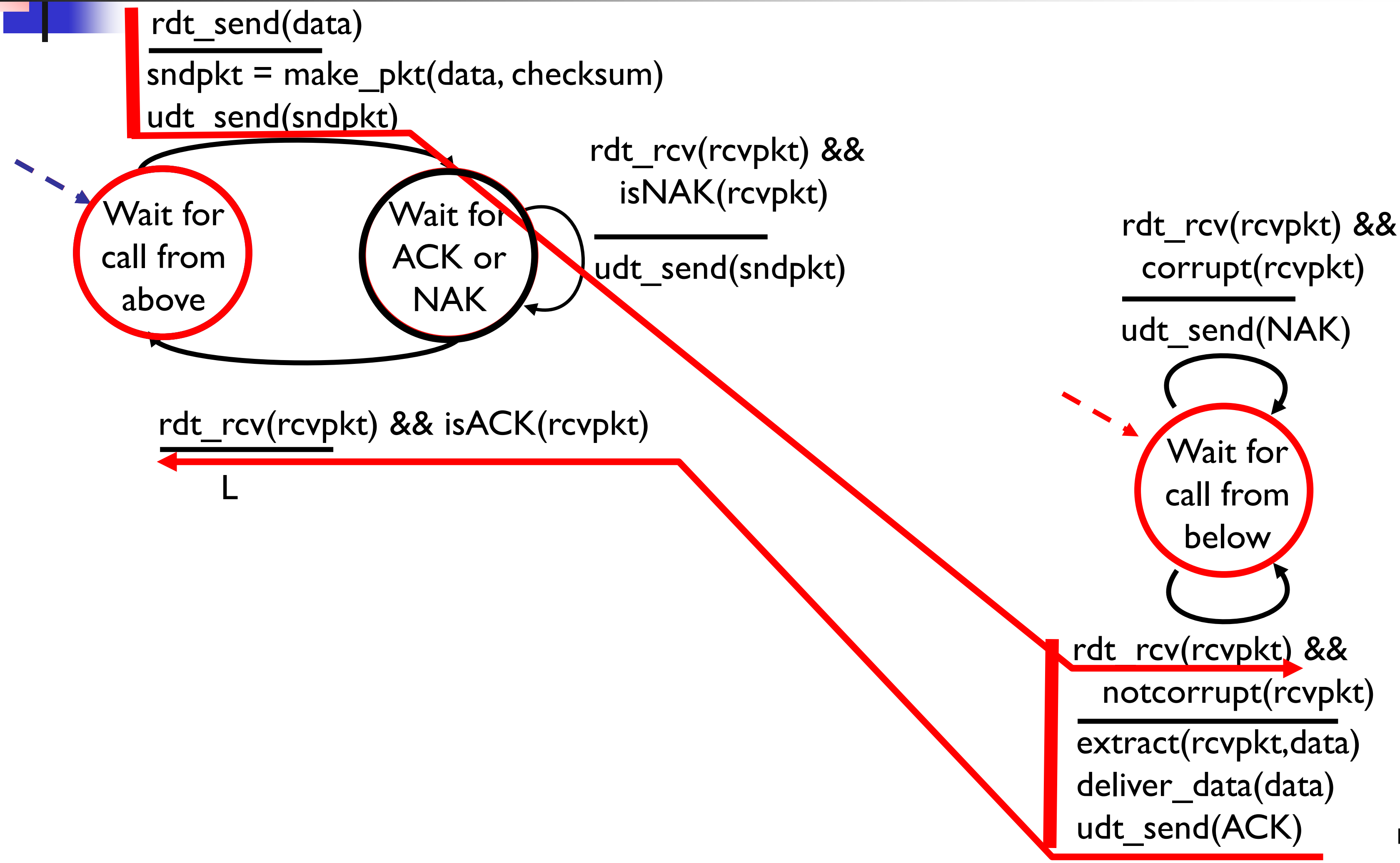
rdt2.0: Channel with Bit Errors

- the question: how to recover from errors?
  - *positive acknowledgements (ACKs):*
    - receiver explicitly tells sender that pkt received OK with *ACKs*
  - *negative acknowledgements (NAKs):*
    - receiver explicitly tells sender that pkt had errors with *NAK*
    - sender retransmits pkt on receipt of *NAK*
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - *receiver feedback*: control msgs (*ACKs* and *NAKs*) from receiver to sender
    - positive acknowledgement (*ACKs*)
    - negative acknowledgement (*NAKs*)

