Transport Layer
Part 3

Mark Allman
mallman@case.edu

EECS 325/425
Fall 2018

"Oh give me the beat boys, and free my soul,
I wanna get lost in your rock 'n' roll…"

These slides are more-or-less directly from the slide set developed by Jim Kurose and Keith Ross for their book "Computer Networking: A Top Down Approach, 5th edition".

The slides have been lightly adapted for Mark Allman's EECS 325/425 Computer Networks class at Case Western Reserve University.

rdt2.0

rdt2.0

- **GOAL**: reliable data delivery

rdt2.0

- **GOAL**: reliable data delivery

- Path between sender and receiver unreliable because it can corrupt packets

rdt2.0

- **GOAL**: reliable data delivery

- Path between sender and receiver unreliable because it can corrupt packets

- Introduced three mechanisms to cope with this:

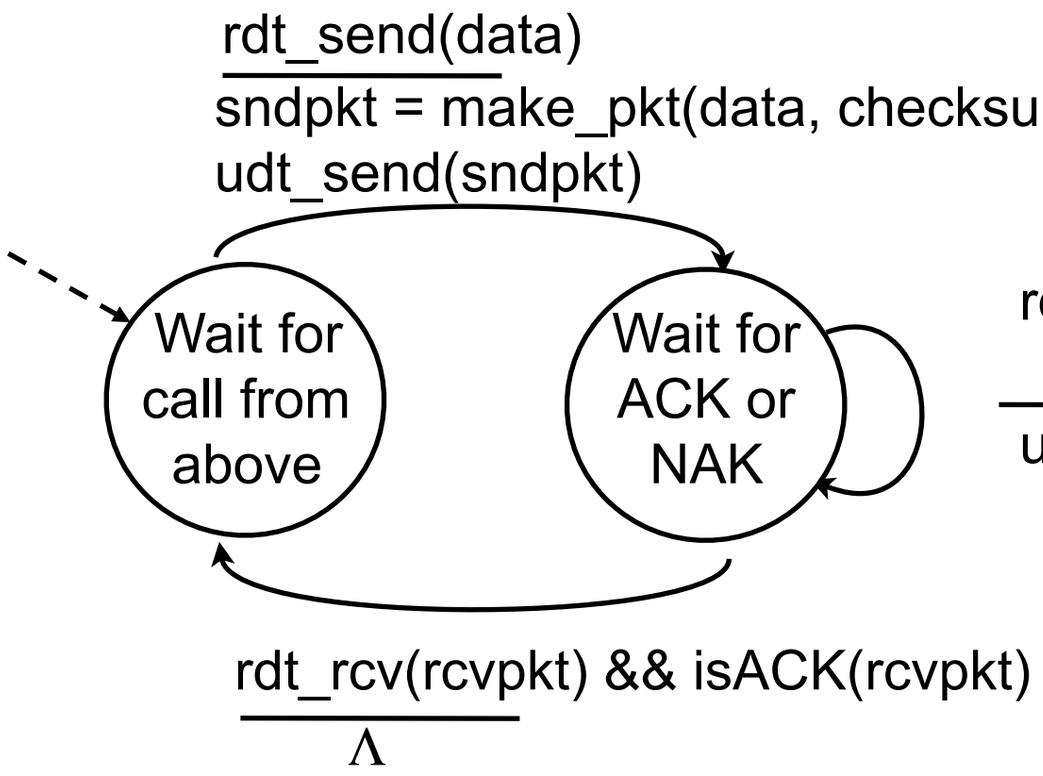
rdt2.0

- **GOAL**: reliable data delivery

- Path between sender and receiver unreliable because it can corrupt packets

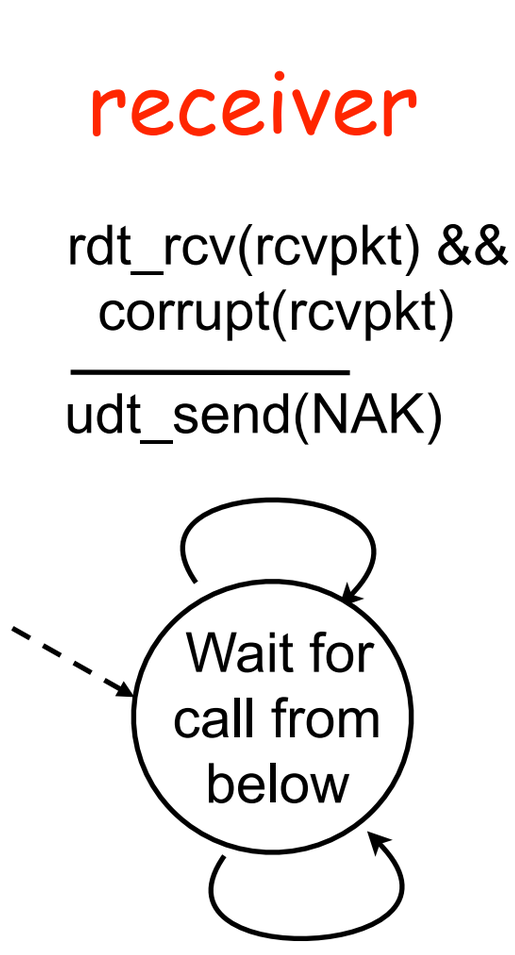
- Introduced three mechanisms to cope with this:
  - checksums
  - ACKs: positive acknowledgments
  - NAKs: negative acknowledgments

rdt2.0: FSM specification



sender

receiver



rdt2.0 has a fatal flaw!

rdt2.0 has a fatal flaw!

What happens if ACK/
NAK 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

rdt2.0 has a fatal flaw!

What happens if ACK/ NAK corrupted?

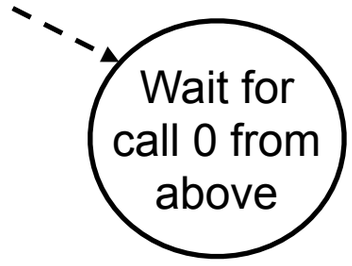
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

Handling duplicates:

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

rdt2.1: sender, handles garbled ACK/NAKs

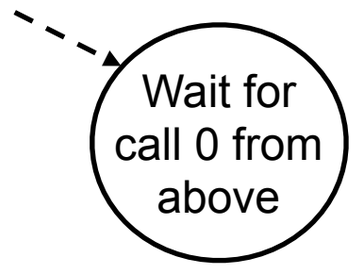
rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: sender, handles garbled ACK/NAKs

rdt_send(data)

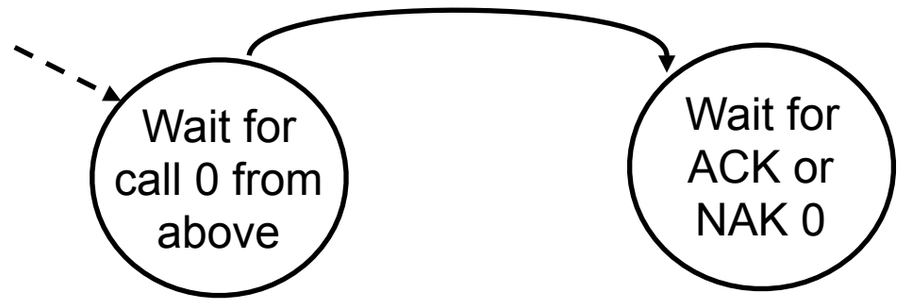
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)



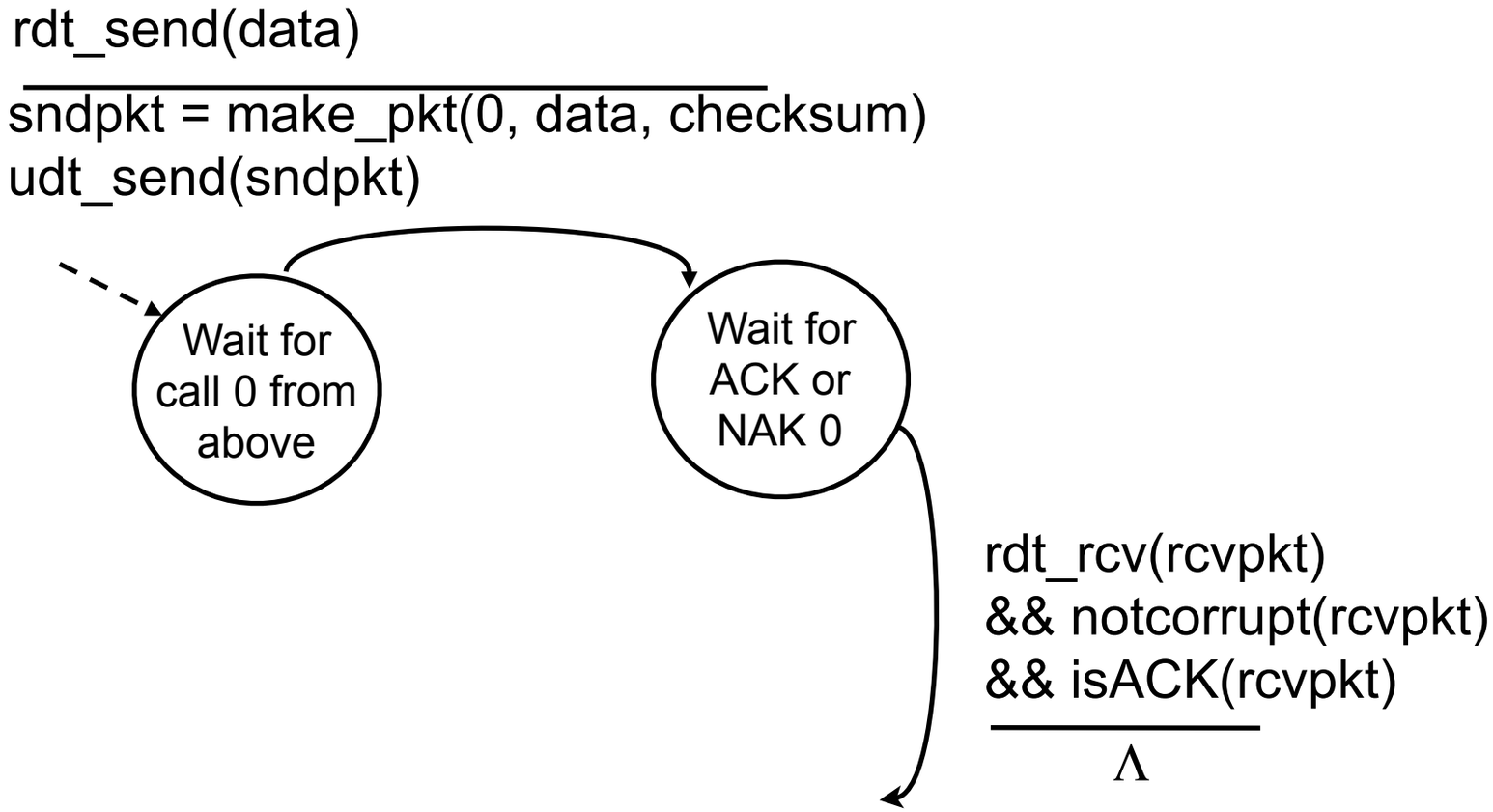
rdt2.1: sender, handles garbled ACK/NAKs

rdt_send(data)

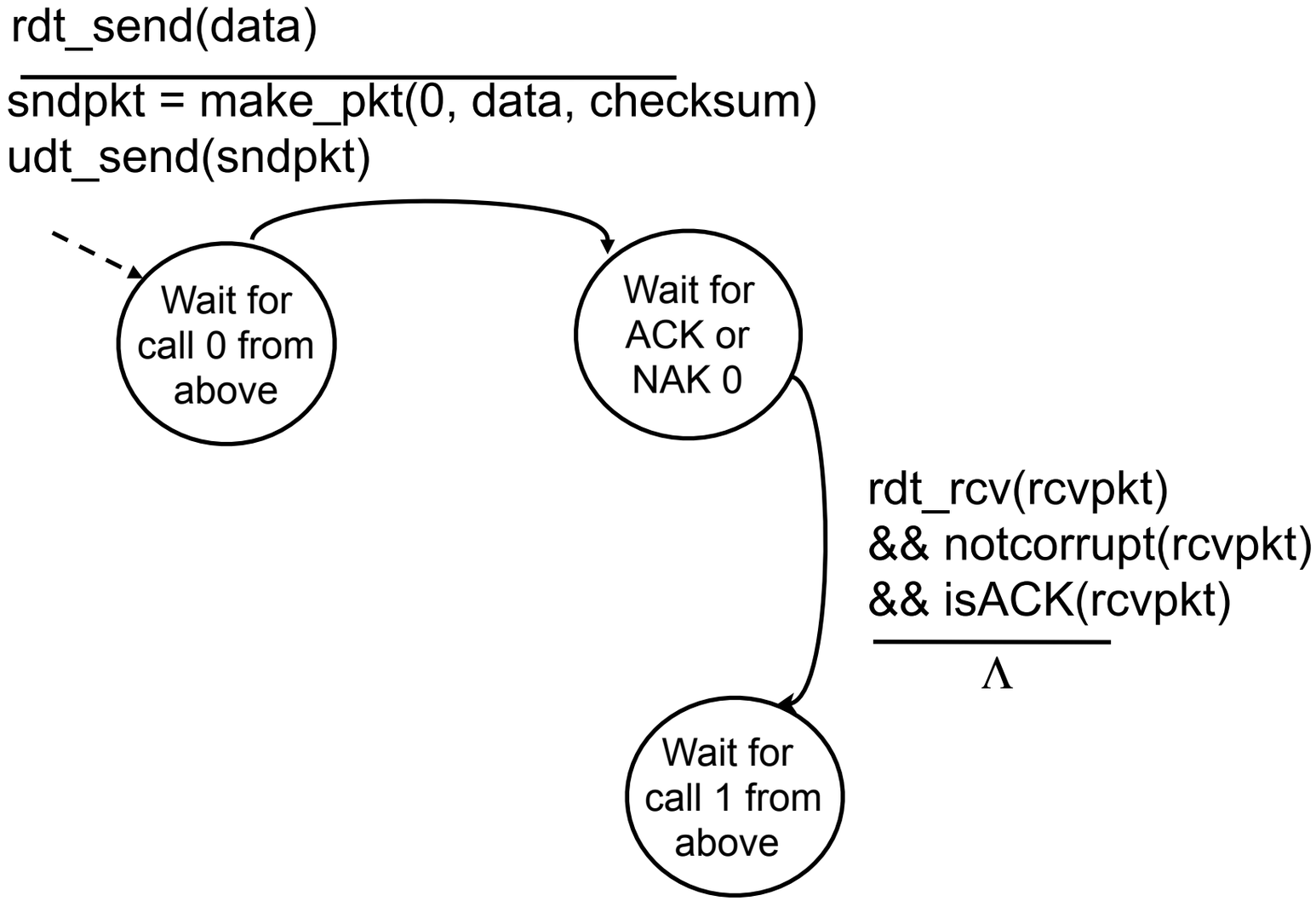
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)



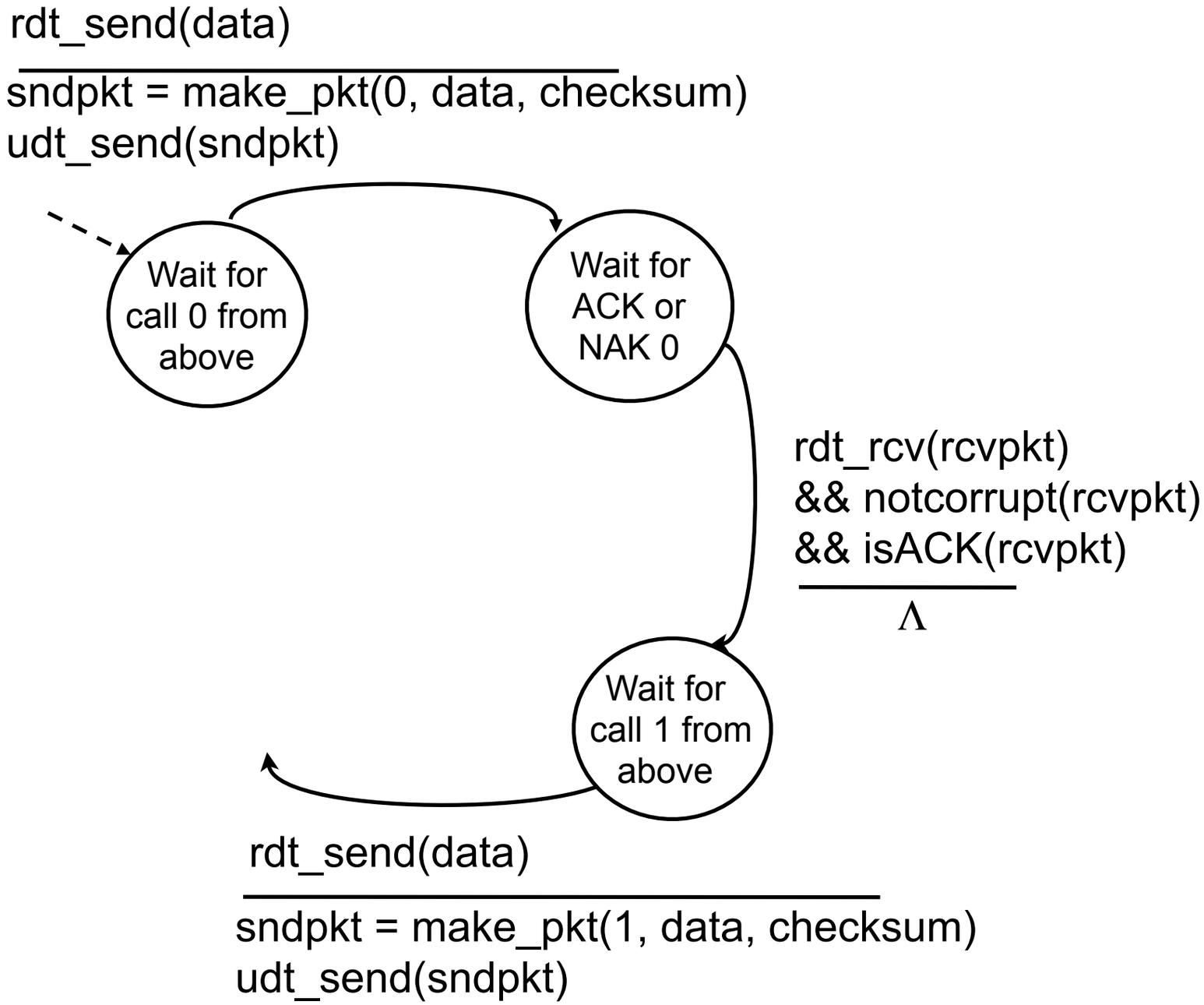
rdt2.1: sender, handles garbled ACK/NAKs



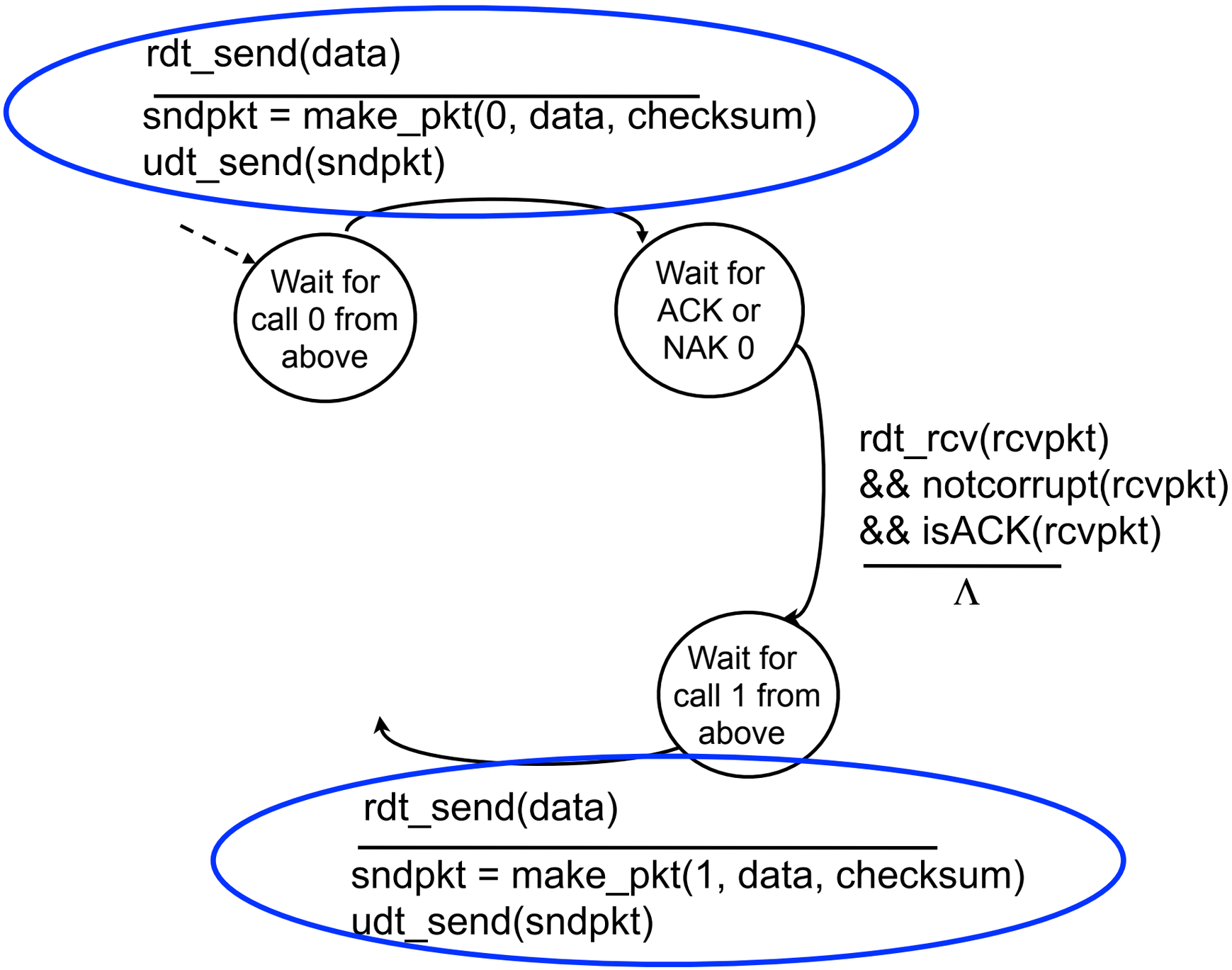
rdt2.1: sender, handles garbled ACK/NAKs



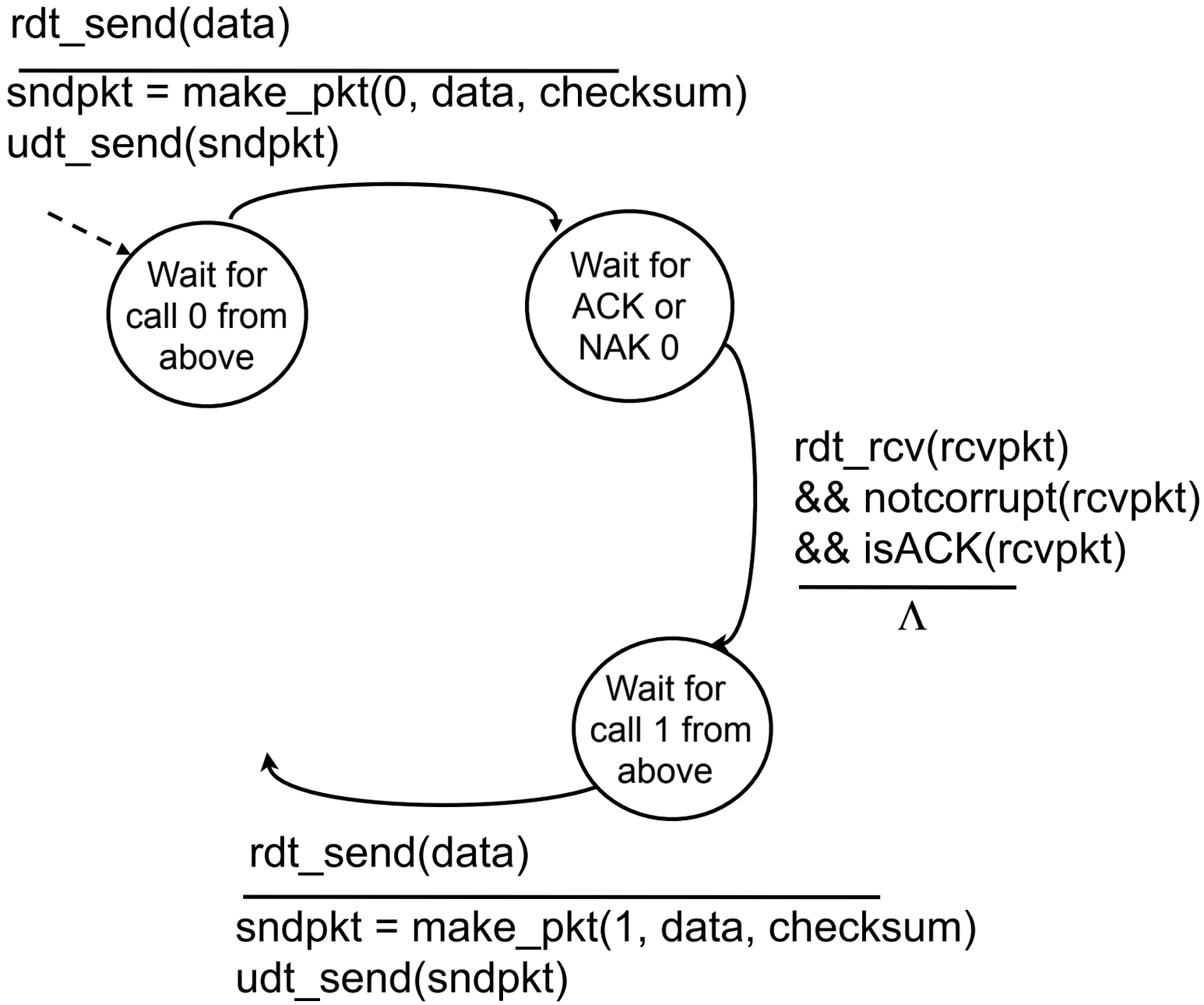
rdt2.1: sender, handles garbled ACK/NAKs