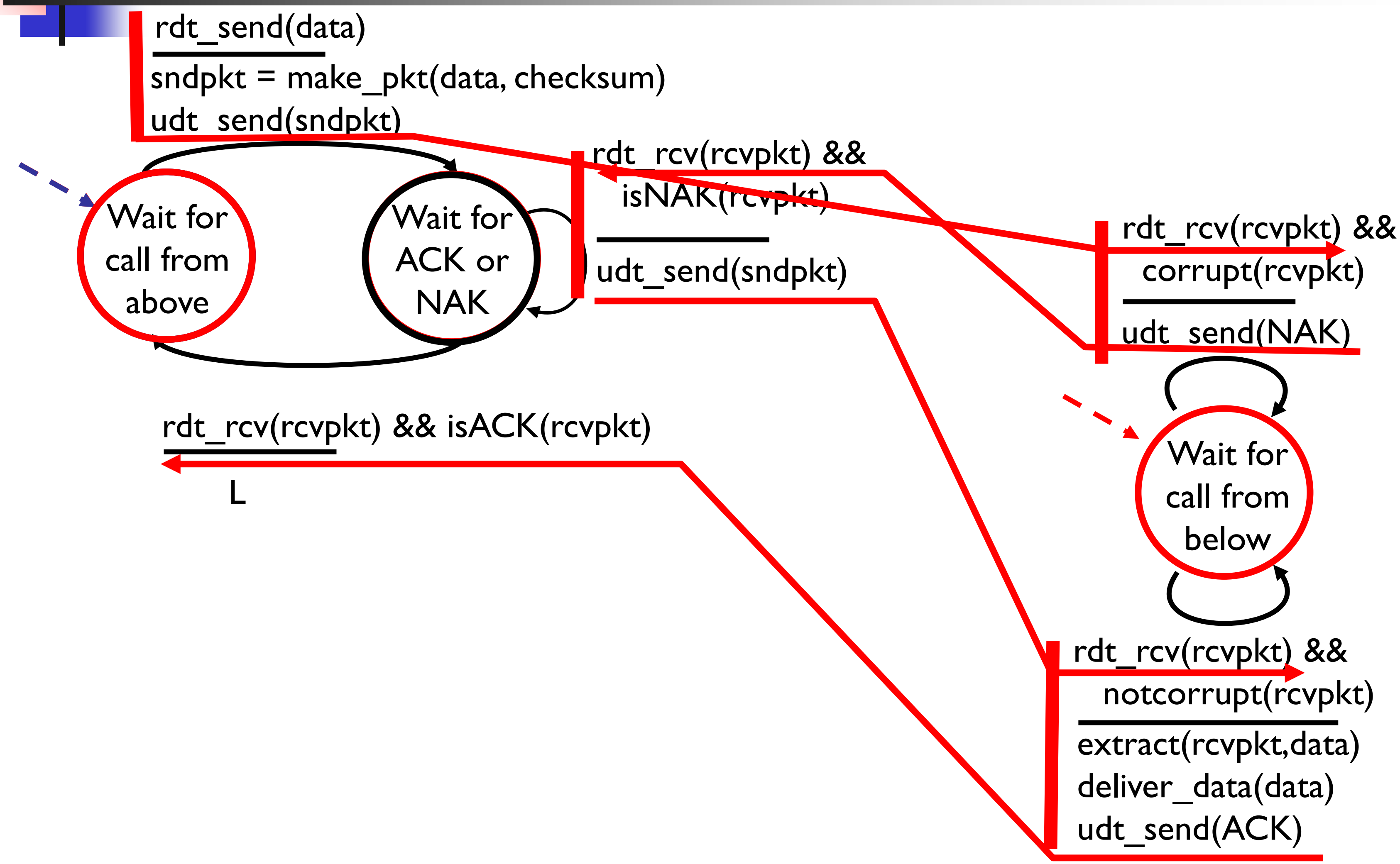
rdt2.0: FSM specification



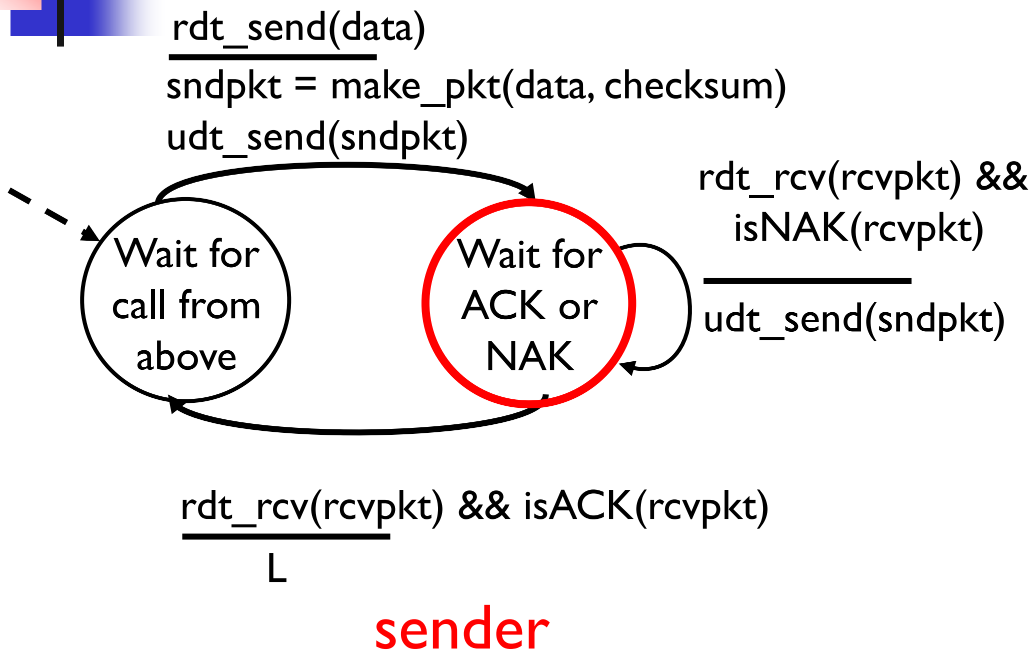
rdt2.0: Operation with No Errors

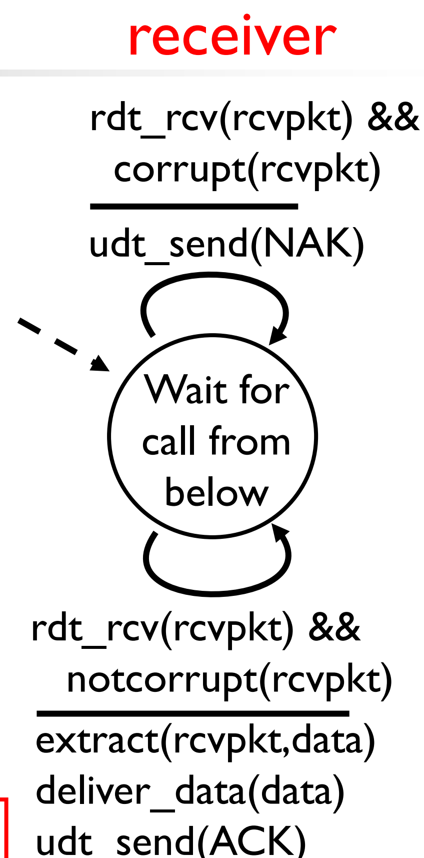


rdt2.0: Error Scenario



rdt2.0: FSM specification





stop and wait

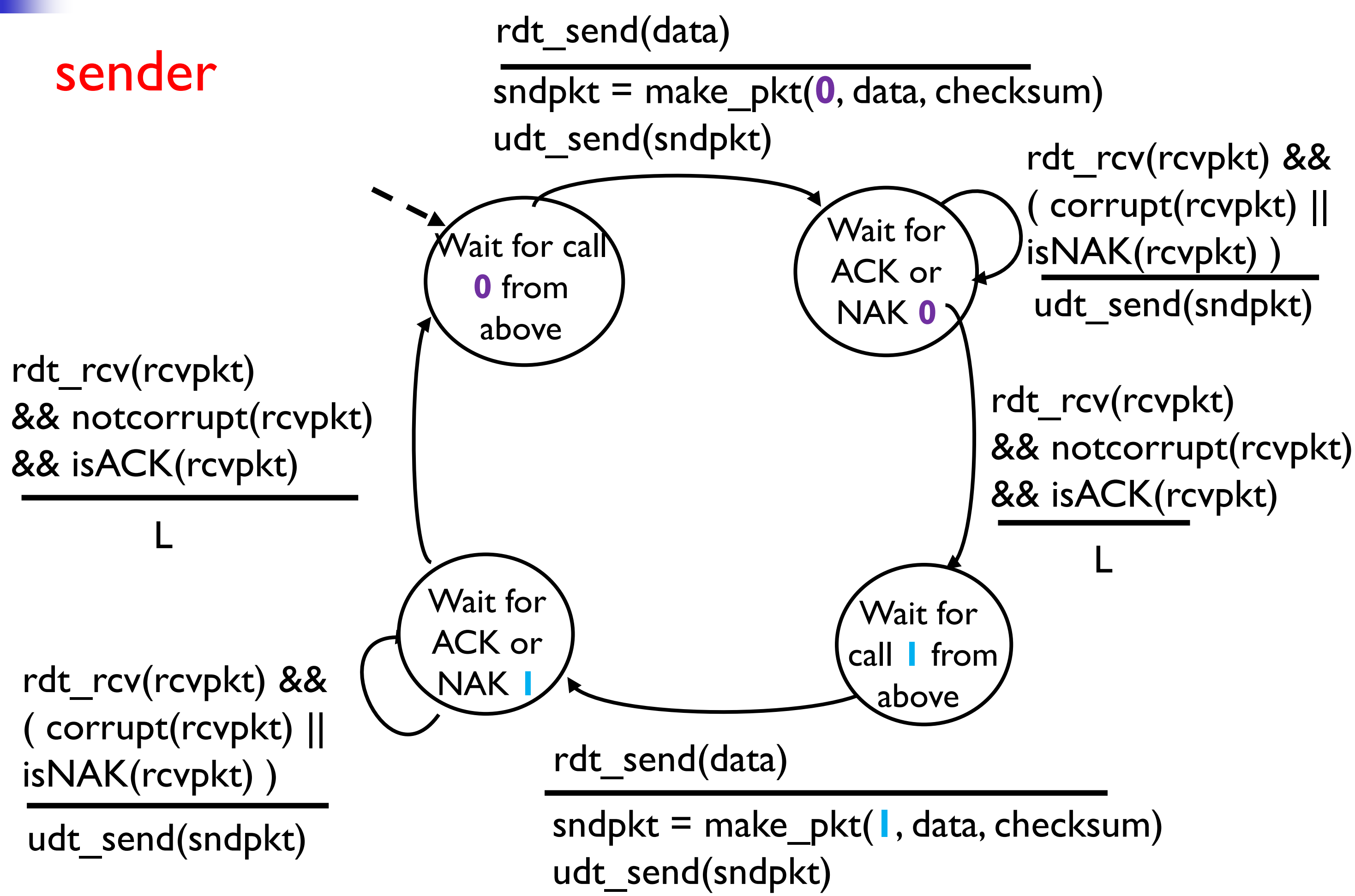
sender sends one packet, then waits for receiver response.

sender will not send a new data until it is sure that the receiver has correctly received the current packet.