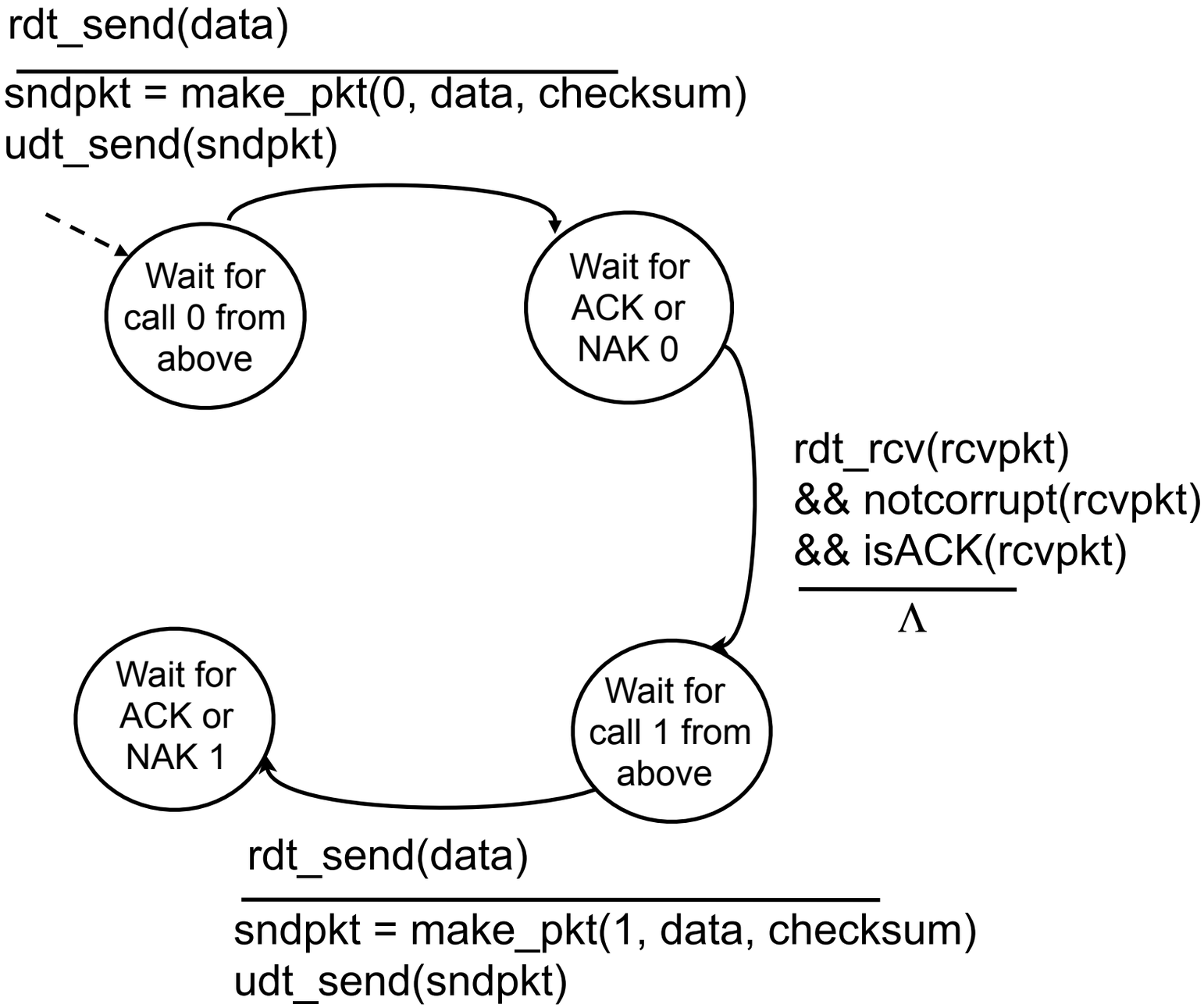
rdt2.1: sender, handles garbled ACK/NAKs



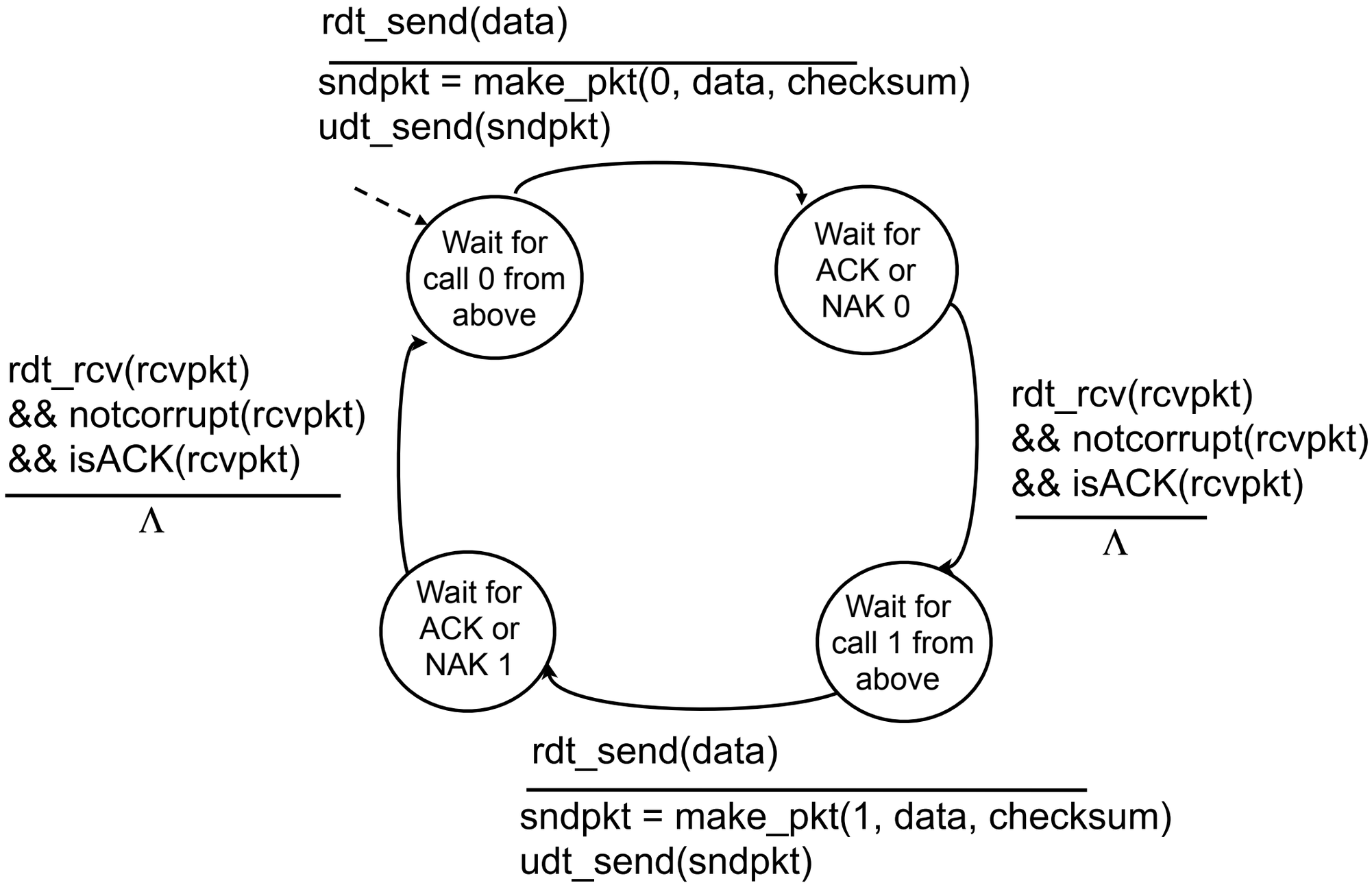
rdt2.1: sender, handles garbled ACK/NAKs



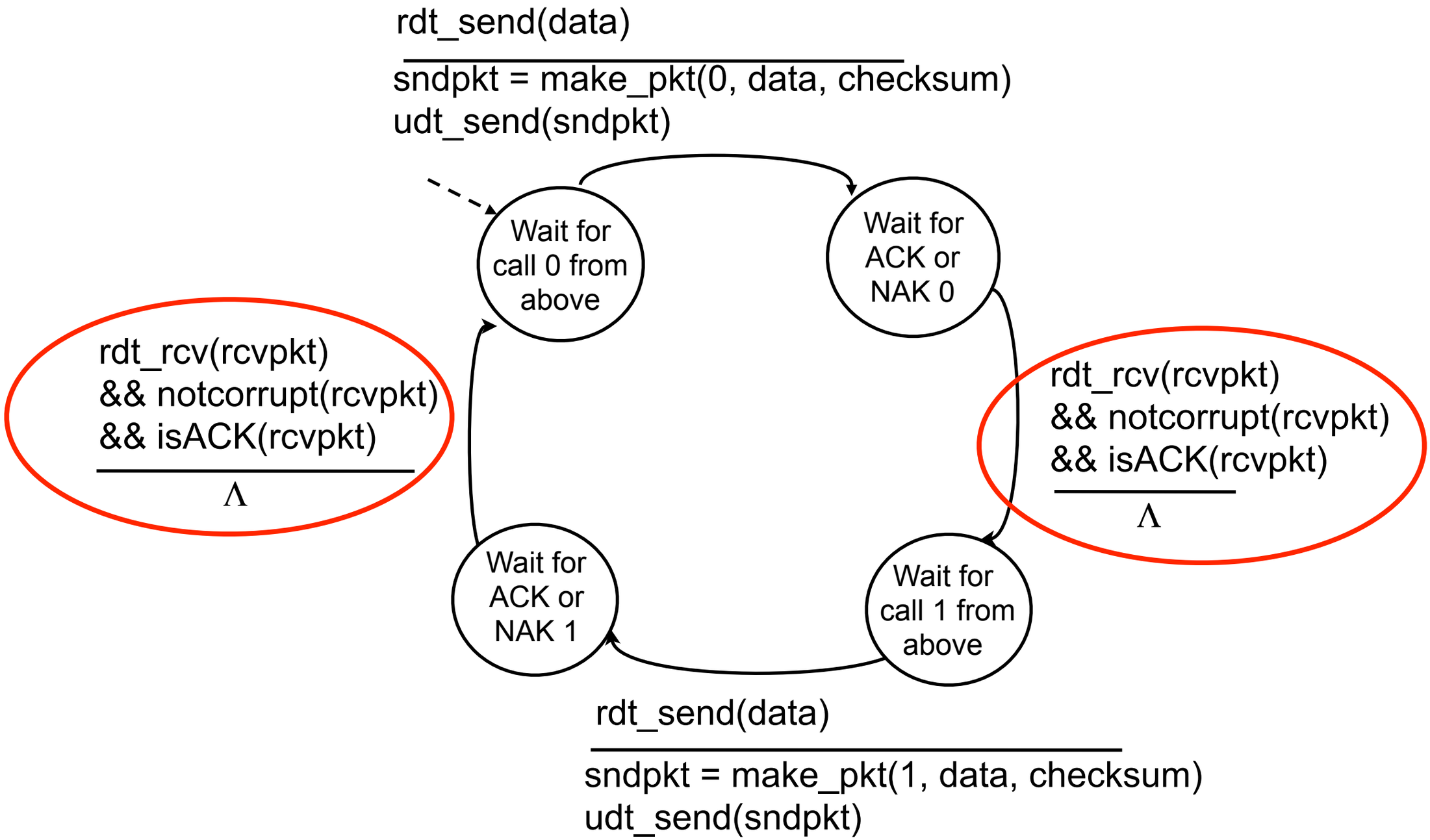
rdt2.1: sender, handles garbled ACK/NAKs



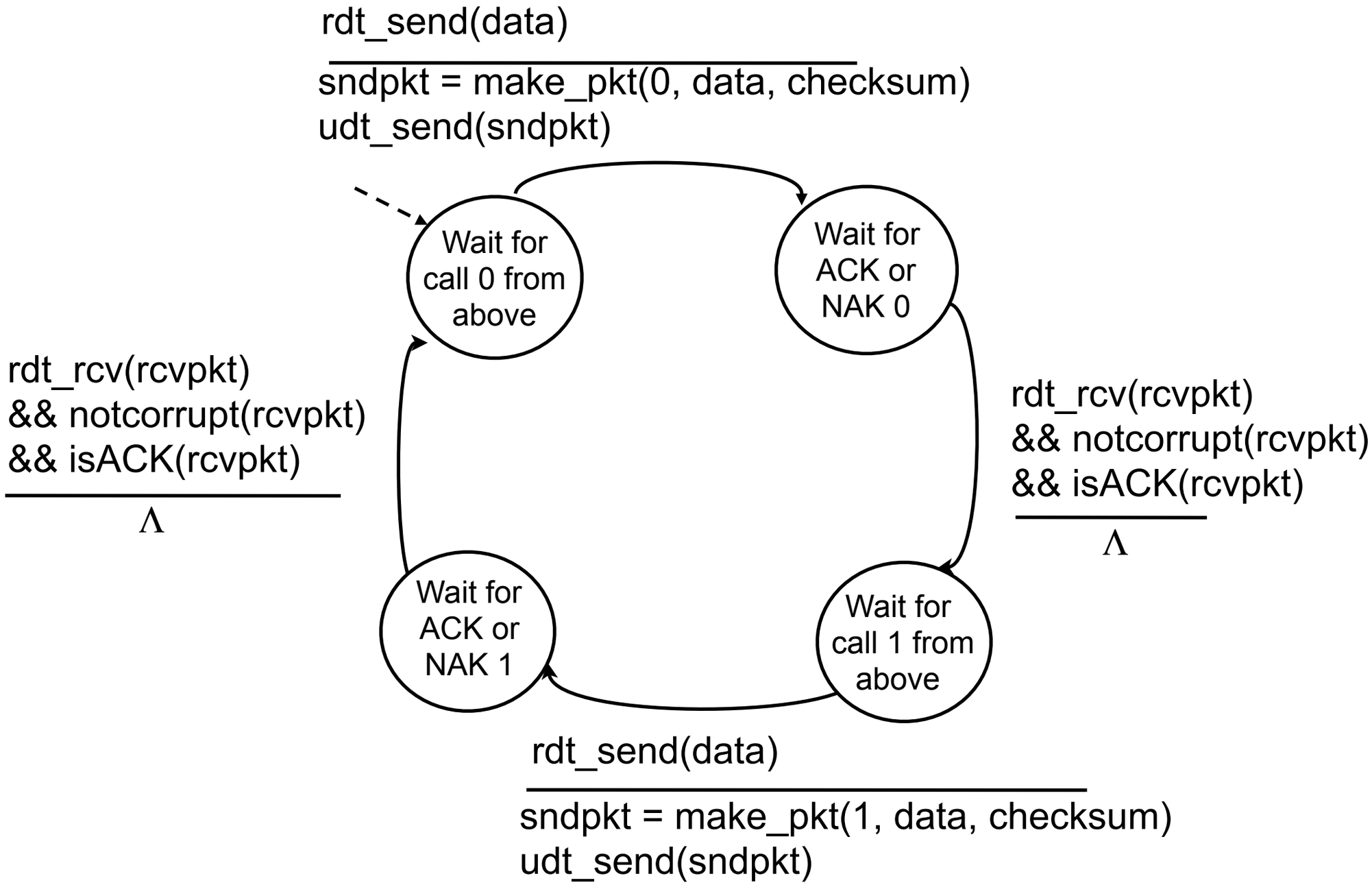
rdt2.1: sender, handles garbled ACK/NAKs



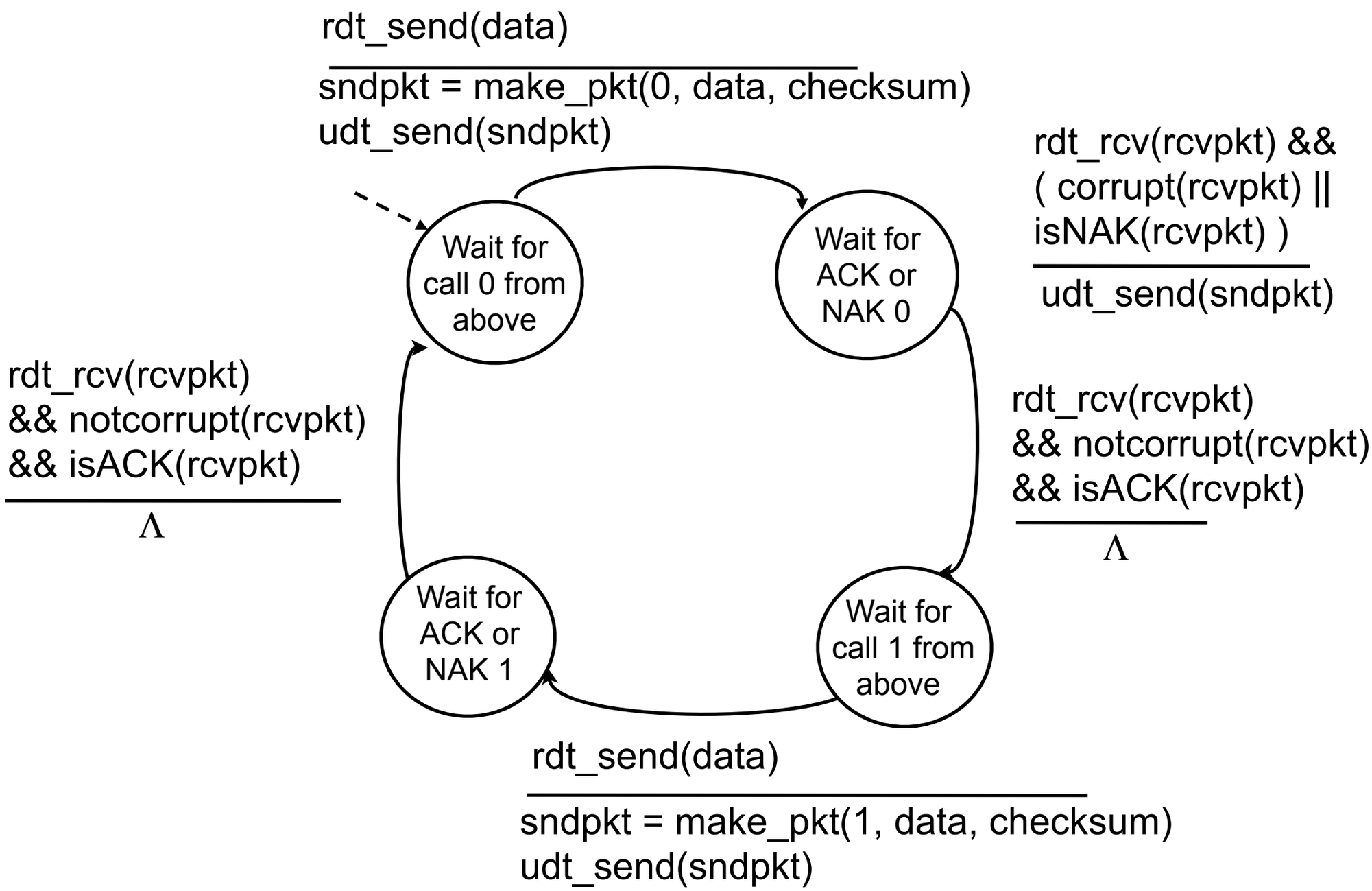
rdt2.1: sender, handles garbled ACK/NAKs



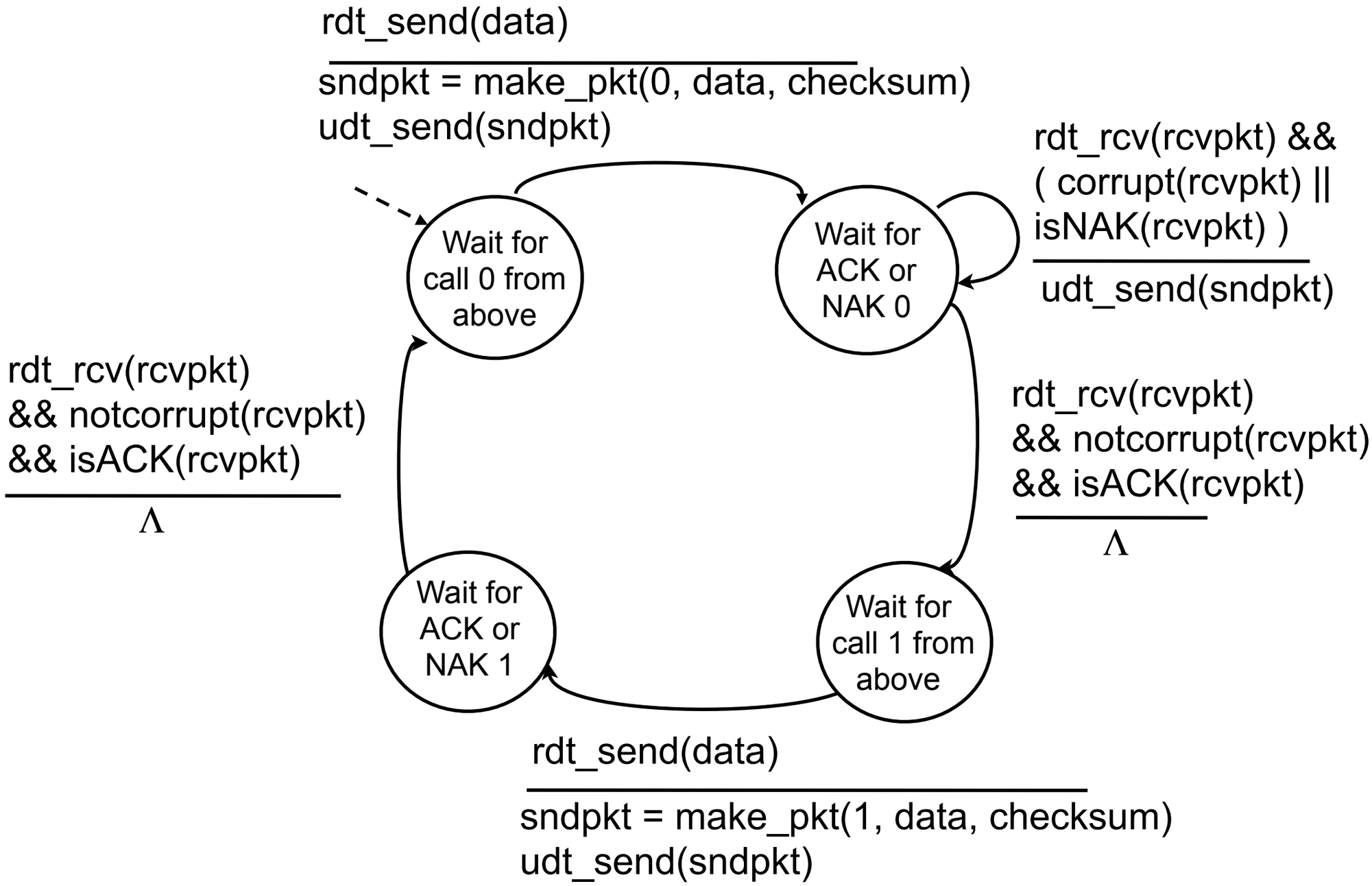
rdt2.1: sender, handles garbled ACK/NAKs