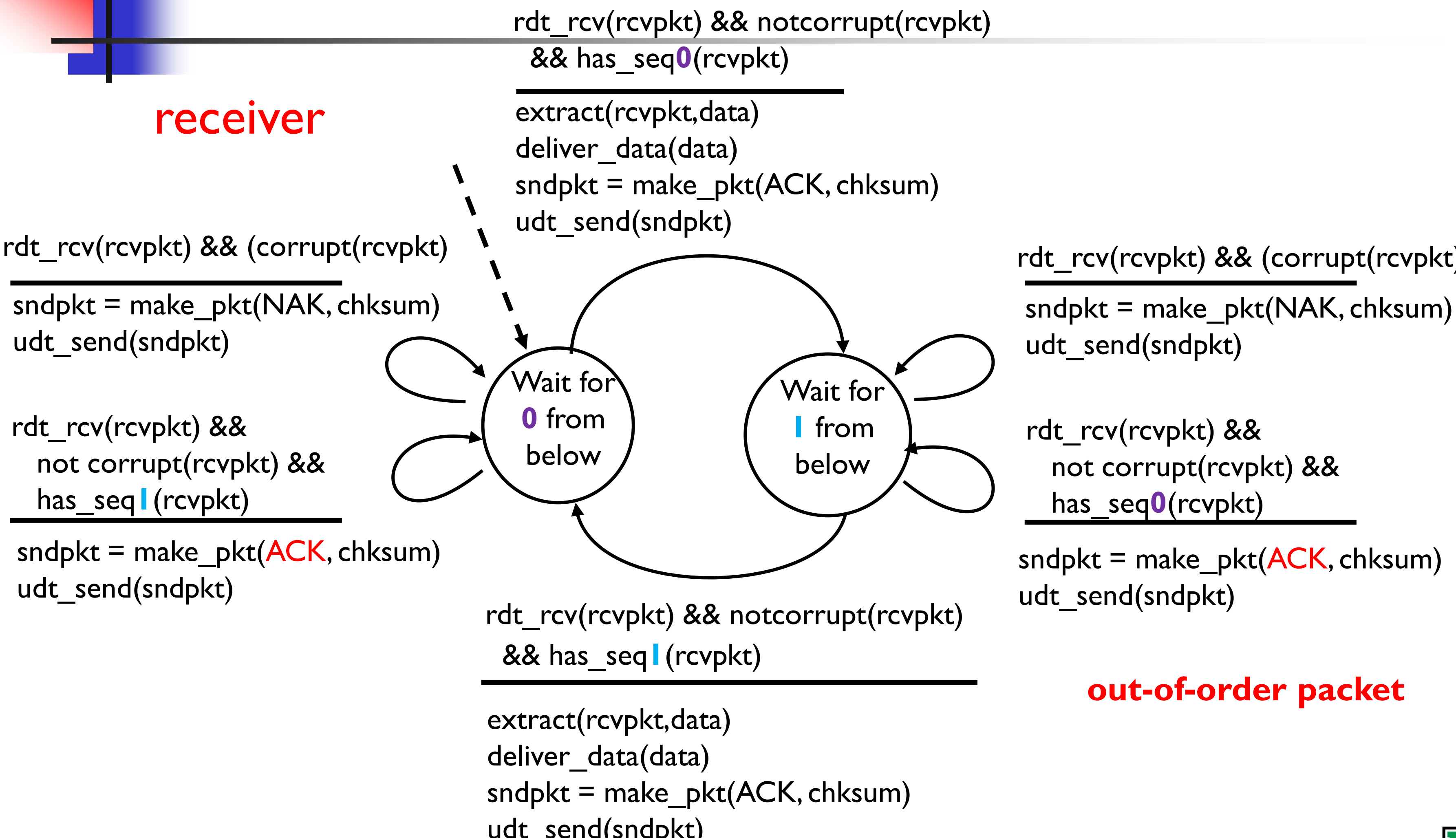
rdt2.0 has a Fatal Flaw!

- What happens if ACK/NAK corrupted?
  - add checksum to ACK/NAK to detect errors
  - more difficult question: how to *recover* from errors in ACK/NCK
  - sender doesn't know what happened at receiver!
    - sender does not know whether or not the receiver has correctly received the last piece of transmitted data
  - can't just retransmit: possible duplicate
- Handling duplicates:
  - sender retransmits current pkt if ACK/NAK garbled
  - sender adds *sequence number*, 1-bit, to each pkt
  - receiver discards (doesn't deliver up) duplicate pkt

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 "current" pkt has 0 or 1 seq. #
- 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

rdt3.0: Channels with Errors and Loss

- new assumption: underlying channel can also *lose packets* (data or ACKs)
  - two additional concerns:
    - how to detect packet loss?
    - what to do when packet loss occurs?
      - checksum, seq. #, ACKs, retransmissions will be of help

- approach: sender waits "reasonable" amount of time for ACK
  - detecting & recovering from lost packets from the sender side
  - how long must the sender wait?
    - at least as long as a round trip time (RTT) between sender and receiver

rdt3.0: Channels with Errors and Loss (cont.)

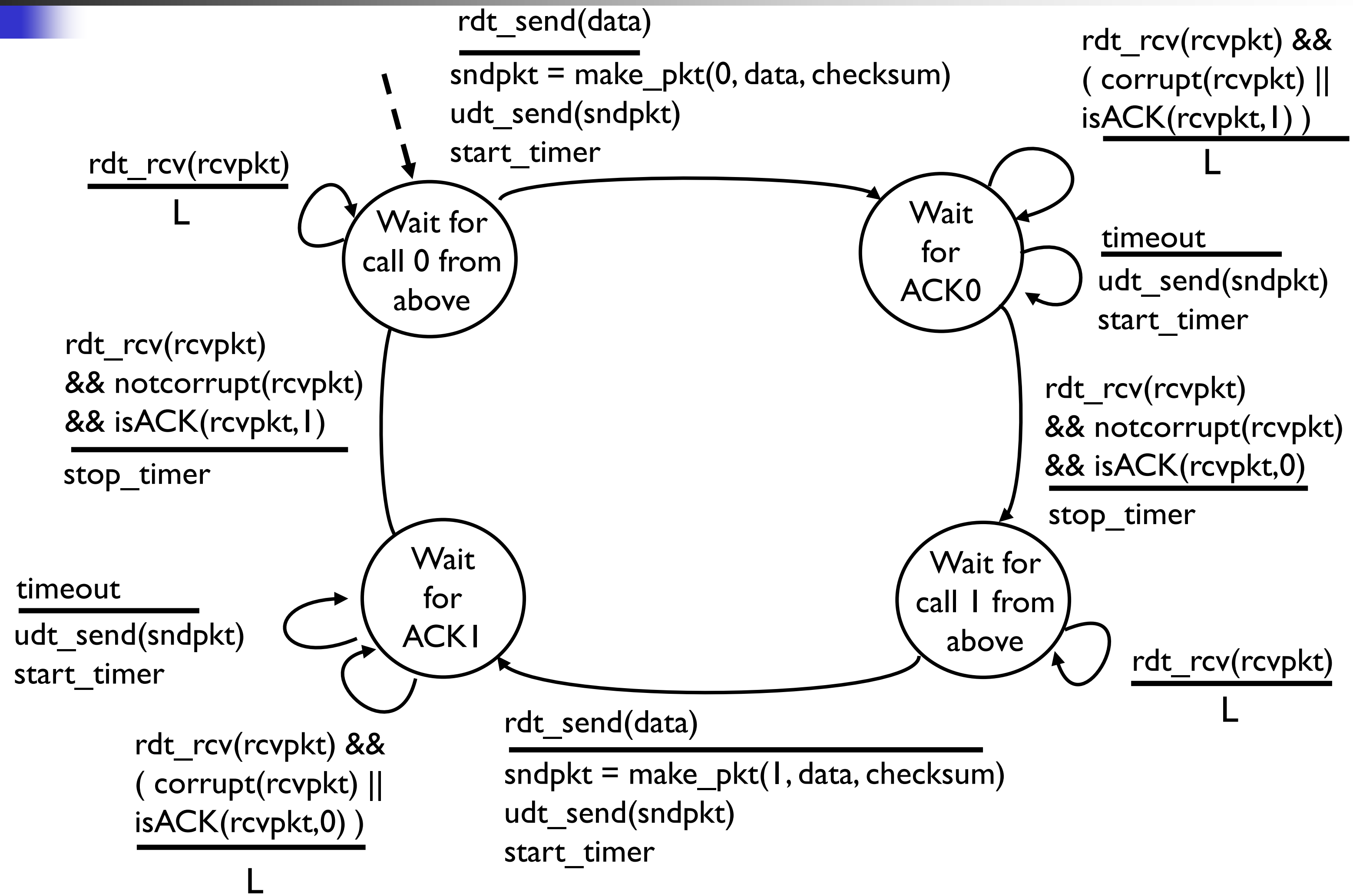
- approach: sender waits "reasonable" amount of time for ACK (cont.)
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #'s already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires ***countdown timer*** (the sender):
    - (1) start the timer each time when a packet is sent
    - (2) respond to a timer interrupt
    - (3) stop the timer

rdt3.0: Channels with Errors and Loss (cont.)

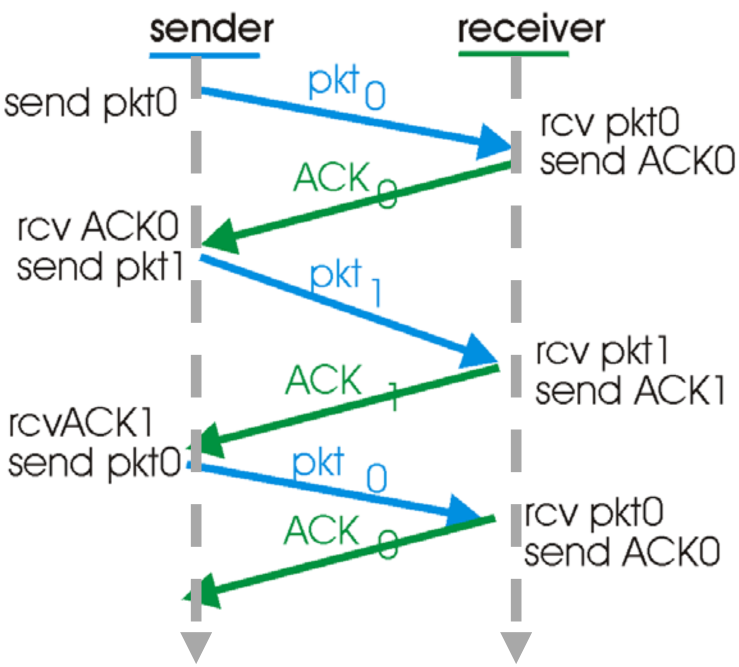
- from the sender's point of view, the sender does NOT know whether
  - a data packet was lost
  - an ACK was lost
  - if the packet or ACK was simple overly delayed
  - → in all case, the sender just retransmits!!