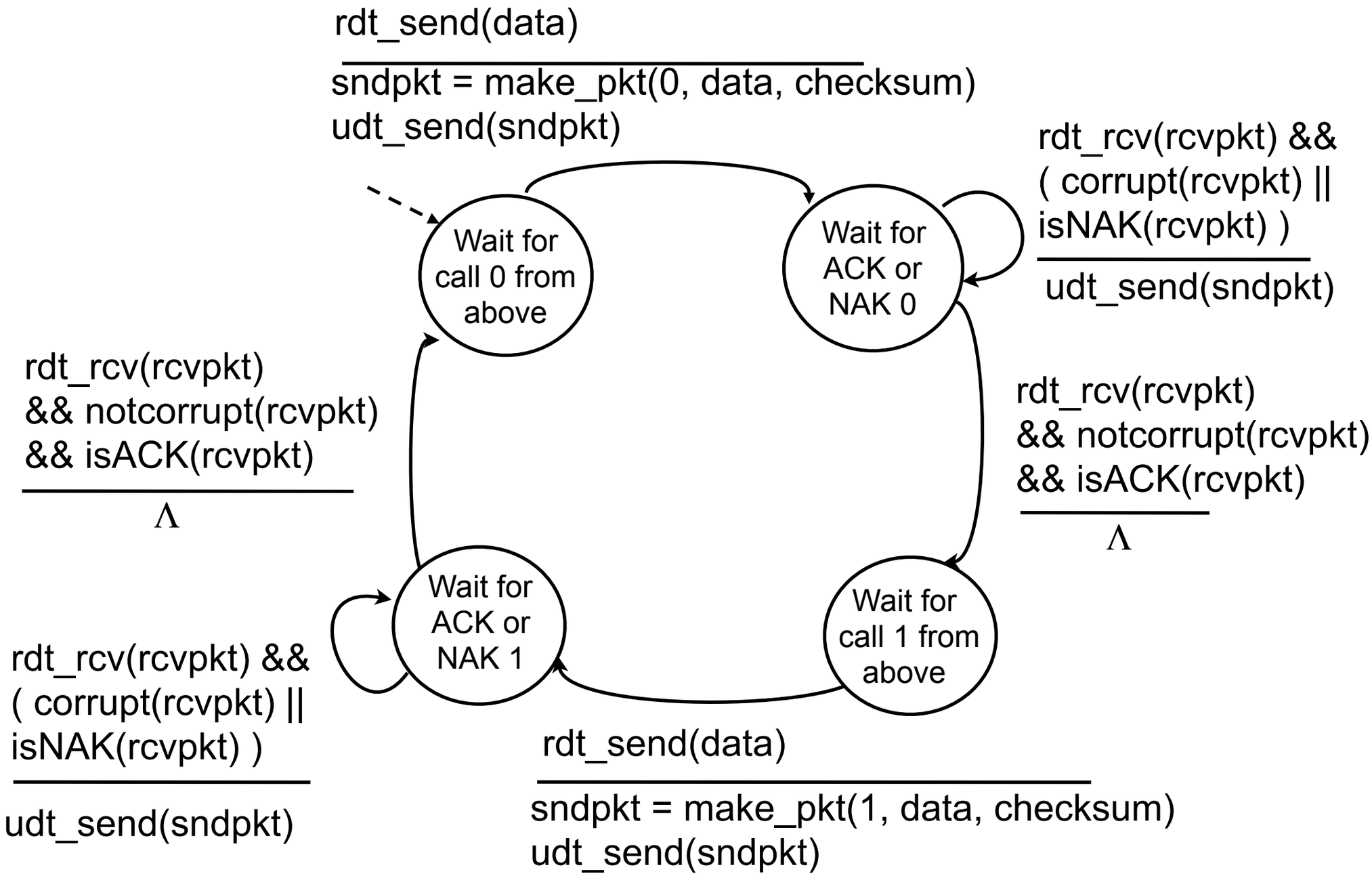
rdt2.1: sender, handles garbled ACK/NAKs



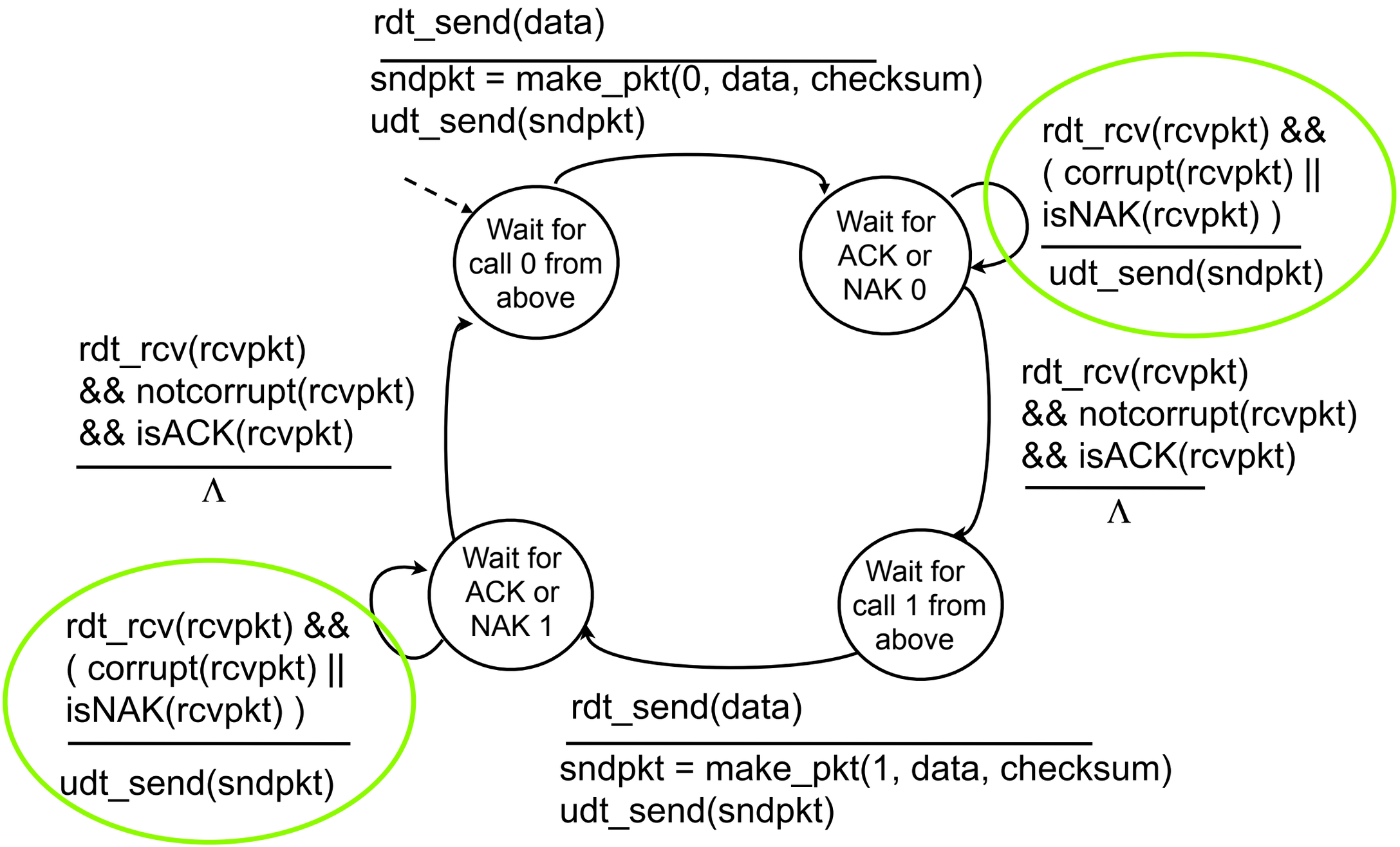
rdt2.1: sender, handles garbled ACK/NAKs



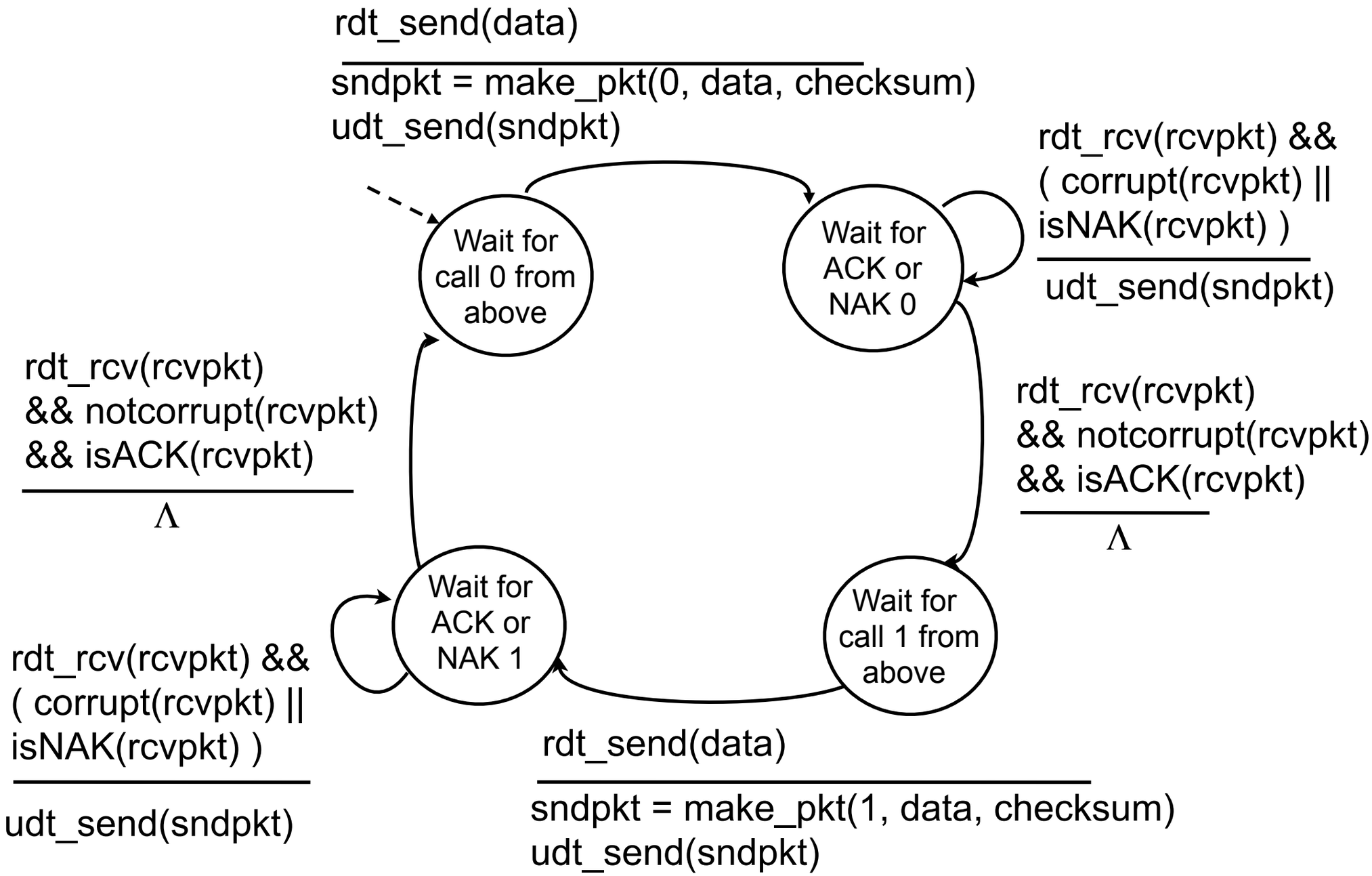
rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: sender, handles garbled ACK/NAKs

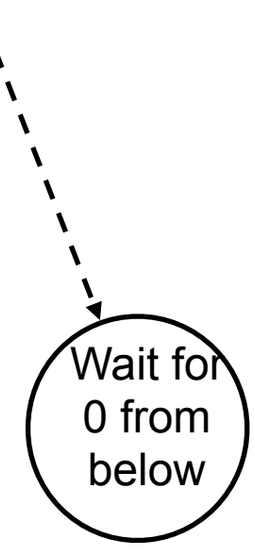


rdt2.1: sender, handles garbled ACK/NAKs

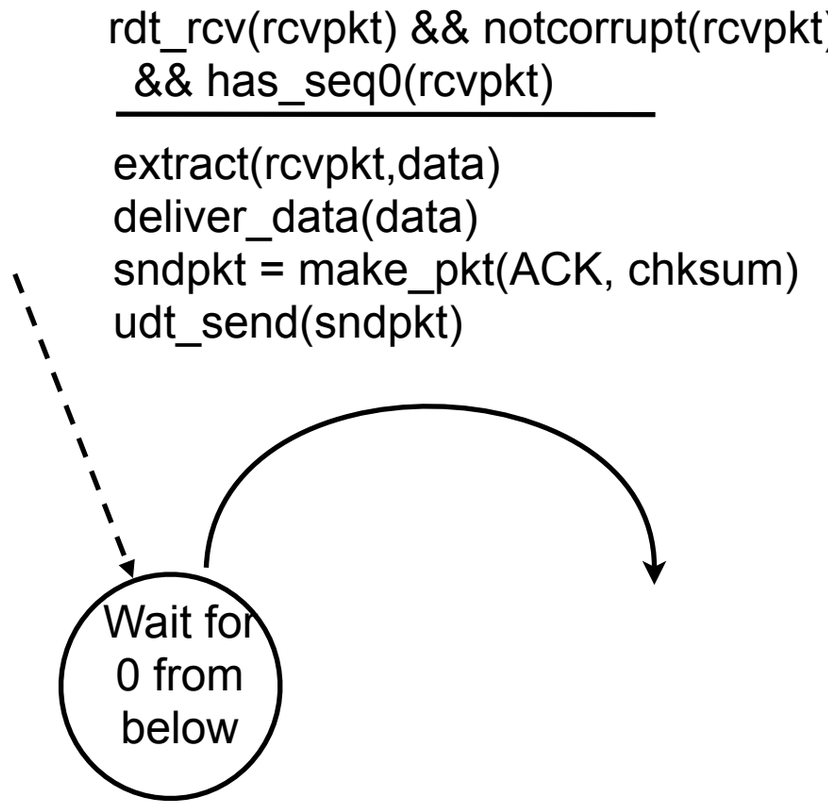


rdt2.1: receiver, handles garbled ACK/NAKs

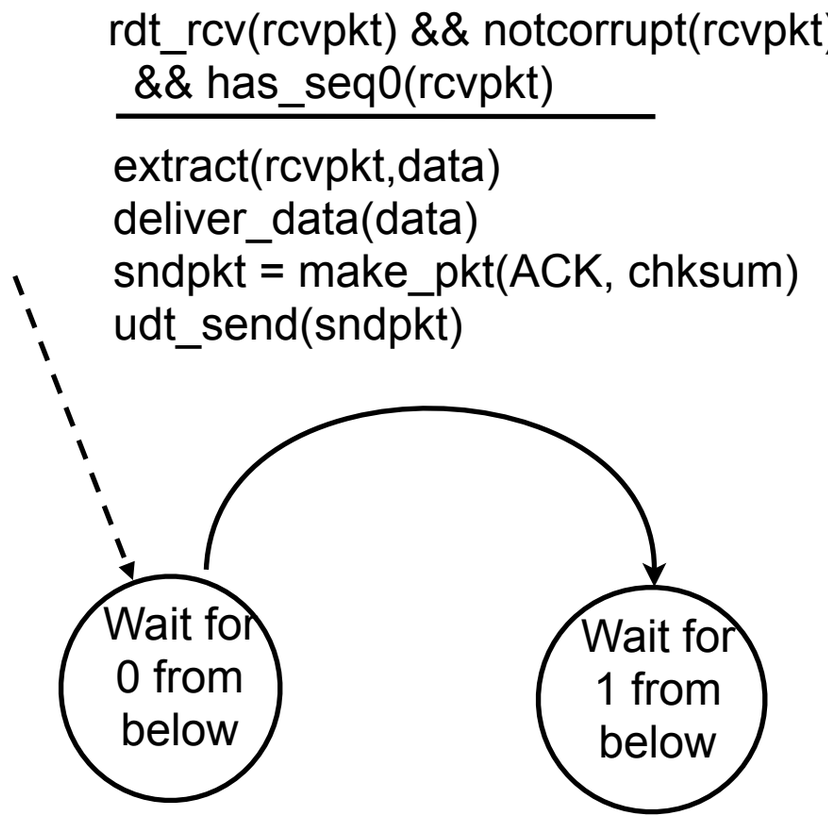
rdt2.1: receiver, handles garbled ACK/NAKs



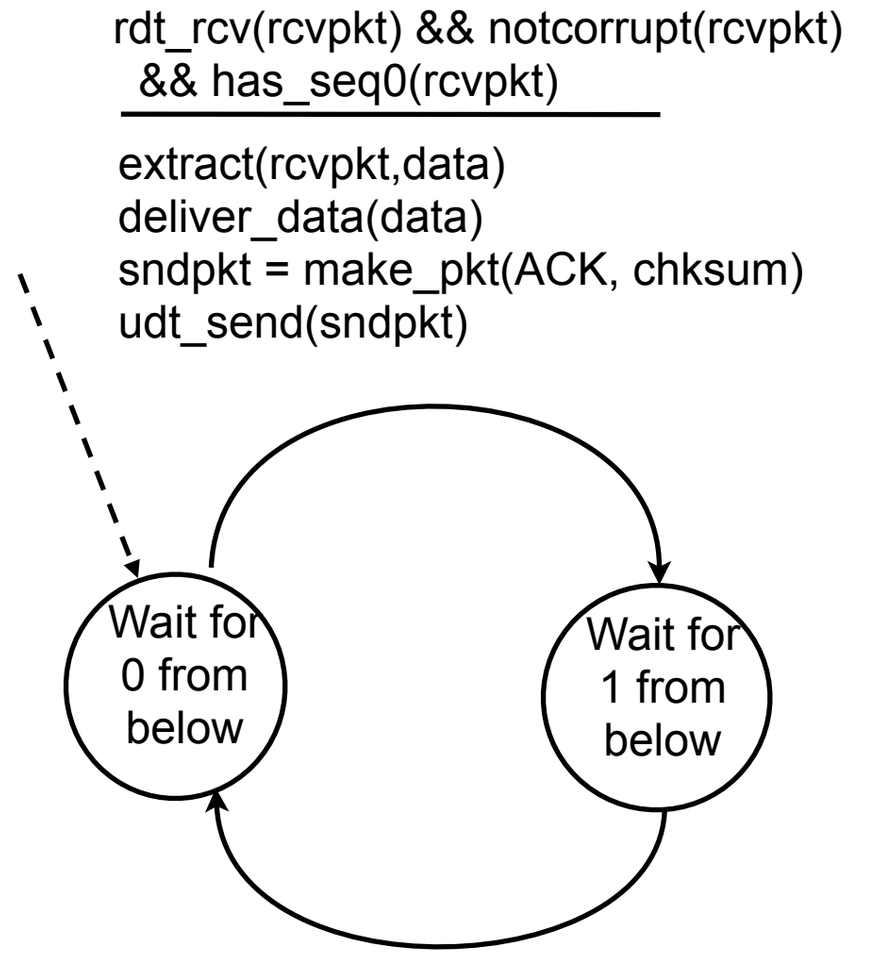
rdt2.1: receiver, handles garbled ACK/NAKs



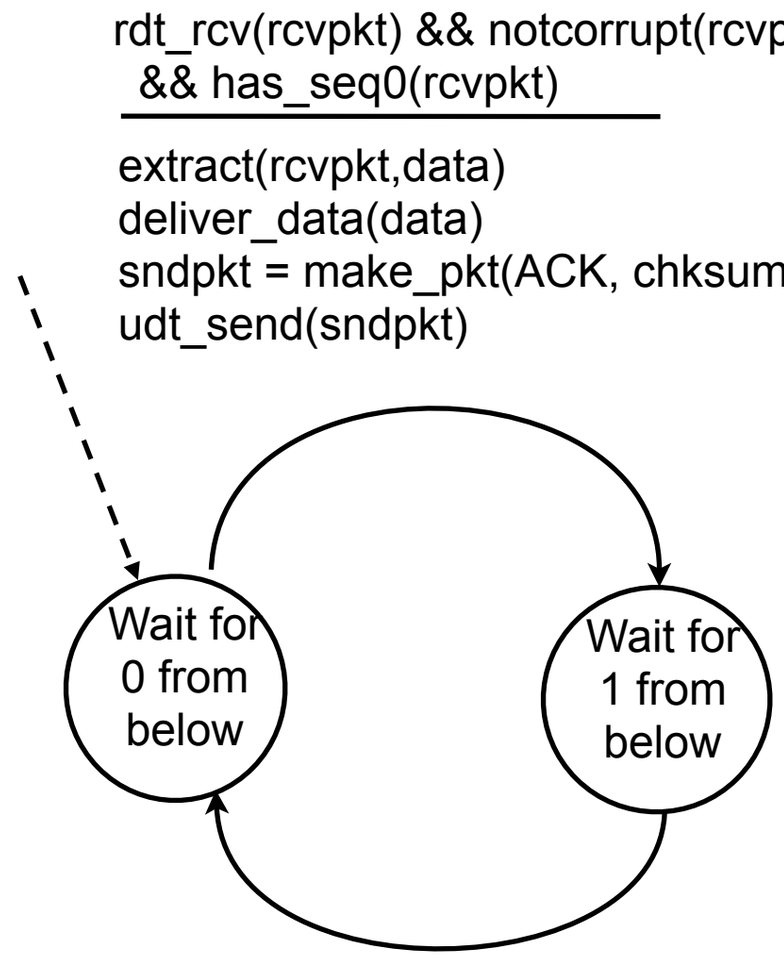
rdt2.1: receiver, handles garbled ACK/NAKs



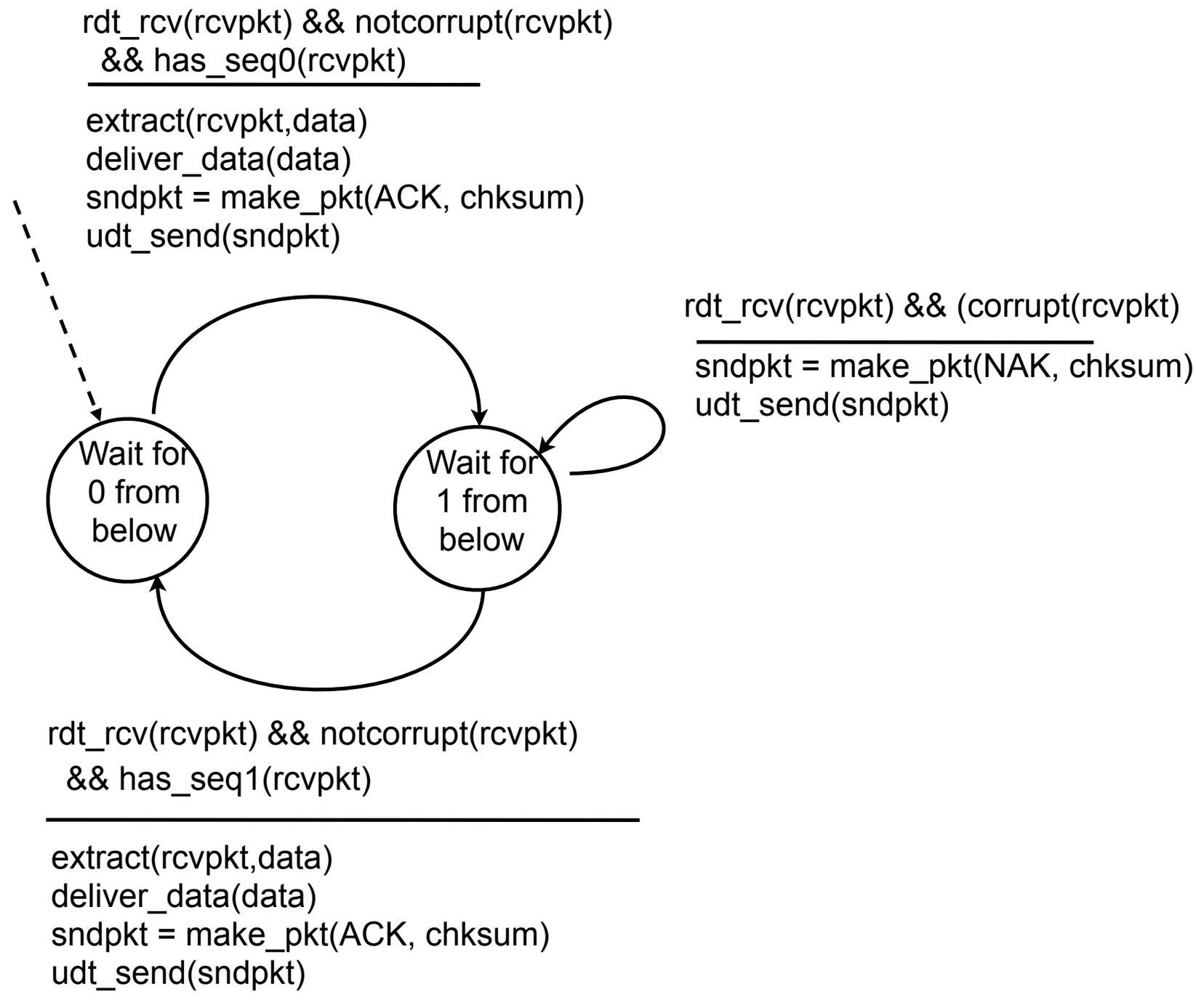
rdt2.1: receiver, handles garbled ACK/NAKs