rdt3.0: Sender

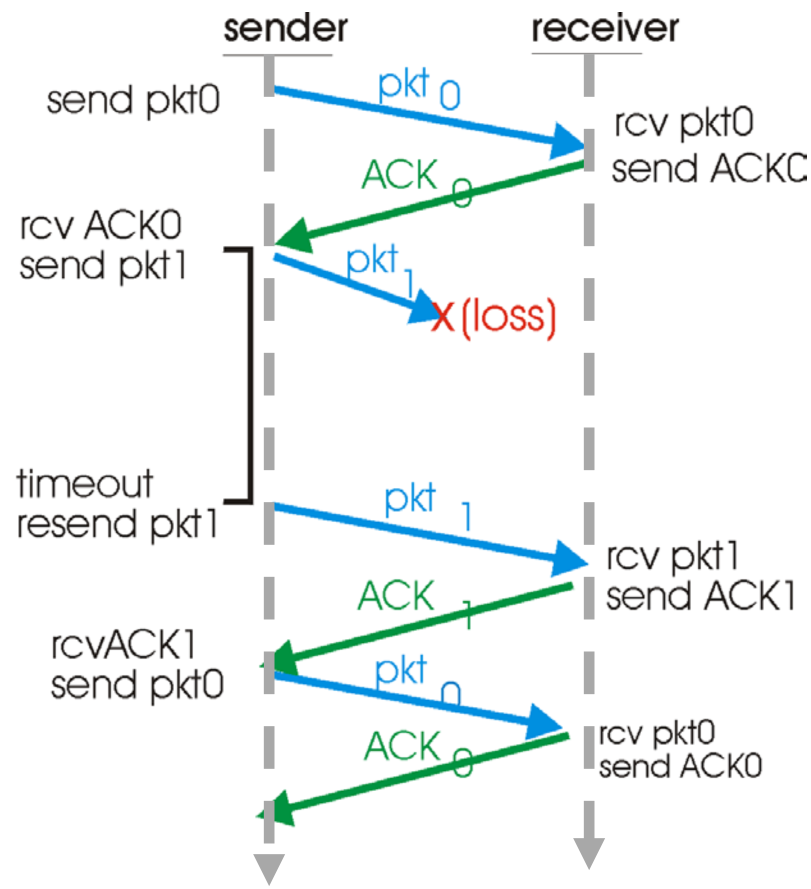
What about a receiver side?