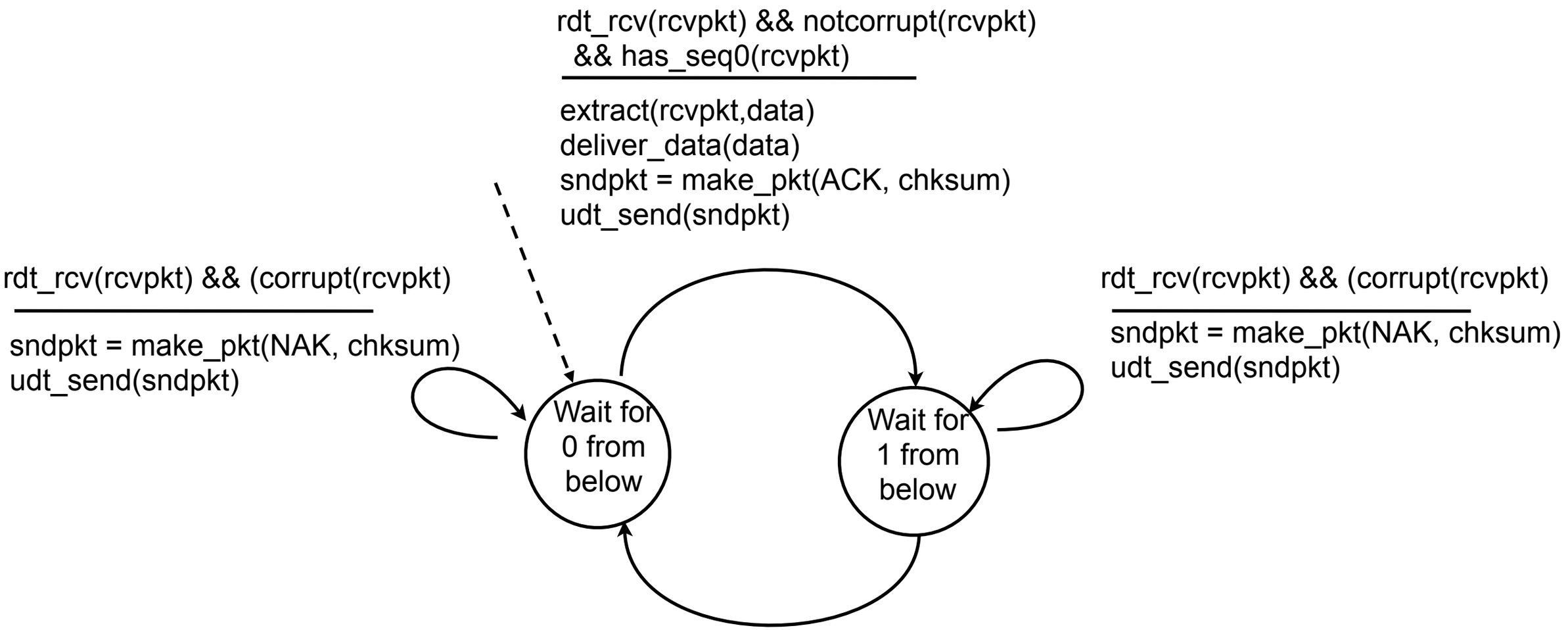
rdt2.1: receiver, handles garbled ACK/NAKs



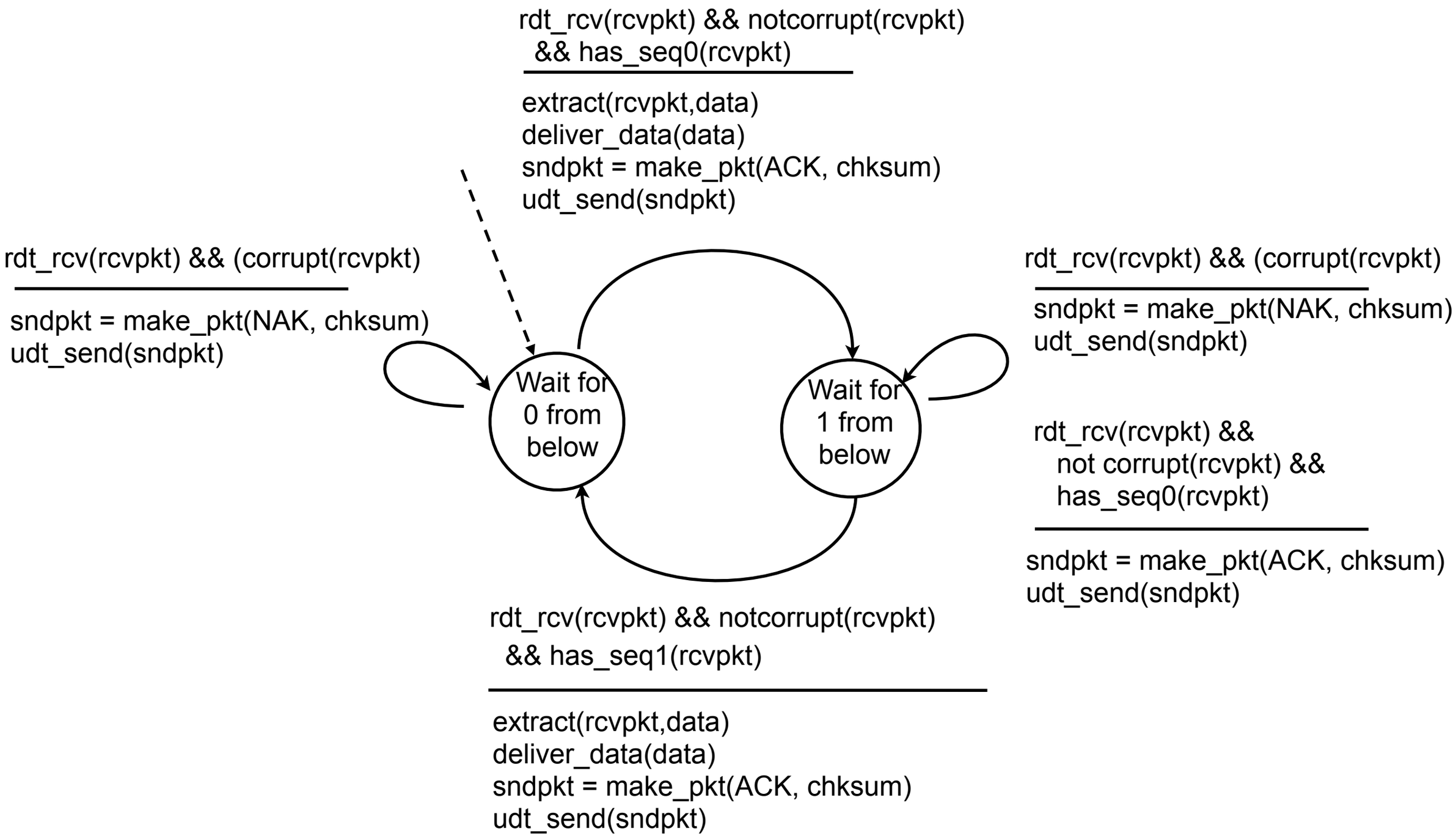
rdt2.1: receiver, handles garbled ACK/NAKs



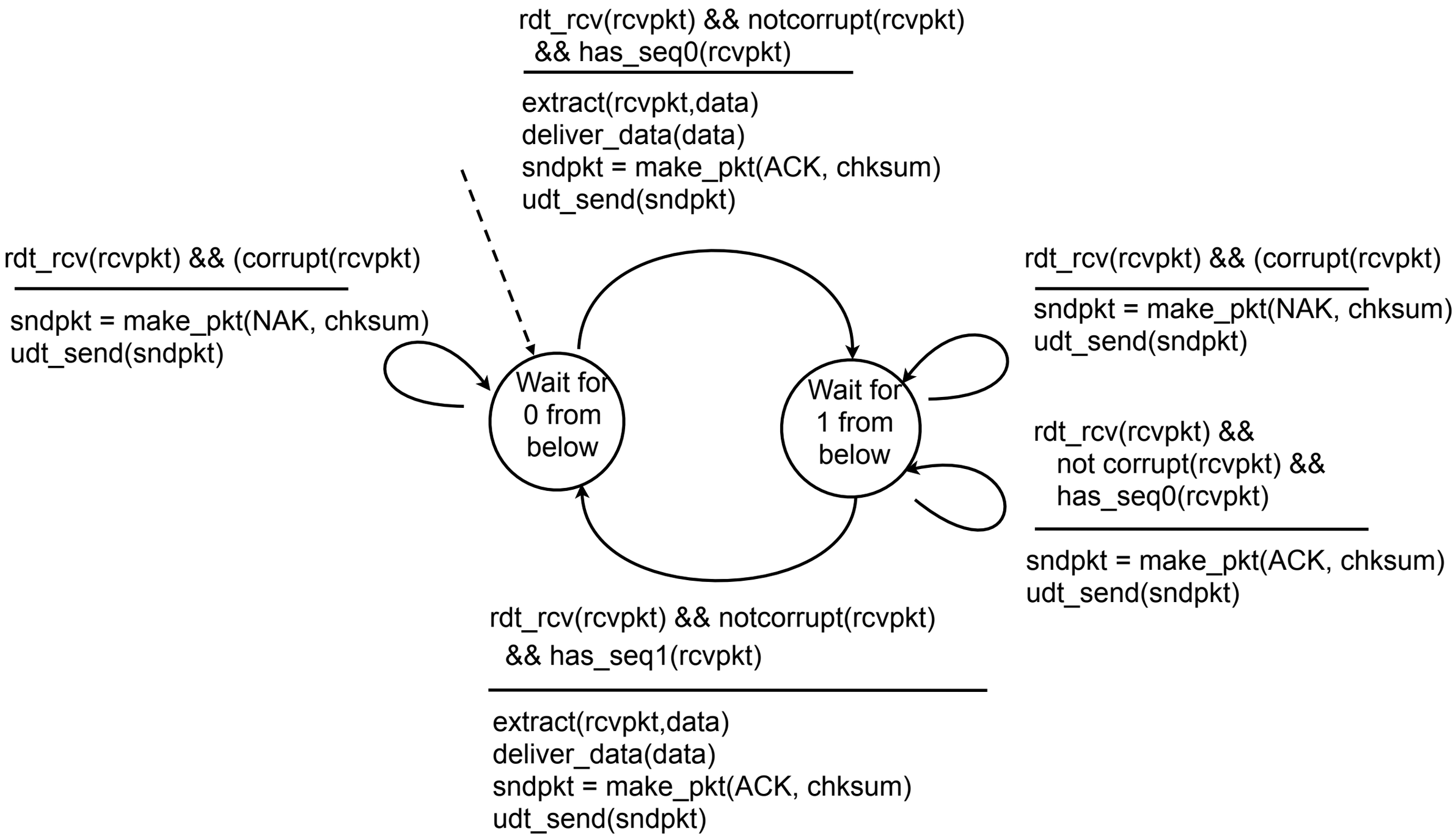
rdt2.1: receiver, handles garbled ACK/NAKs



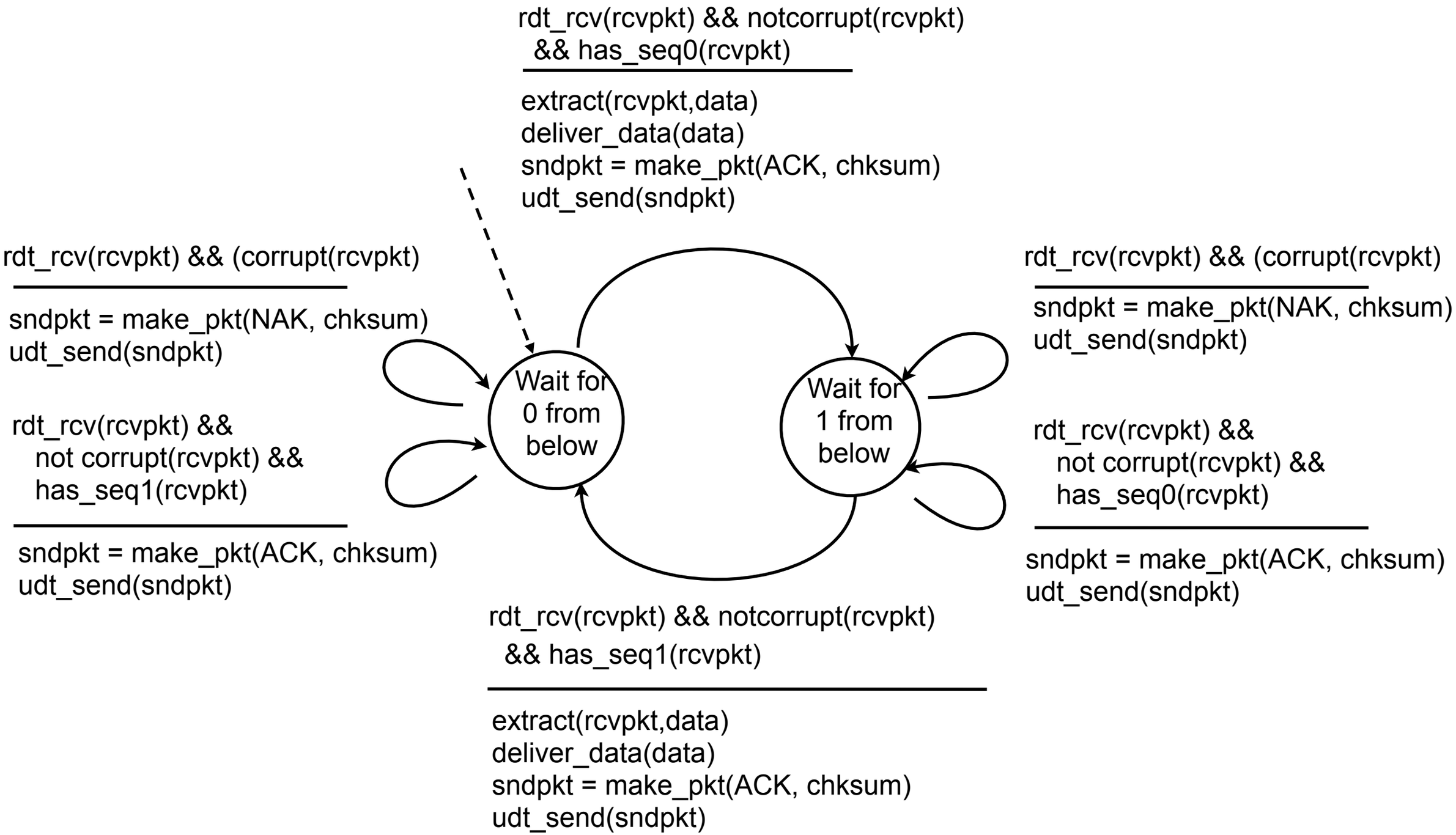
rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: receiver, 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. #

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. #

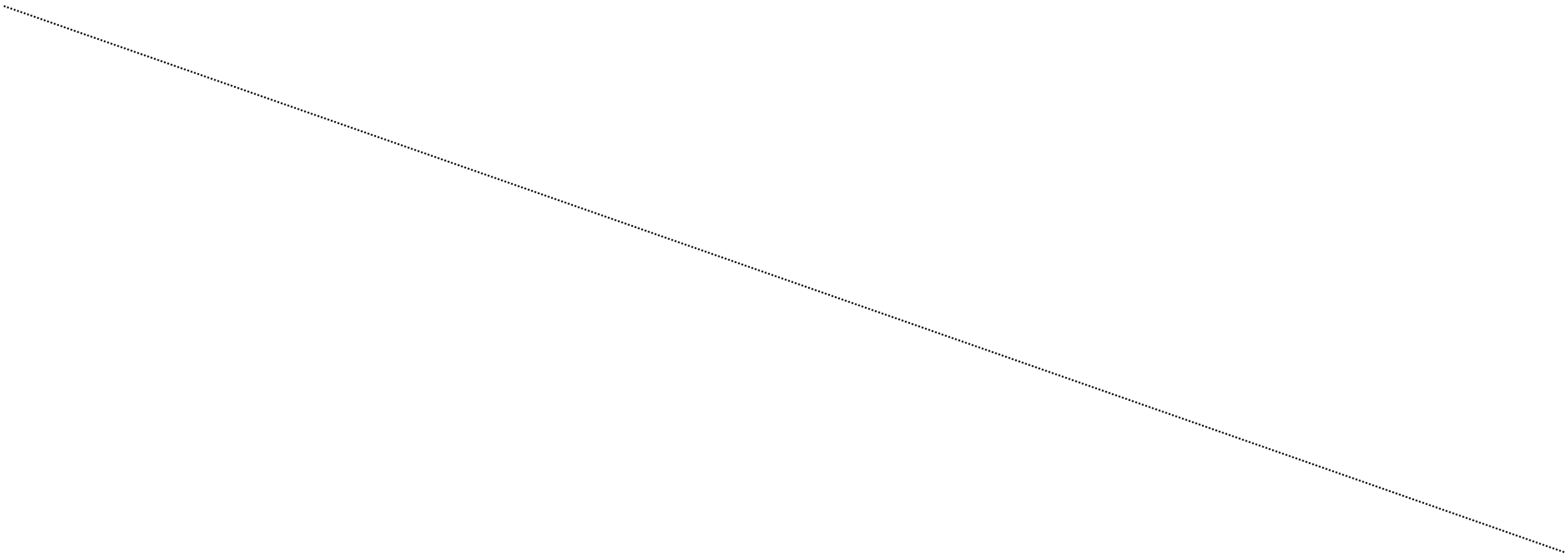
Receiver:

- 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

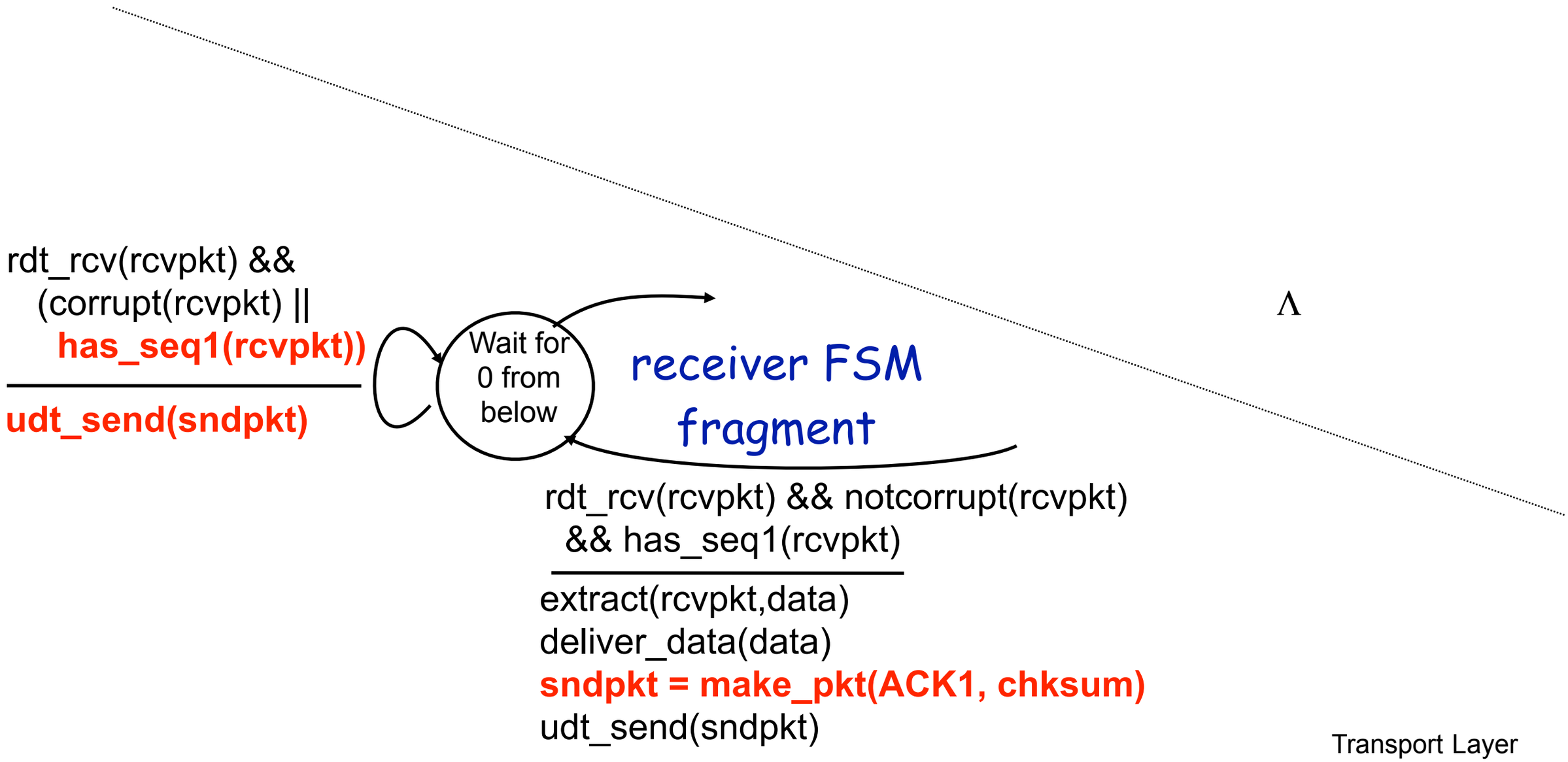
rdt2.2: a NAK-free protocol

- 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

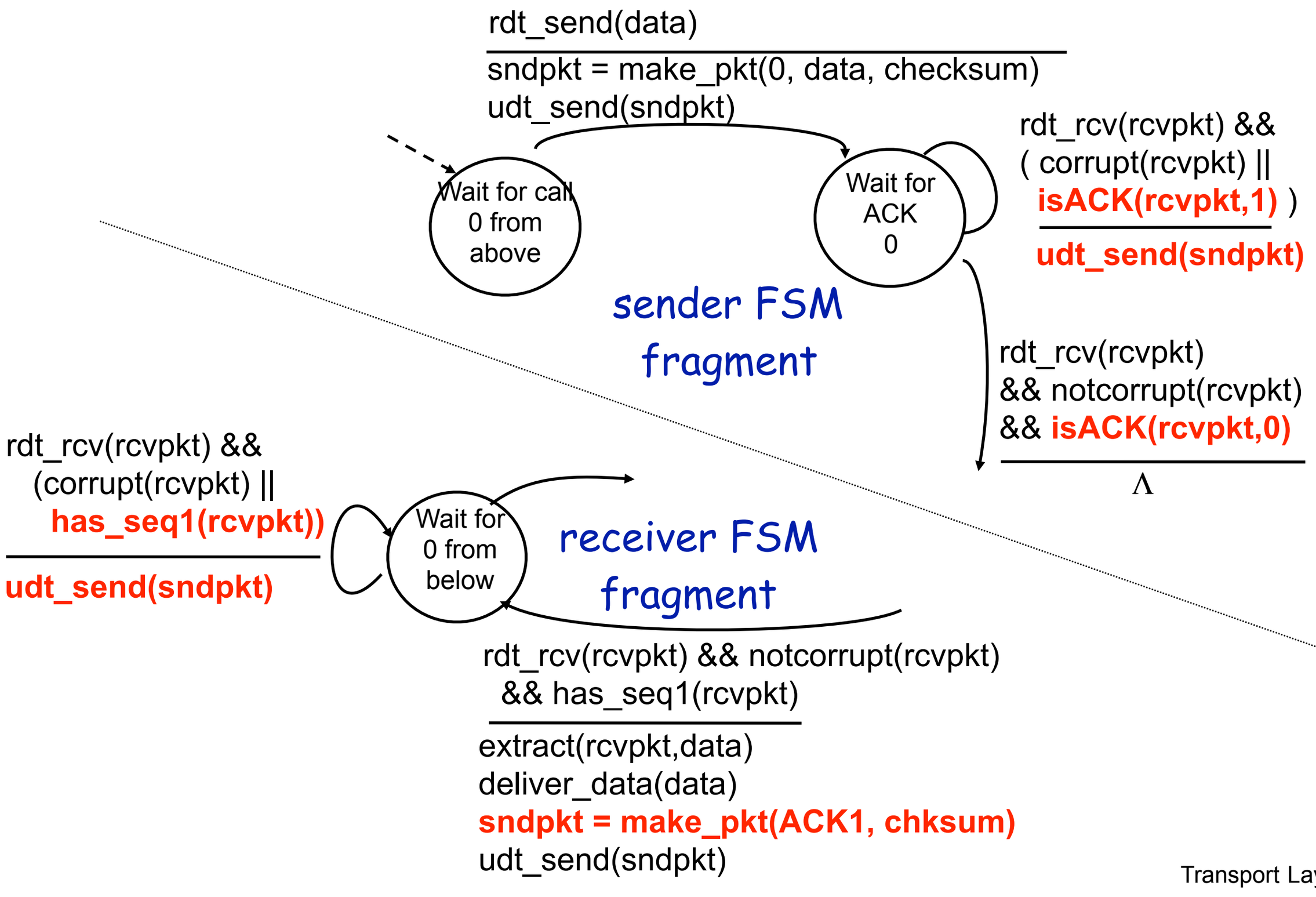
rdt2.2: sender, receiver fragments



rdt2.2: sender, receiver fragments



rdt2.2: sender, receiver fragments



rdt3.0: channels with errors and loss

New context: underlying channel can also lose packets (data or ACKs)