rdt3.0 in Action

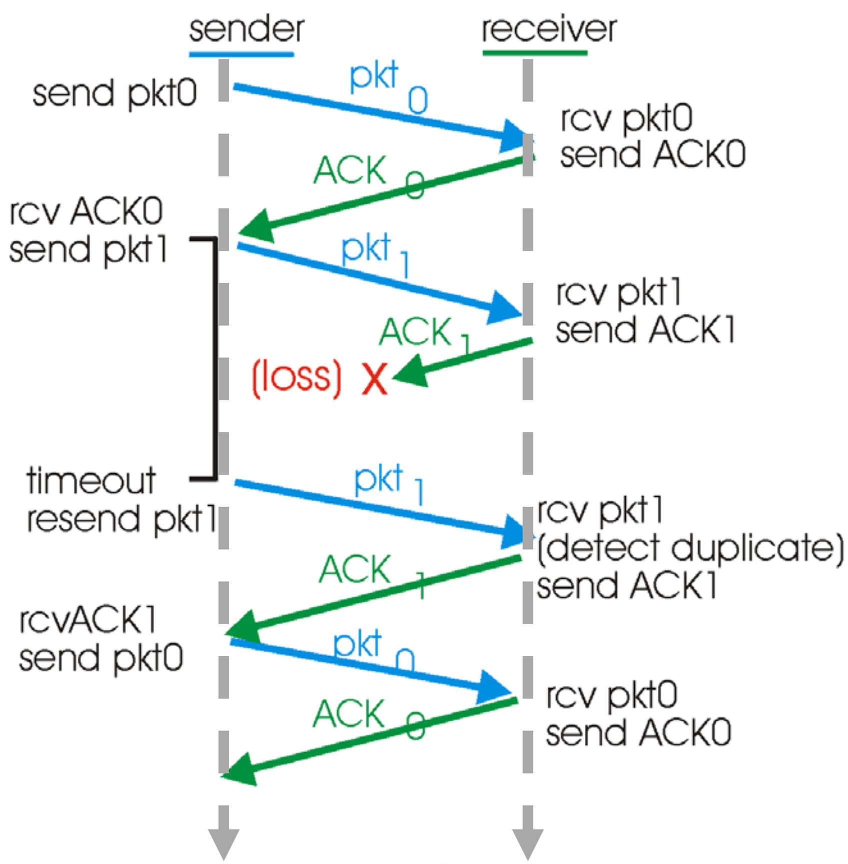


(a) operation with no loss

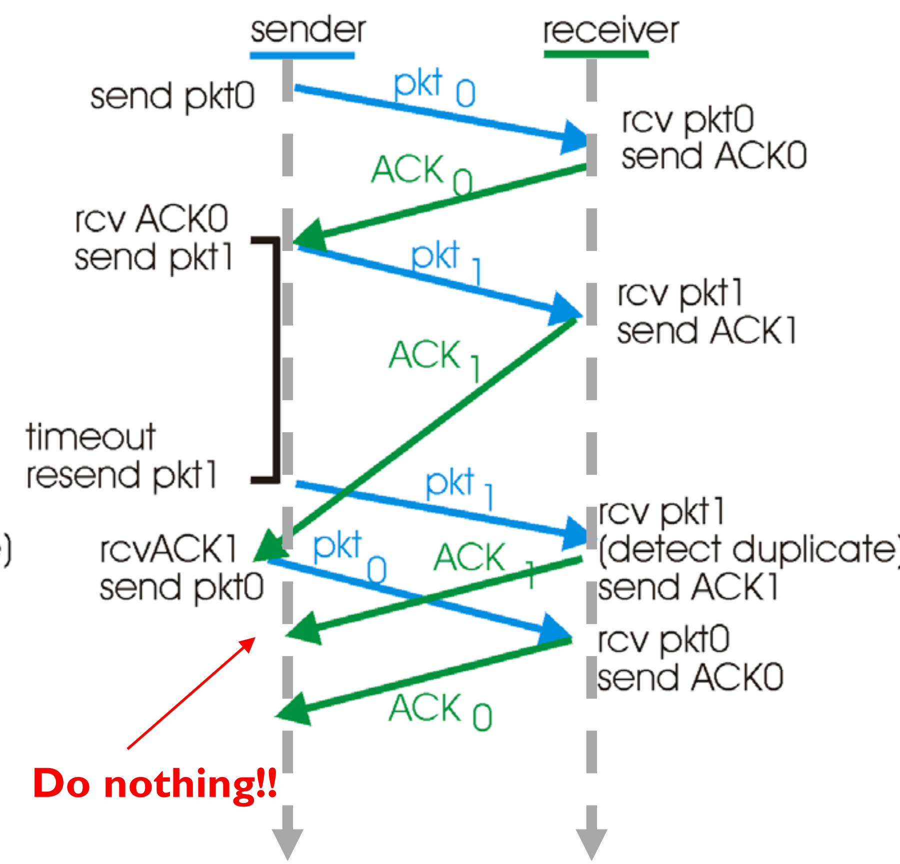


(b) lost packet

rdt3.0 in Action (cont.)



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

- rdt3.0 works, but performance stinks
  - For example: 1 Gbps link, 15 ms end-to-end delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8000 \text{ bit/pkt}}{10^{**}9 \text{ bit/sec}} = 8 \text{ microsec}$$

- RTT: the speed-of-light propagation delay, approximately 30 msec
  - the time t when the last bit of the packet emerging at the receiver

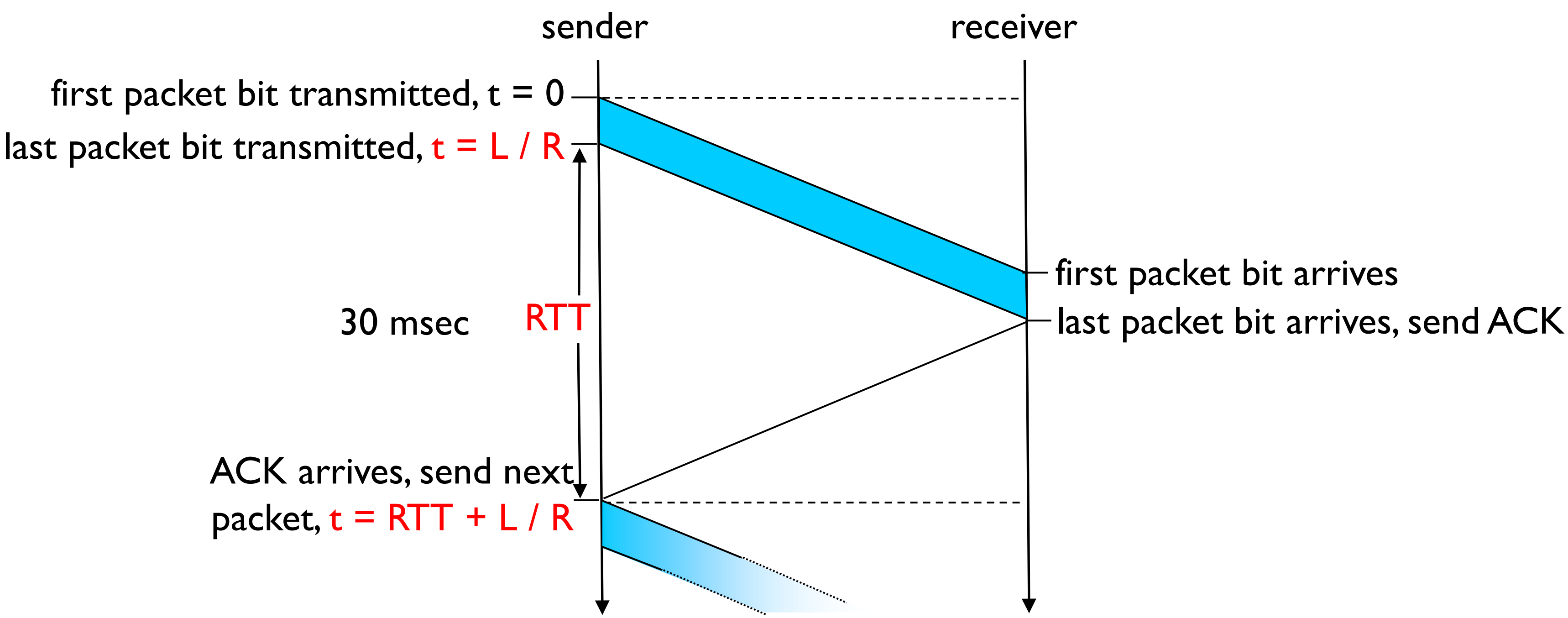
$$t = RTT/2 + L/R = 15.008 \text{ msec}$$

- When the ACK will come back?