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

rdt3.0: channels with errors and loss

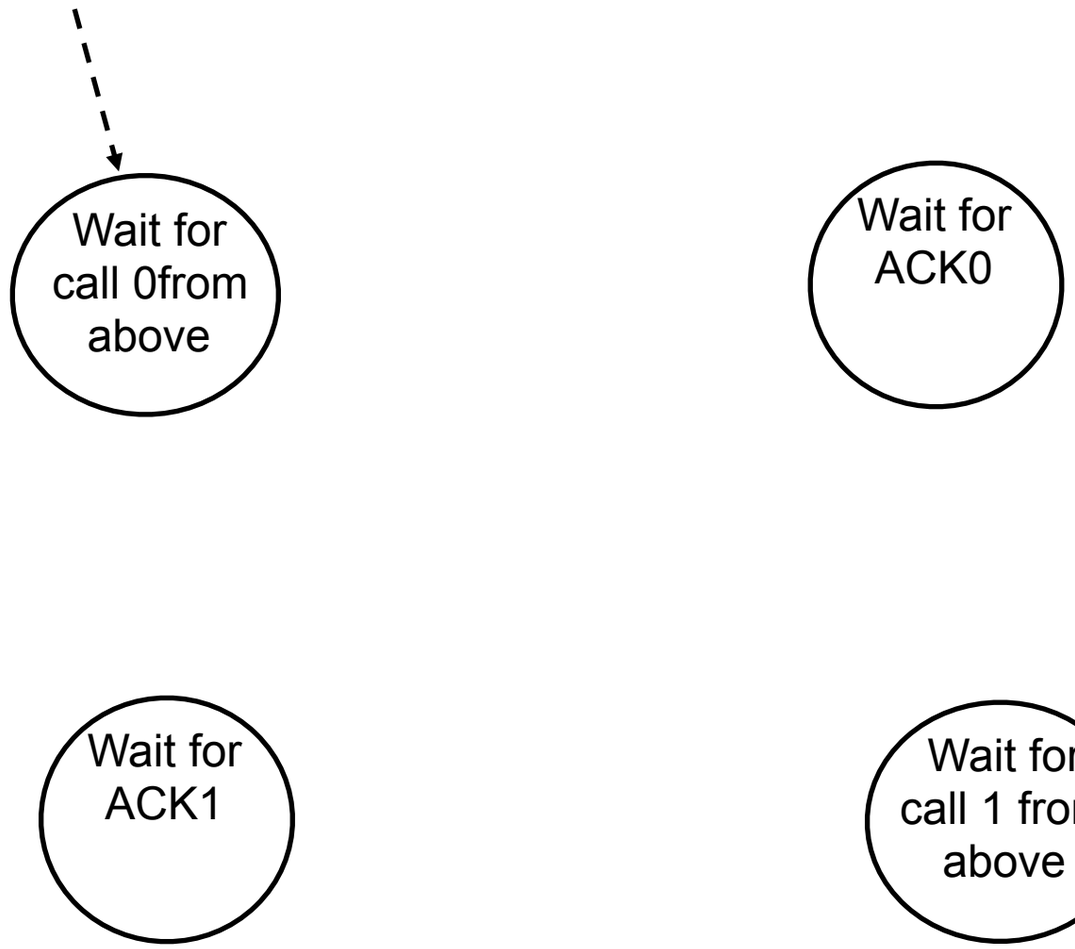
New context: underlying channel can also lose packets (data or ACKs)

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

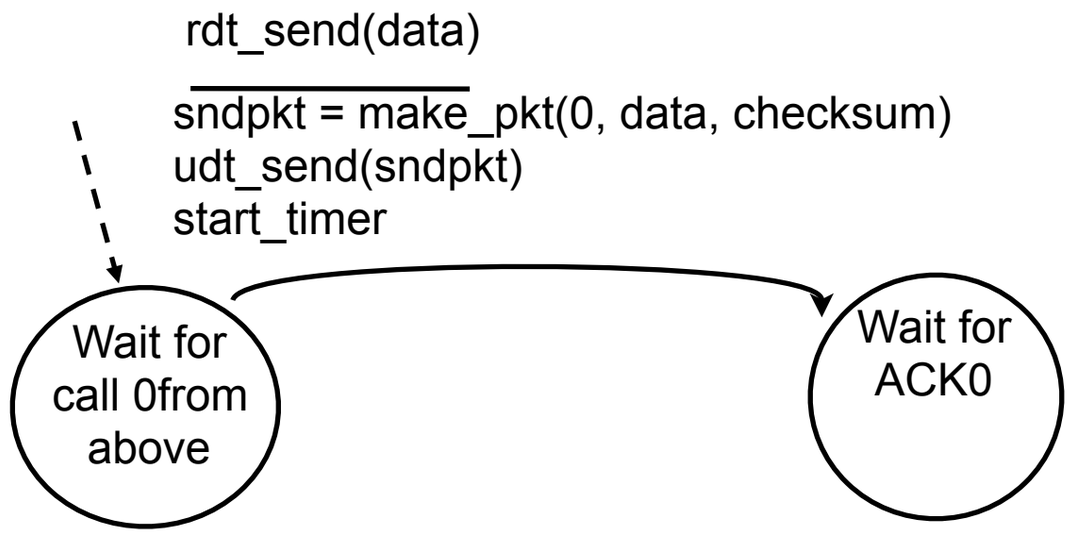
Approach: sender waits "reasonable" amount of time for ACK