$$t = RTT + L/R = 30.008 \text{ msec}$$

rdt3.0: stop-and-wait Operation

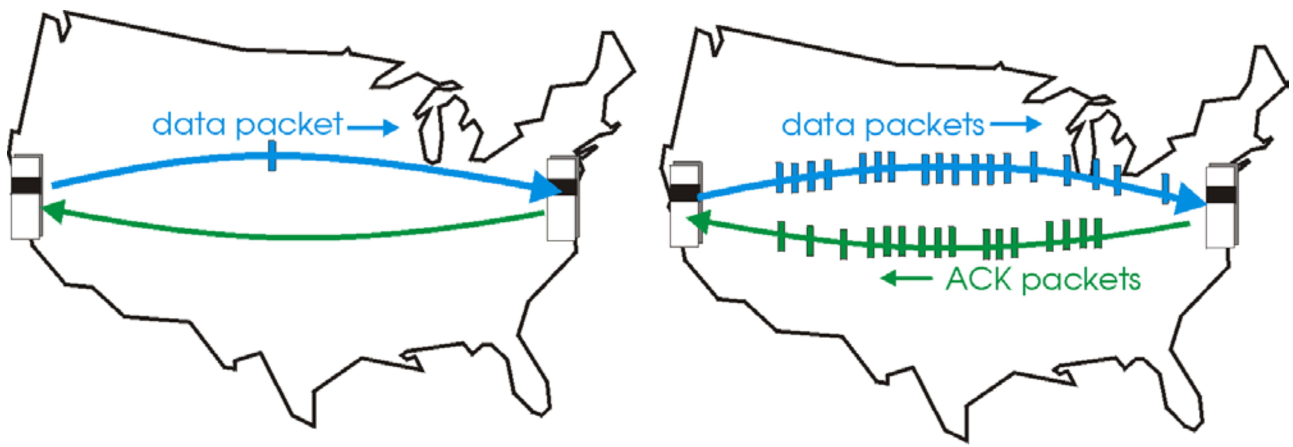
- U_{sender}: utilization – fraction of time sender busy sending



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

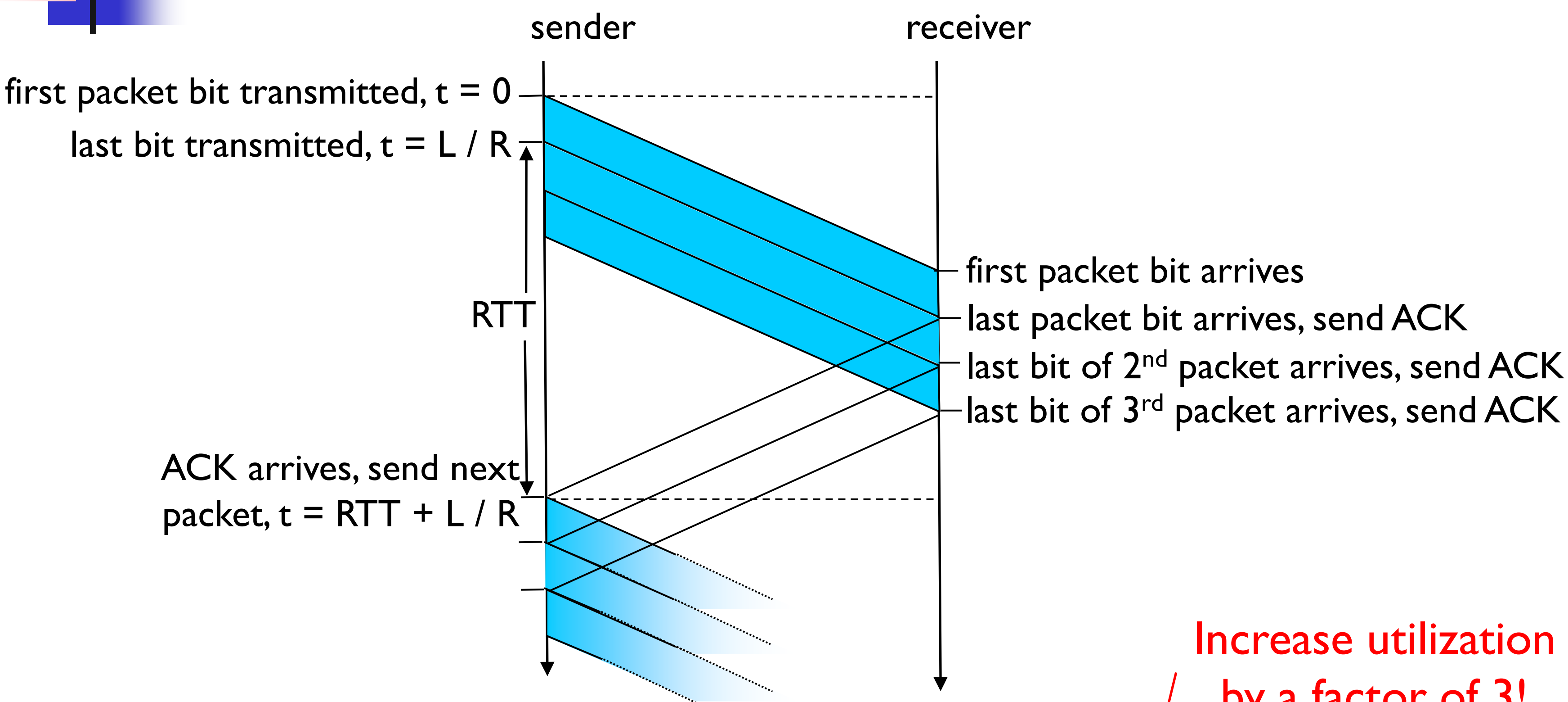
Pipelined Protocols

- pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts. sending multiple pkts without waiting for acknowledgments
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver
  - two generic forms of pipelined protocols: *Go-Back-N, selective repeat*



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

Pipelining: Increased Utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$