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

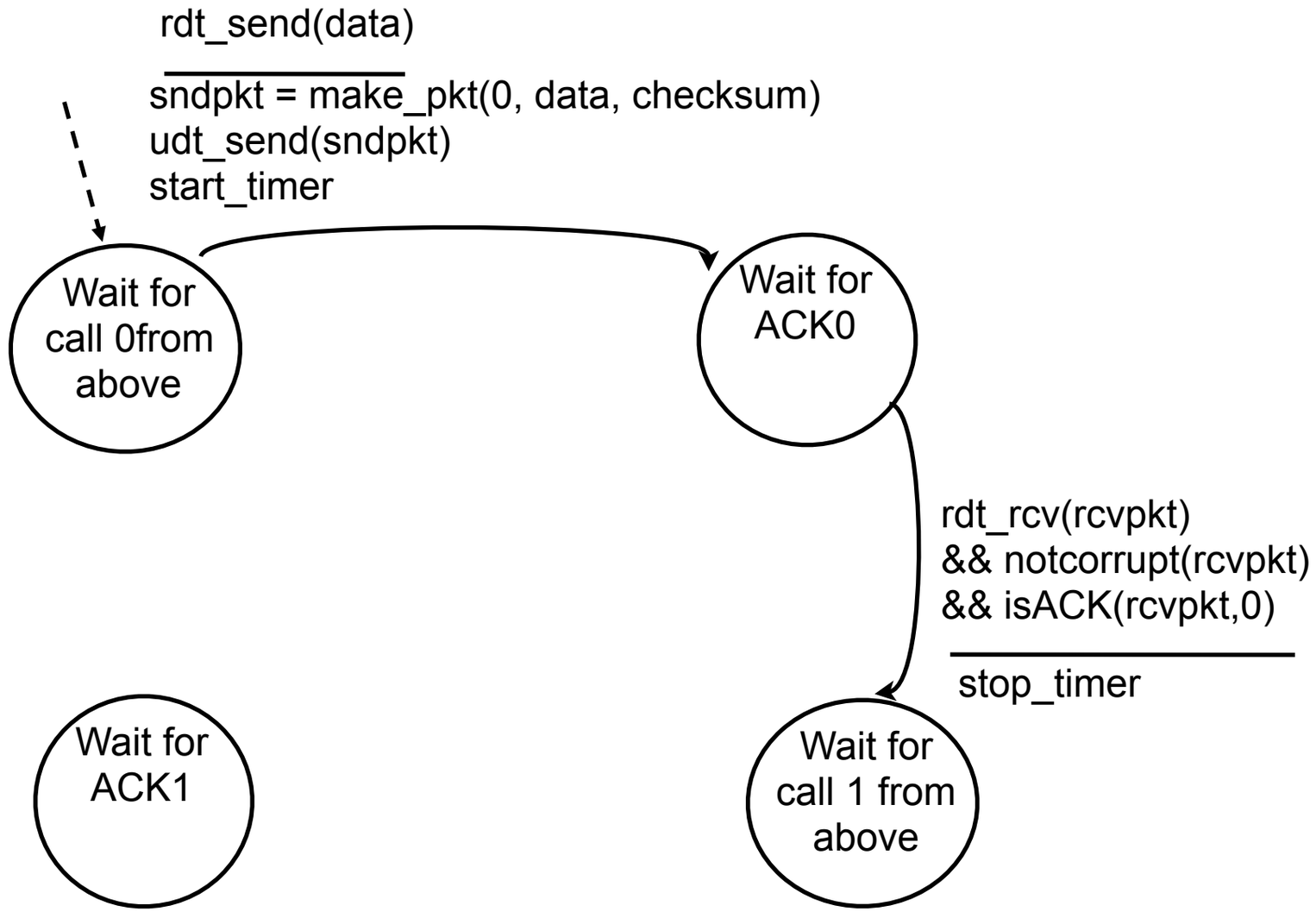
rdt3.0 sender



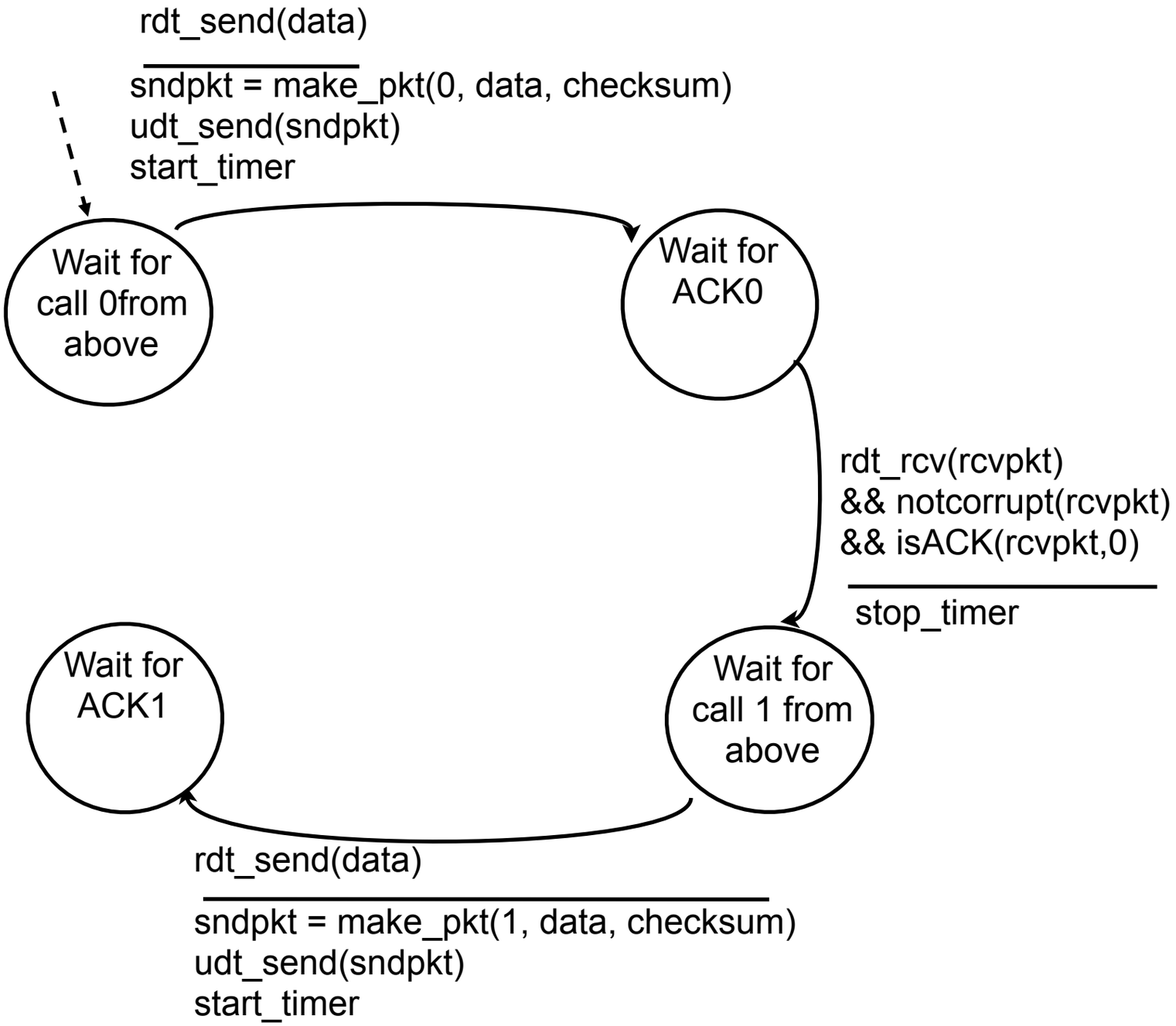
rdt3.0 sender



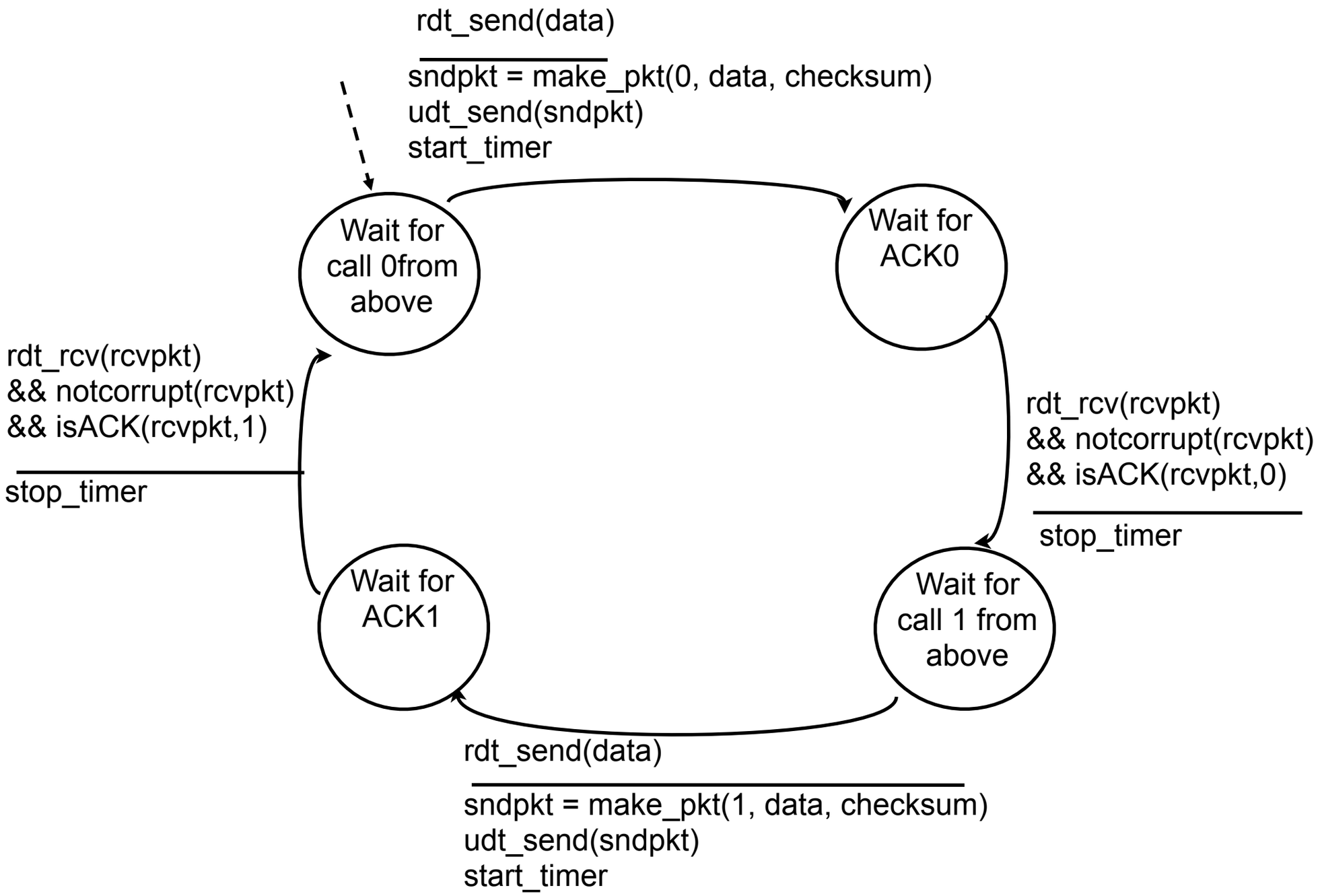
rdt3.0 sender



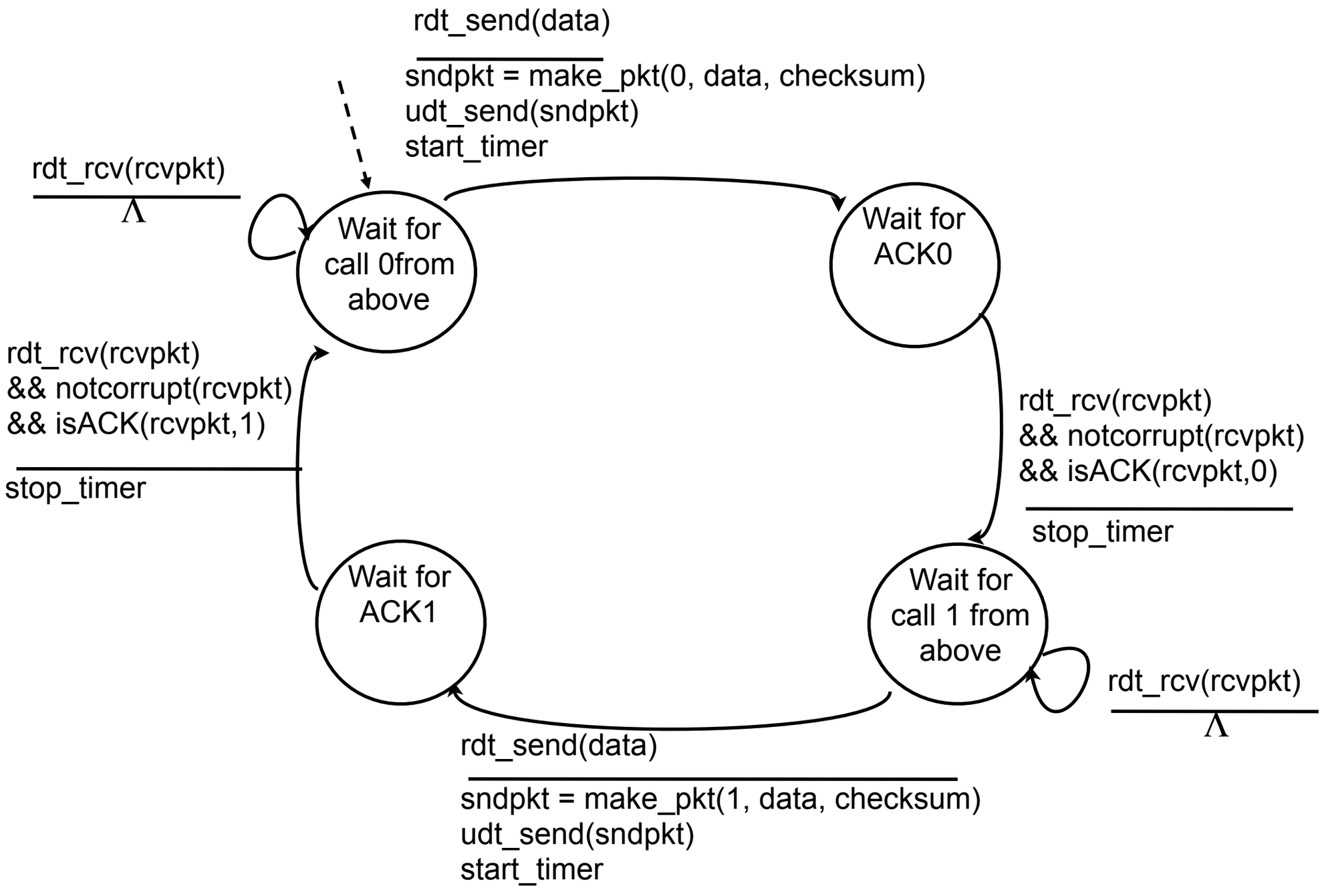
rdt3.0 sender



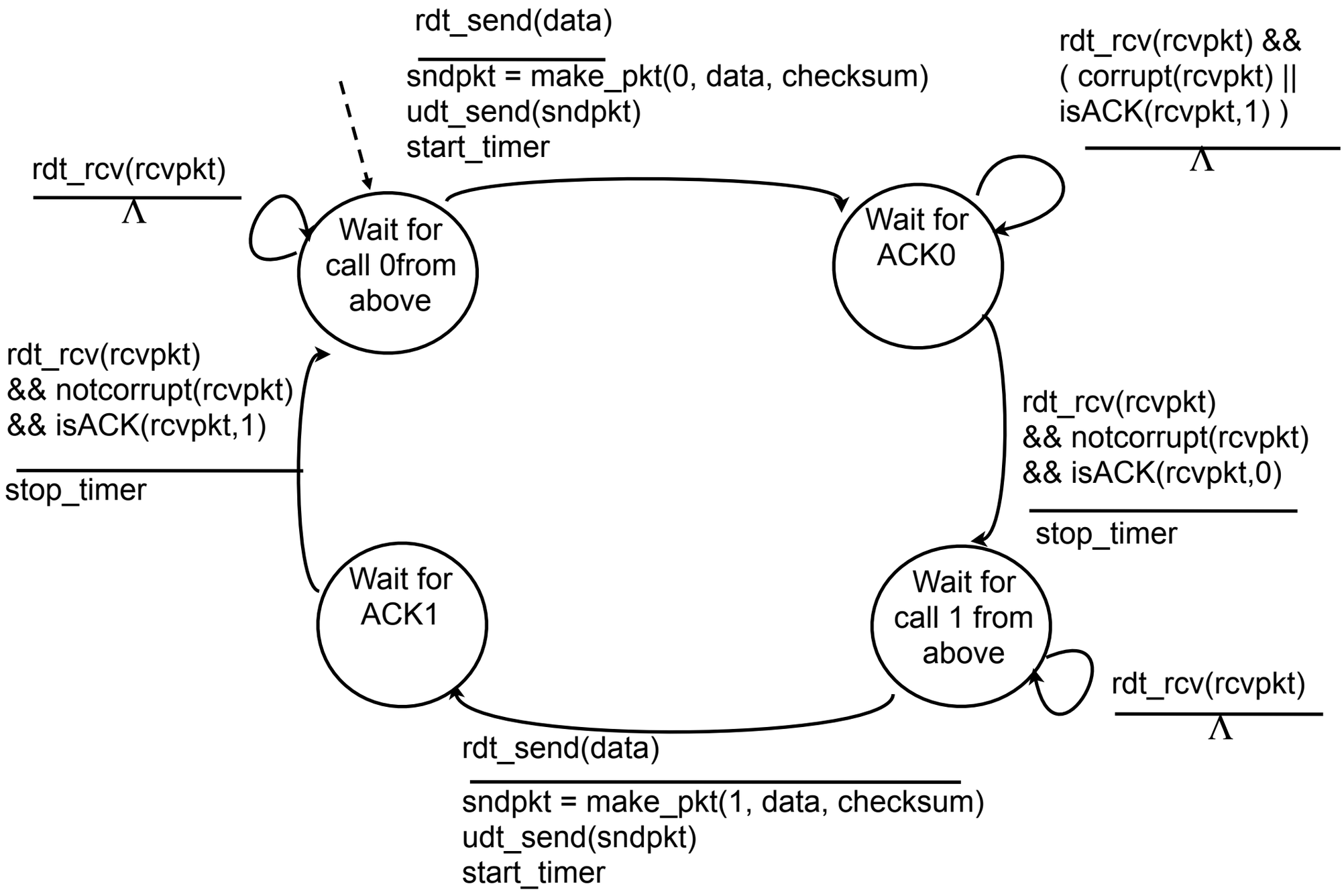
rdt3.0 sender



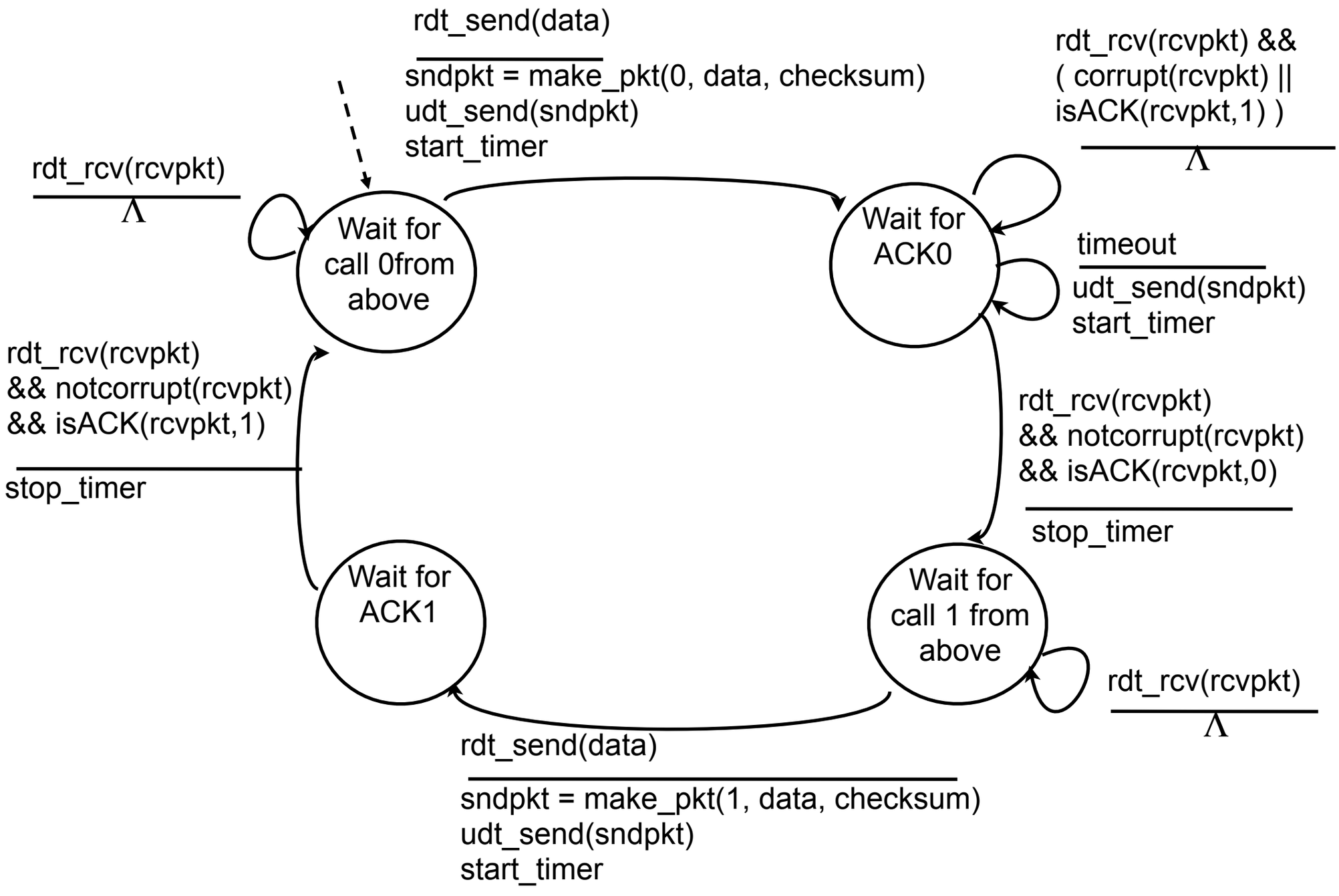
rdt3.0 sender



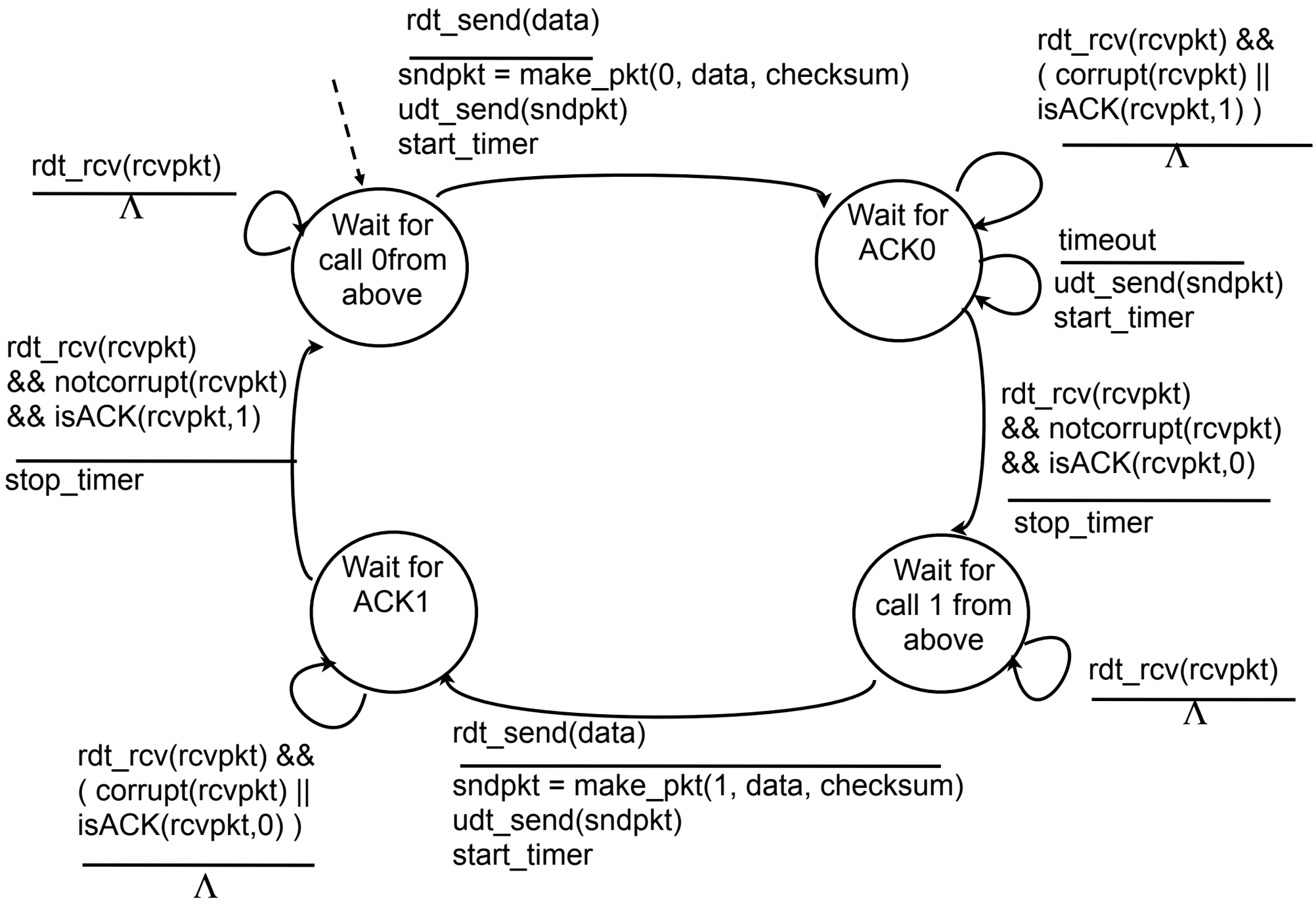
rdt3.0 sender



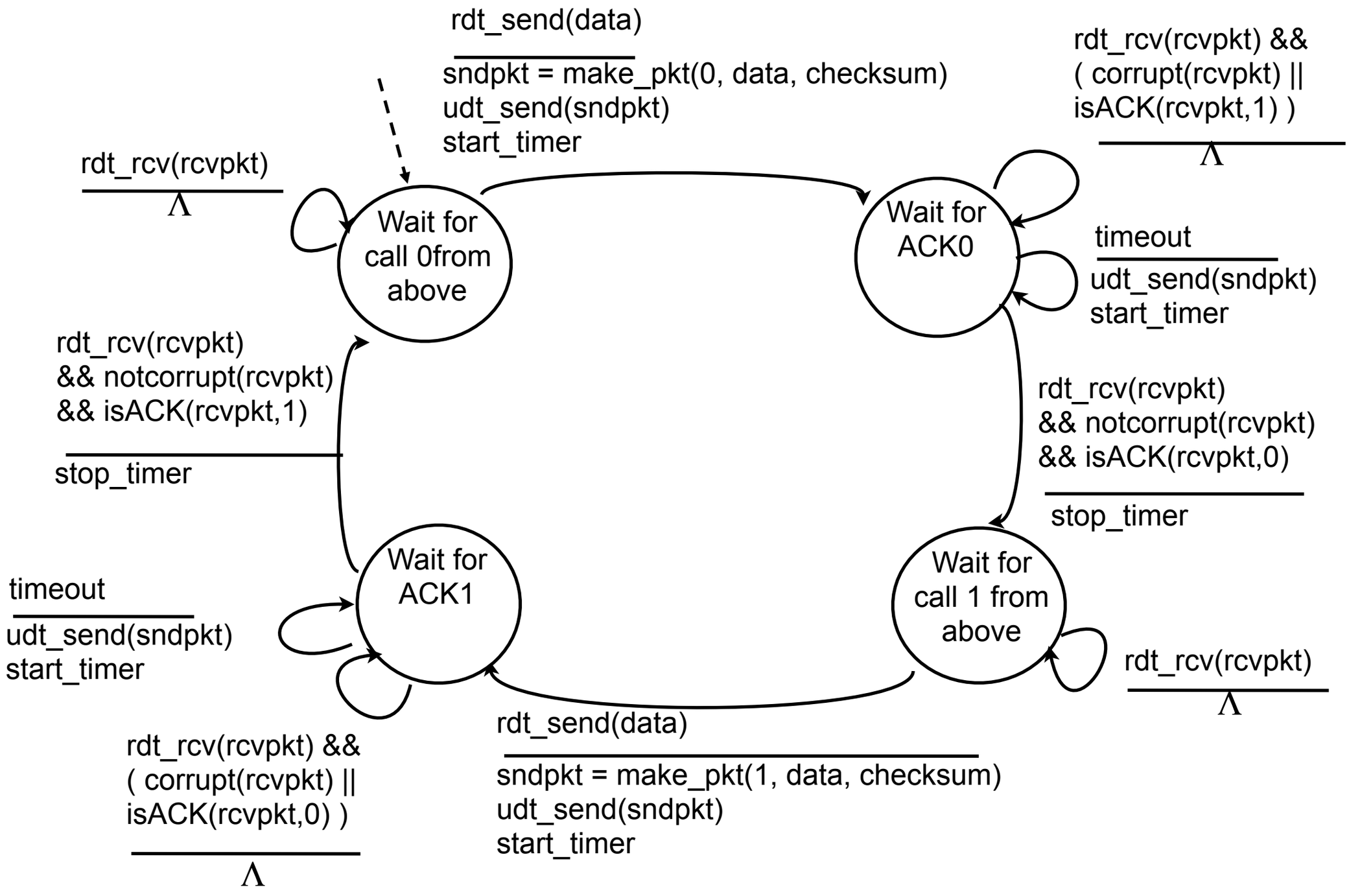
rdt3.0 sender



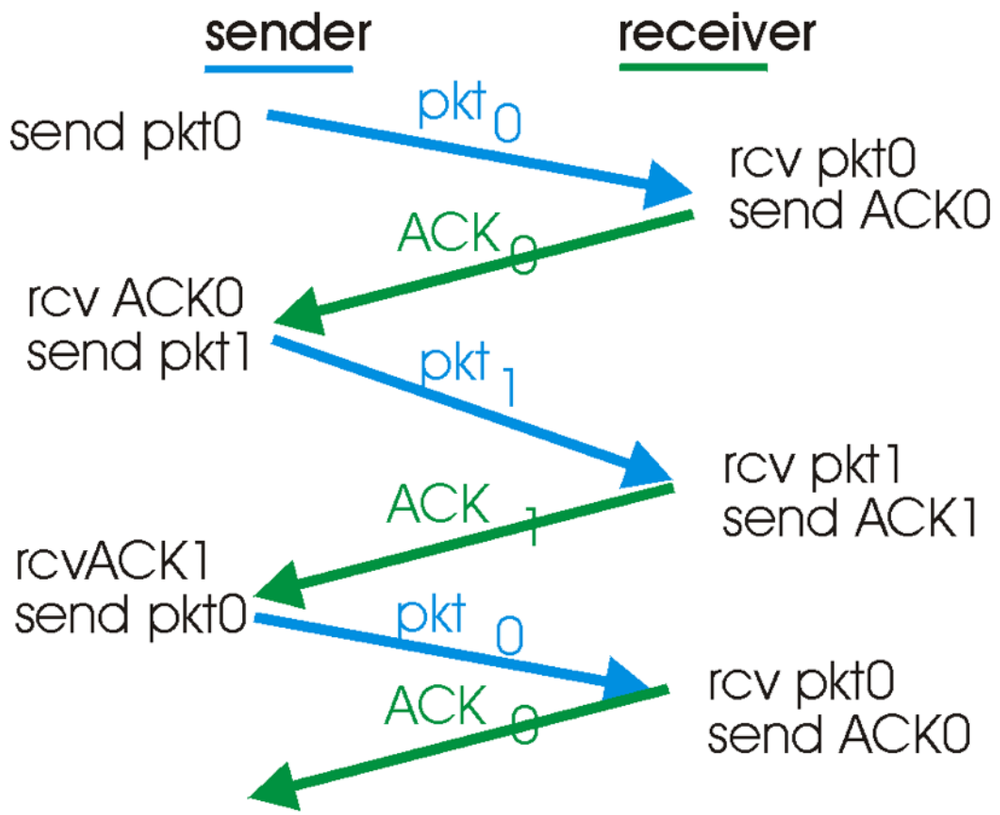
rdt3.0 sender



rdt3.0 sender

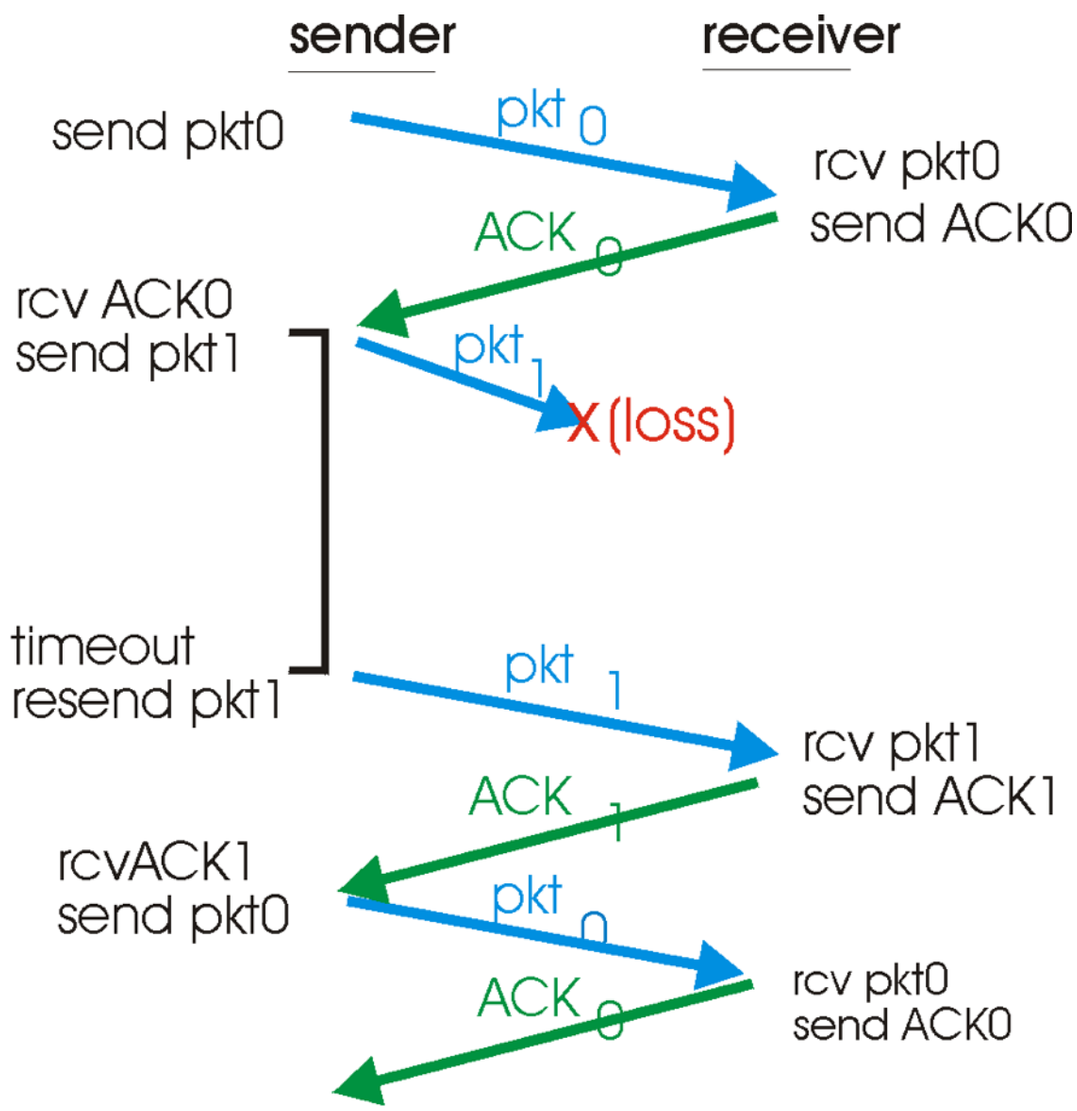


rdt3.0 in action



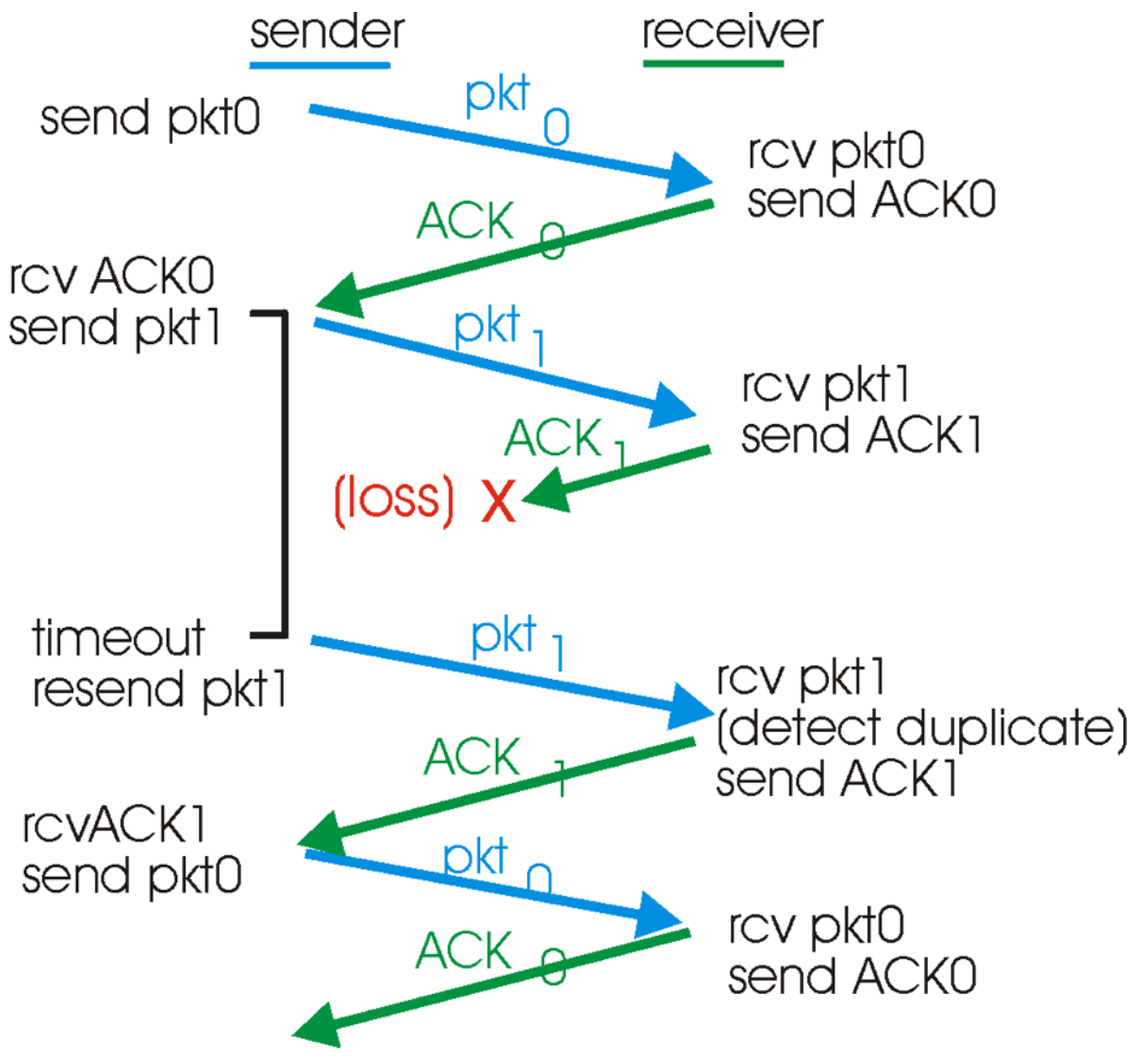
(a) operation with no loss

rdt3.0 in action



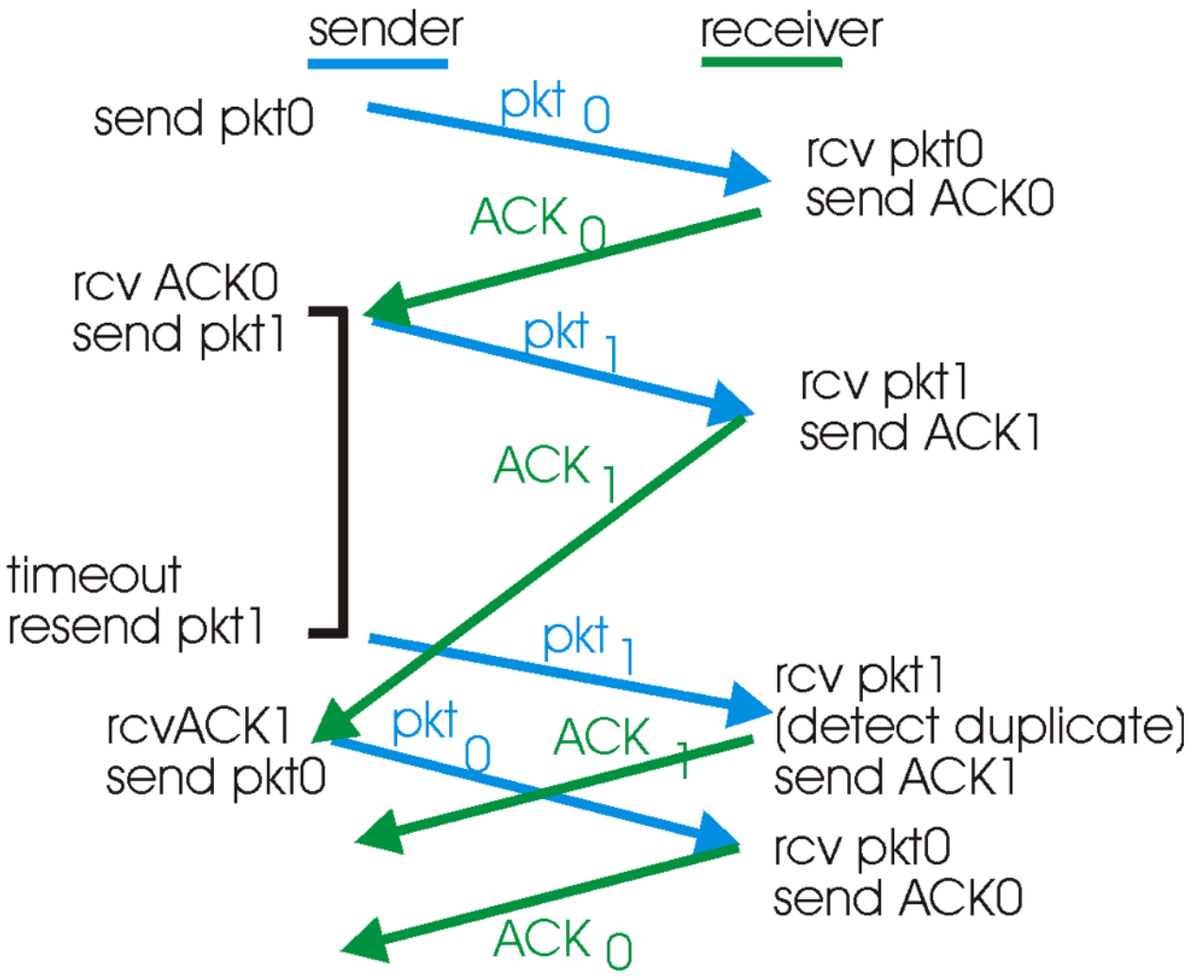
(b) lost packet

rdt3.0 in action



(c) lost ACK

rdt3.0 in action



(d) premature timeout

Performance of rdt3.0

Performance of rdt3.0

- rdt3.0 works, but performance stinks

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

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

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

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

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

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

- 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

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

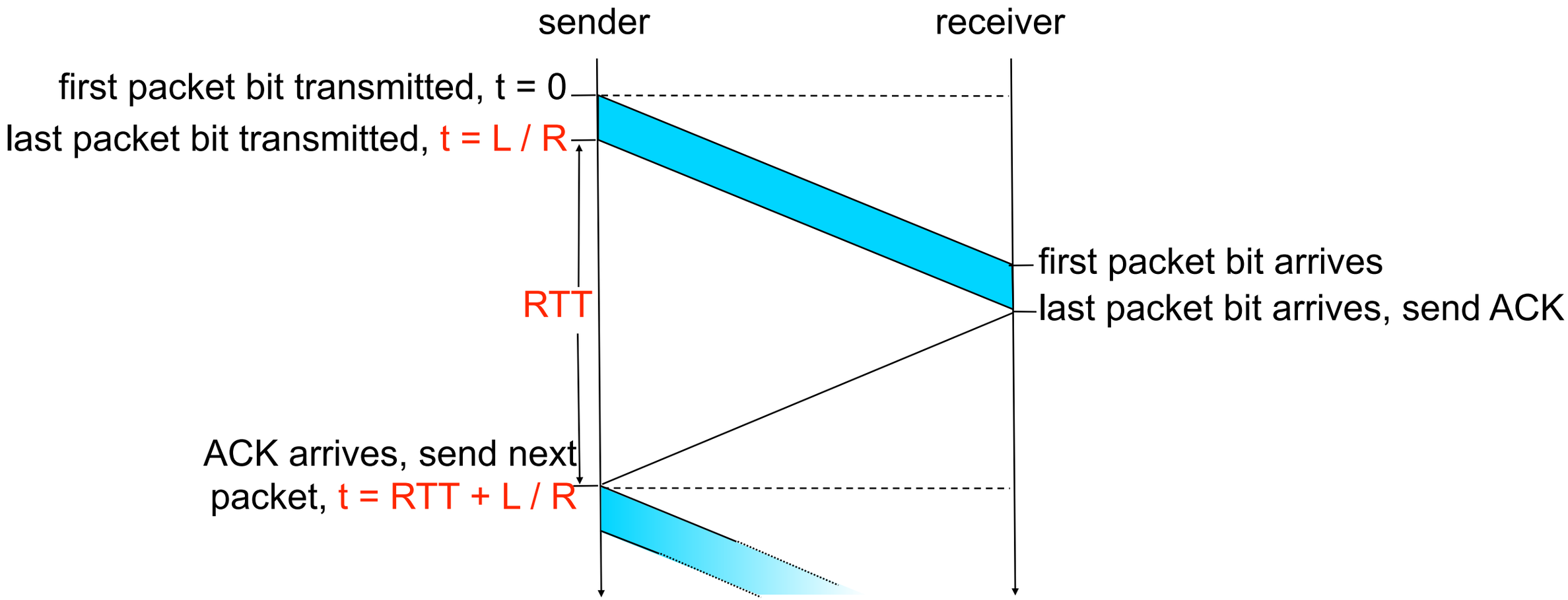
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

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

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

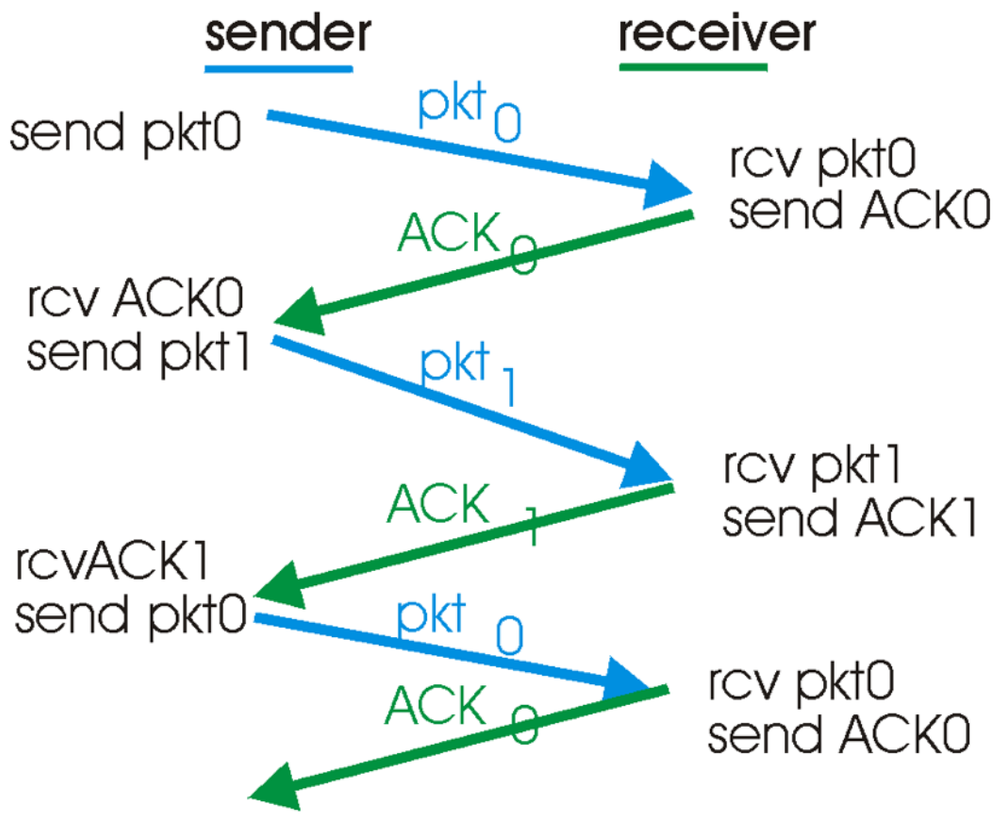
- if RTT=30 msec, 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- **protocol limits use of physical resources!**

rdt3.0: stop-and-wait operation



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

rdt3.0: stop-and-wait

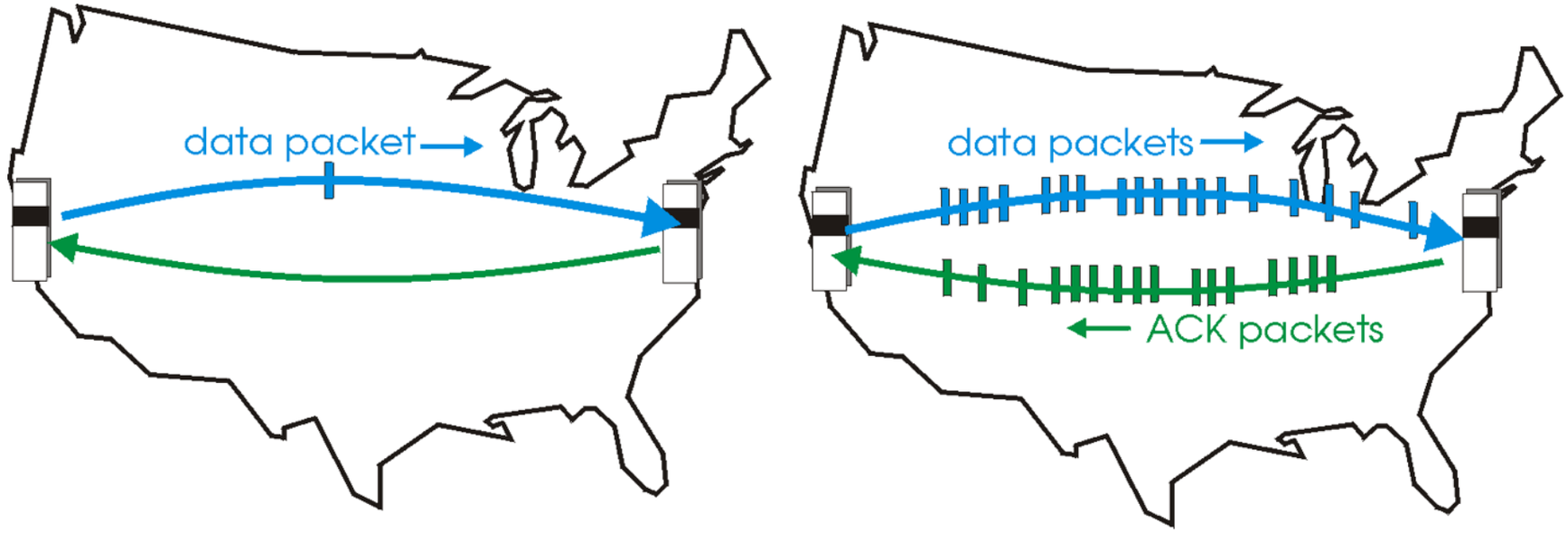


(a) operation with no loss

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

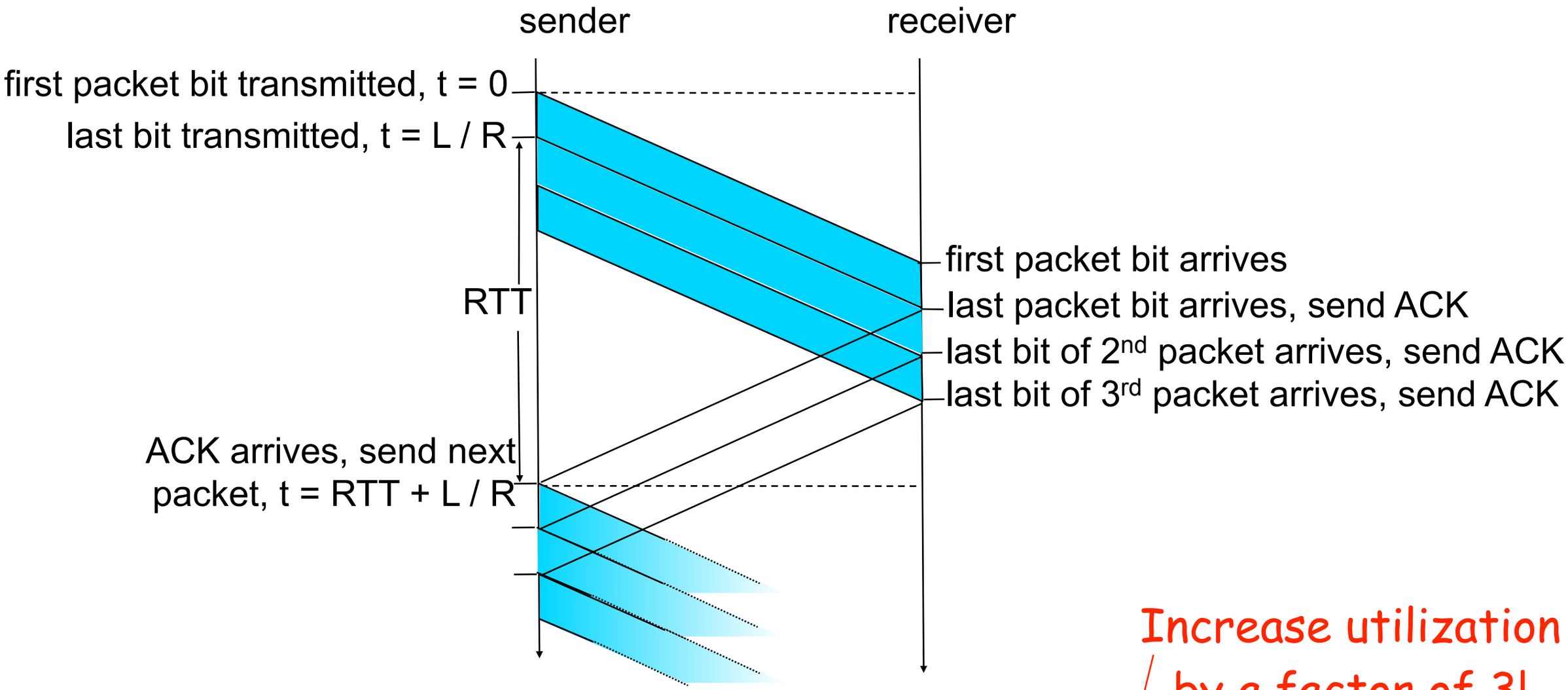


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- two generic forms of pipelined protocols: go-Back-N, selective repeat

Pipelining: increased utilization



Increase utilization by a factor of 3!

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

Pipelined Protocols

Pipelined Protocols

Go-back-N: big picture:

- sender can have up to N unacked packets in pipeline
- rcvr only sends cumulative acks
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - if timer expires, retransmit all unack'ed packets

Pipelined Protocols

Go-back-N: big picture:

- sender can have up to N unacked packets in pipeline
- rcvr only sends cumulative acks
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - if timer expires, retransmit all unack'ed packets

Selective Repeat: big pic

- sender can have up to N unack'ed packets in pipeline
- rcvr sends individual ack for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only unack'ed packet

Go-Back-N

Sender:

- k-bit seq # in pkt header
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window

GBN: receiver

GBN: receiver

ACK-only: always send ACK for correctly-received pkt with highest in-order seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

GBN: receiver

ACK-only: always send ACK for correctly-received pkt with highest in-order seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**

❖ out-of-order pkt:

- discard (don't buffer) -> no receiver buffering!
- Re-ACK pkt with highest in-order seq #

Selective Repeat

- receiver individually acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACK'ed pkts

Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

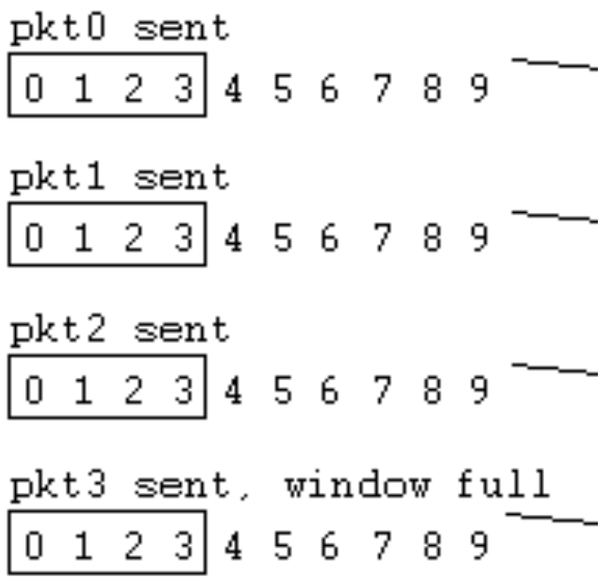
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

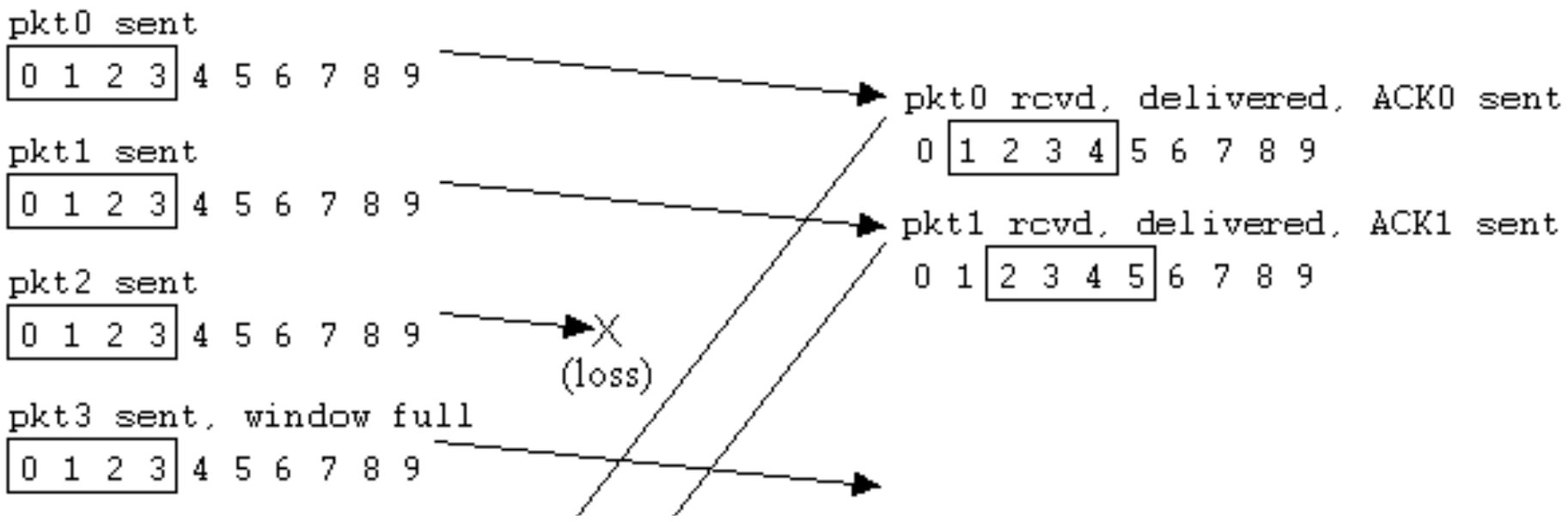
otherwise:

- ignore

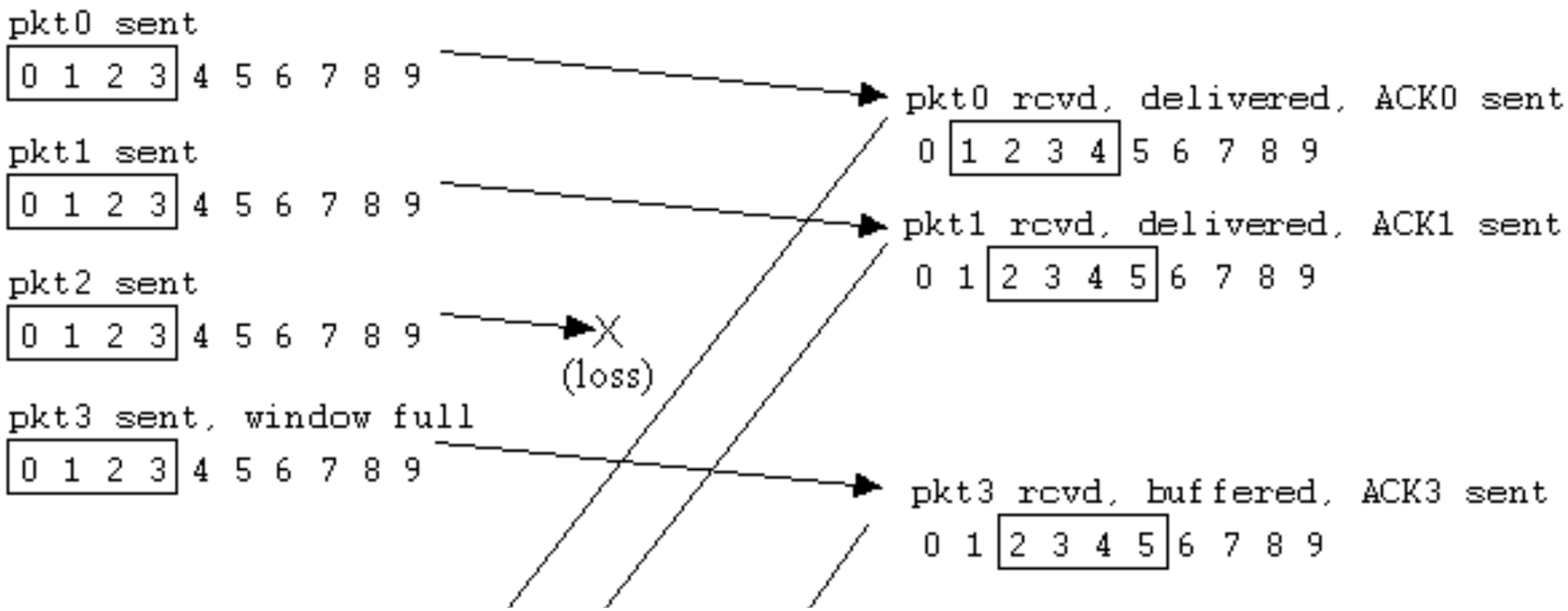
Selective repeat in action



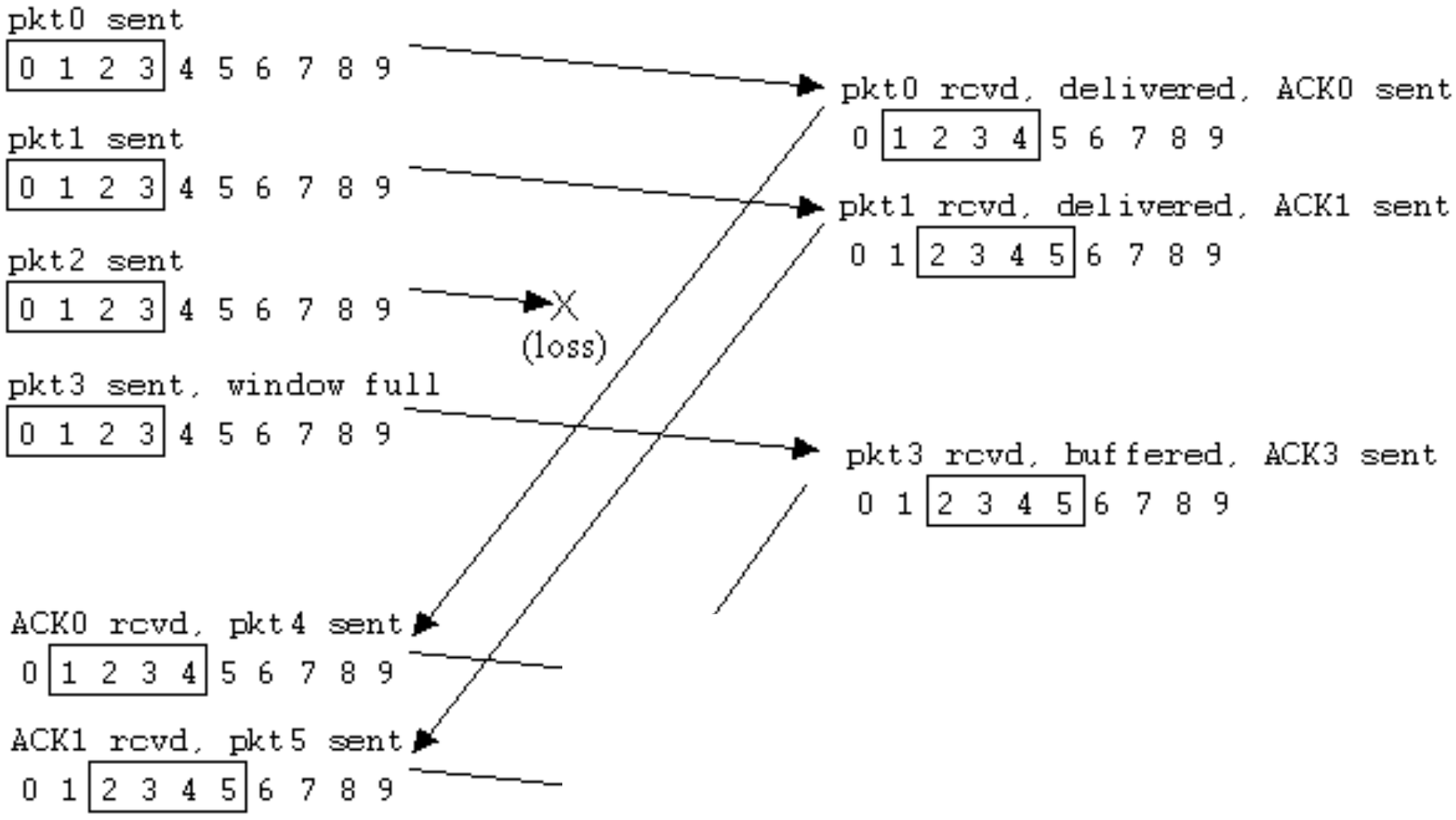
Selective repeat in action



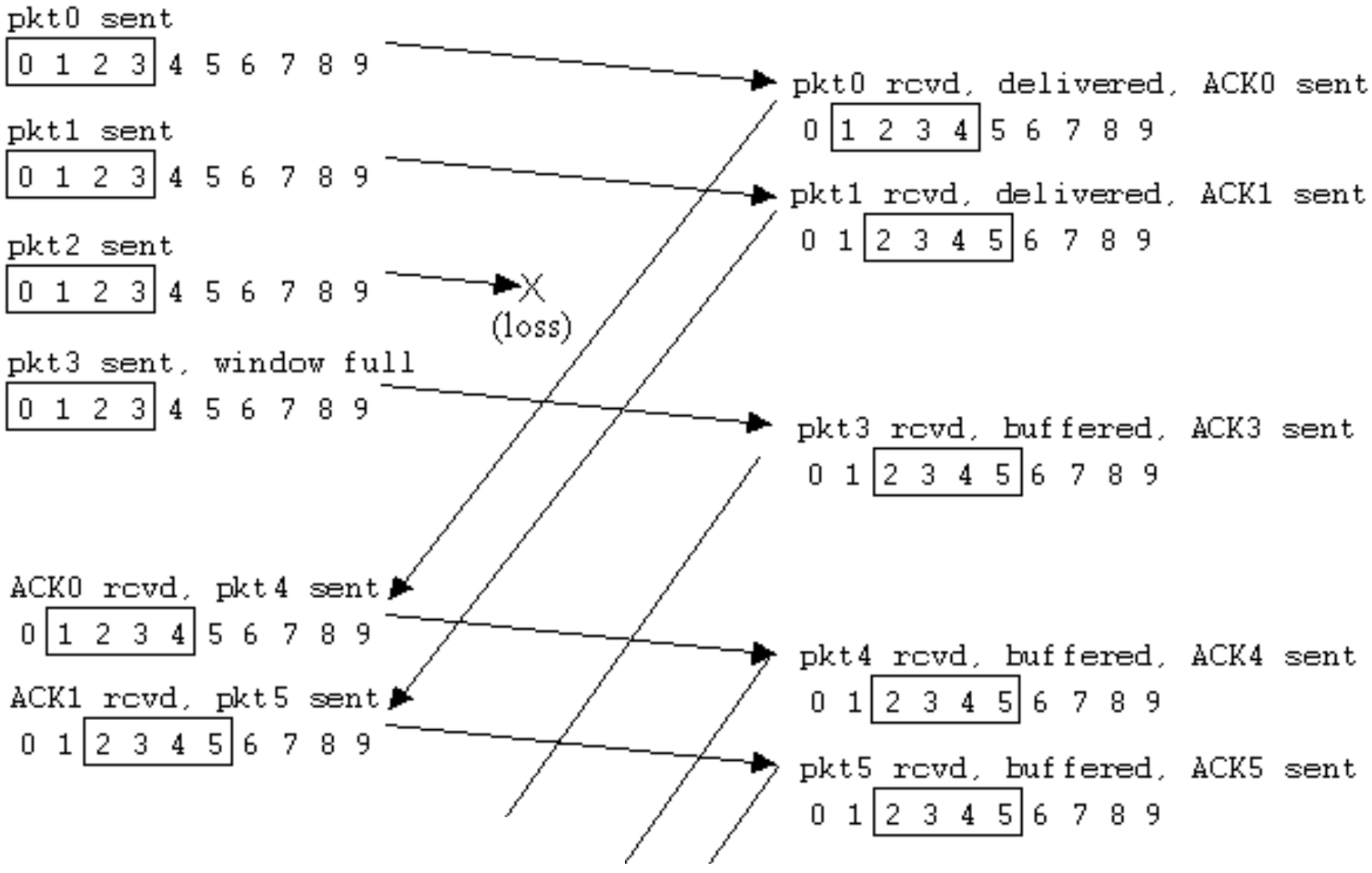
Selective repeat in action



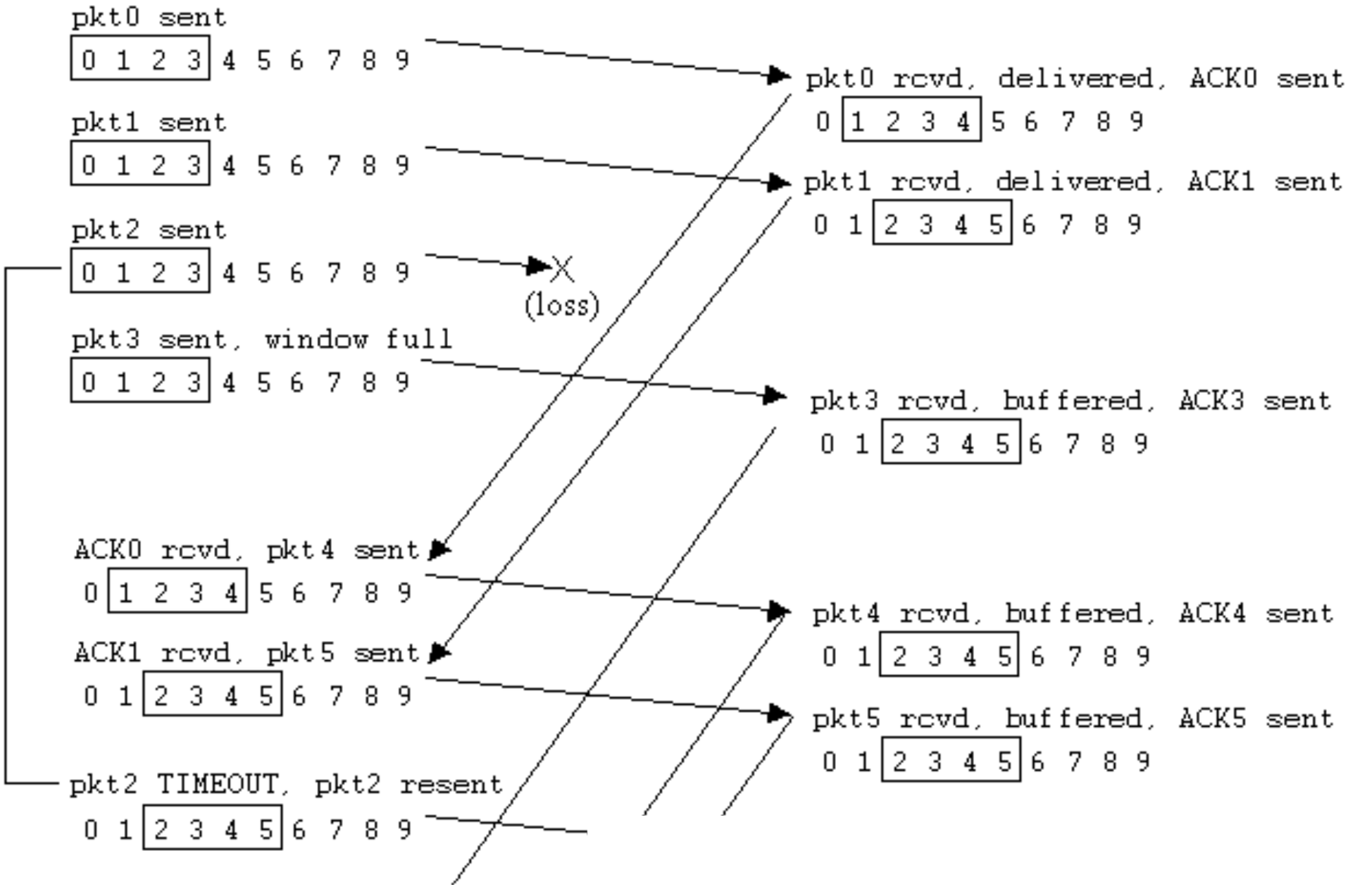
Selective repeat in action



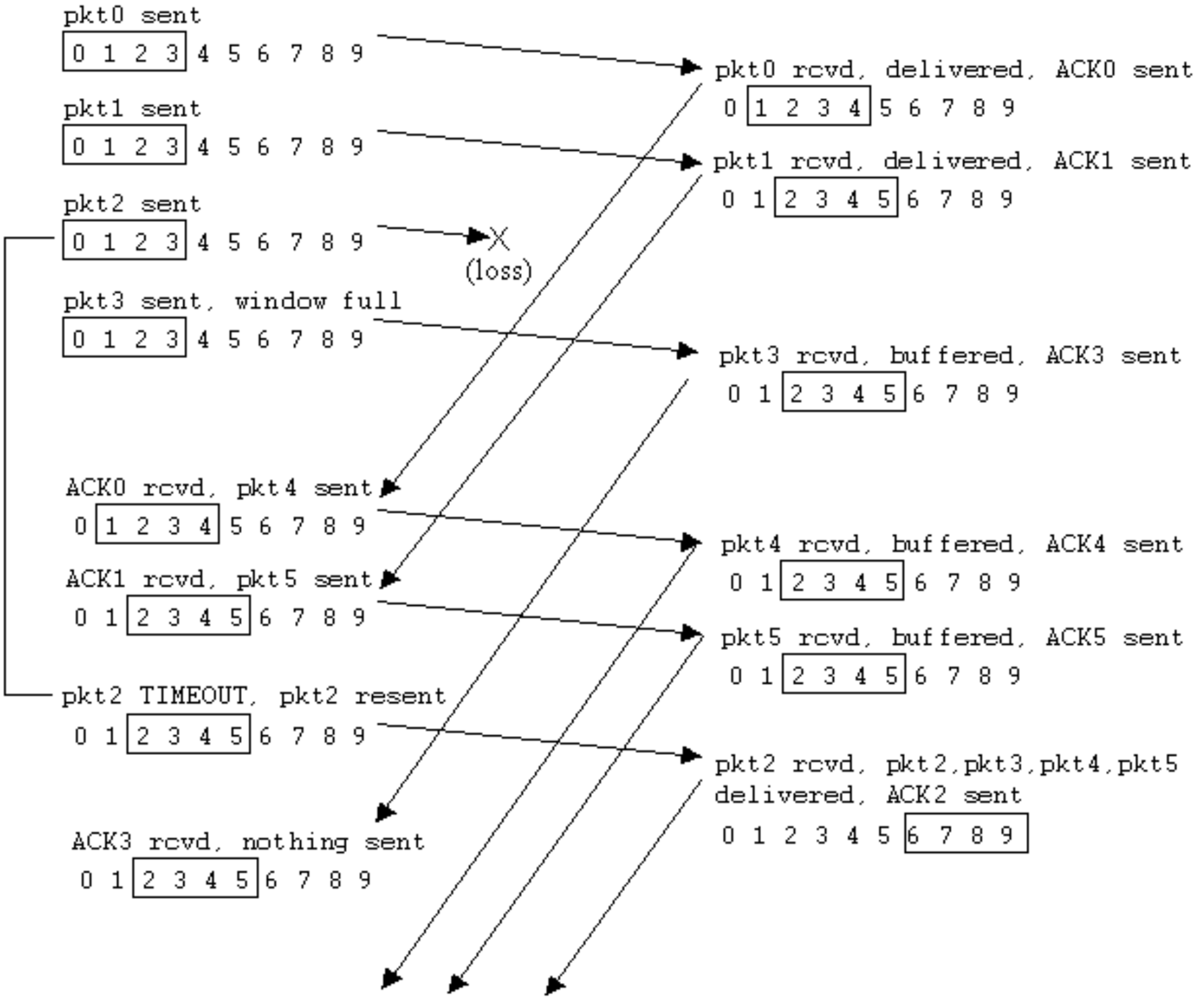
Selective repeat in action



Selective repeat in action



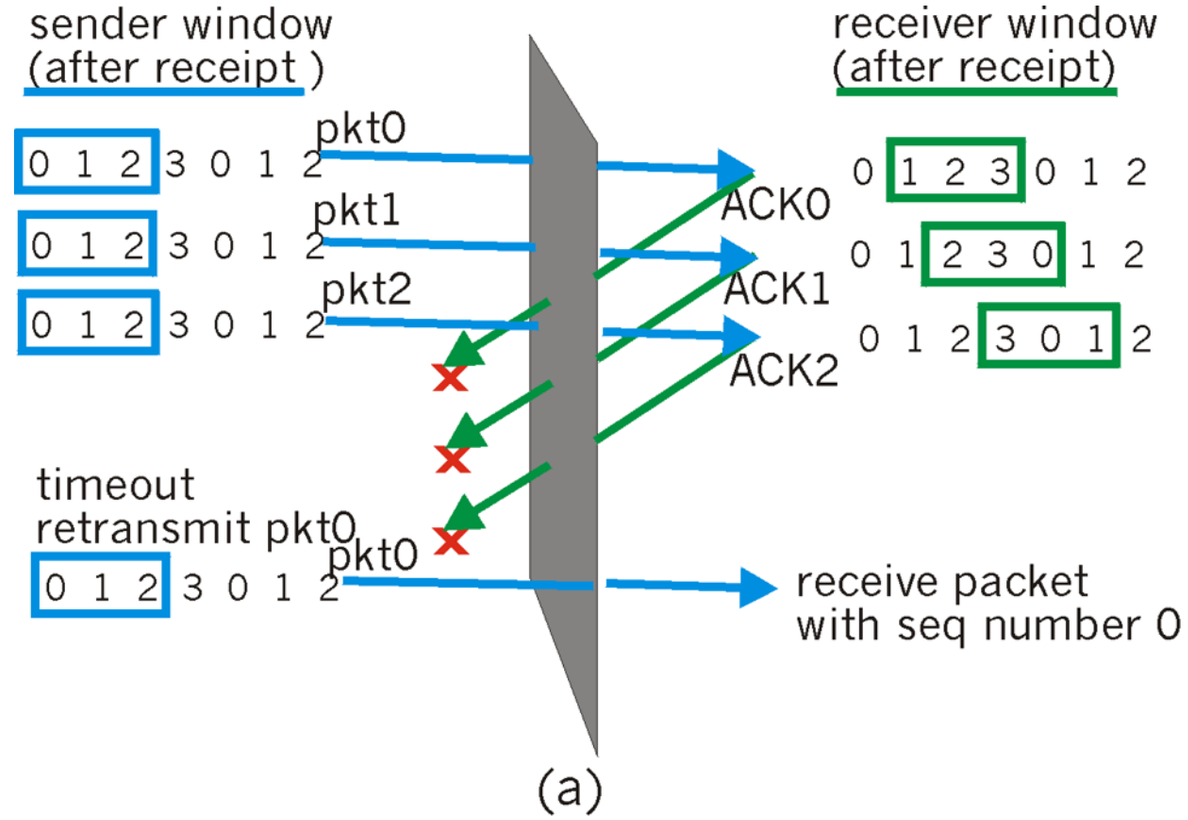
Selective repeat in action



Selective repeat: dilemma

Example:

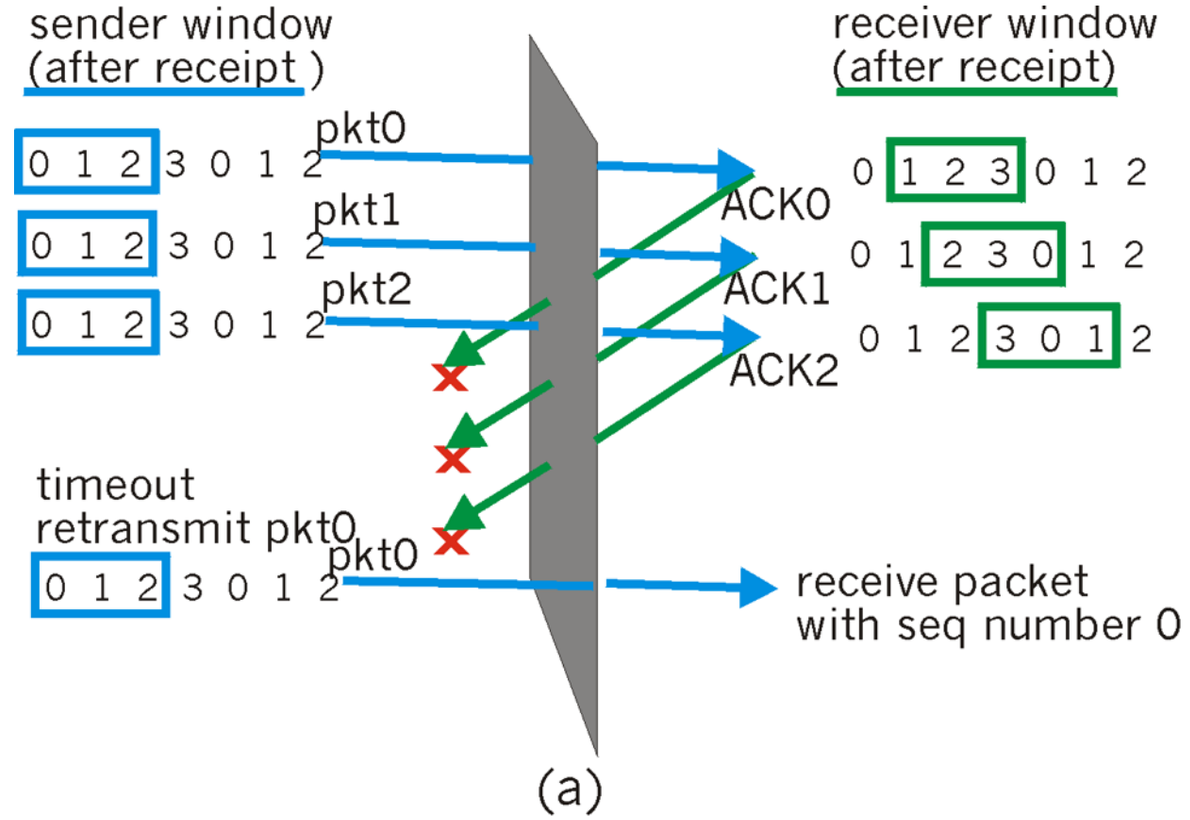
- seq #'s: 0, 1, 2, 3
- window size=3



Selective repeat: dilemma

Example:

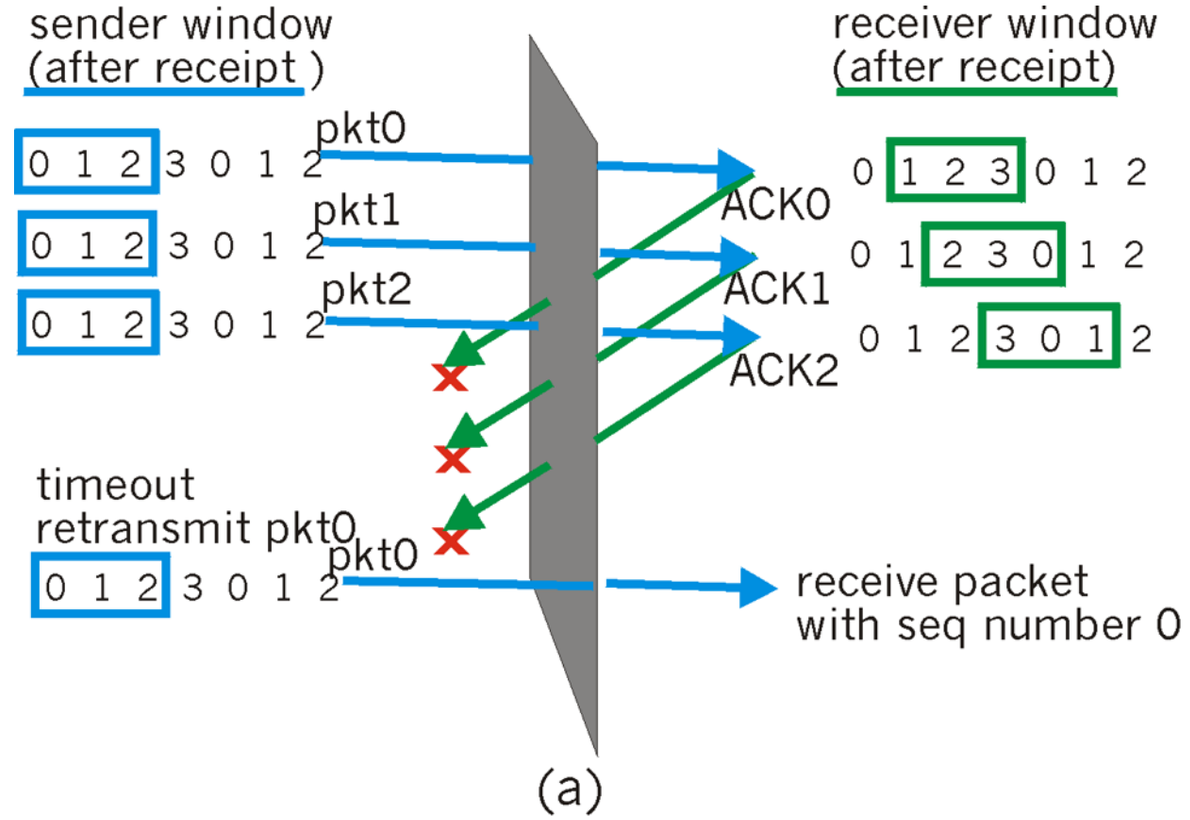
- seq #'s: 0, 1, 2, 3
- window size=3



Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

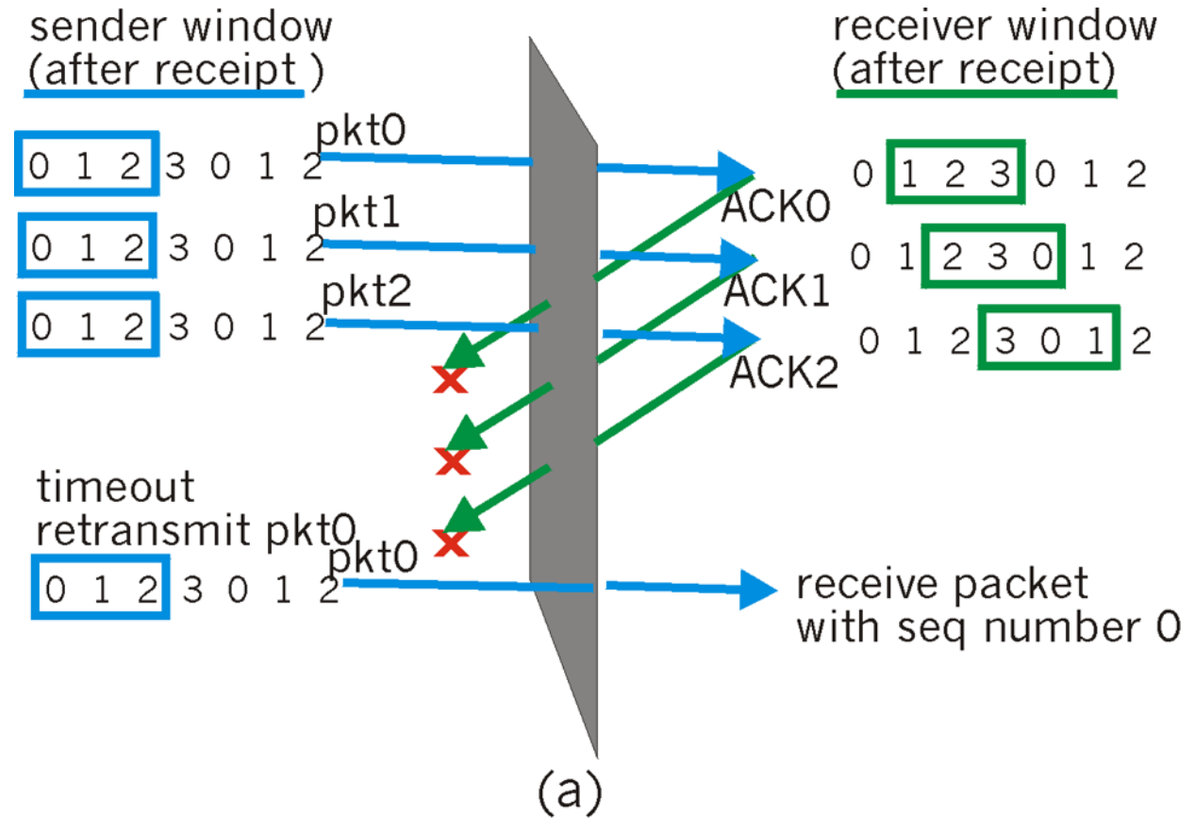


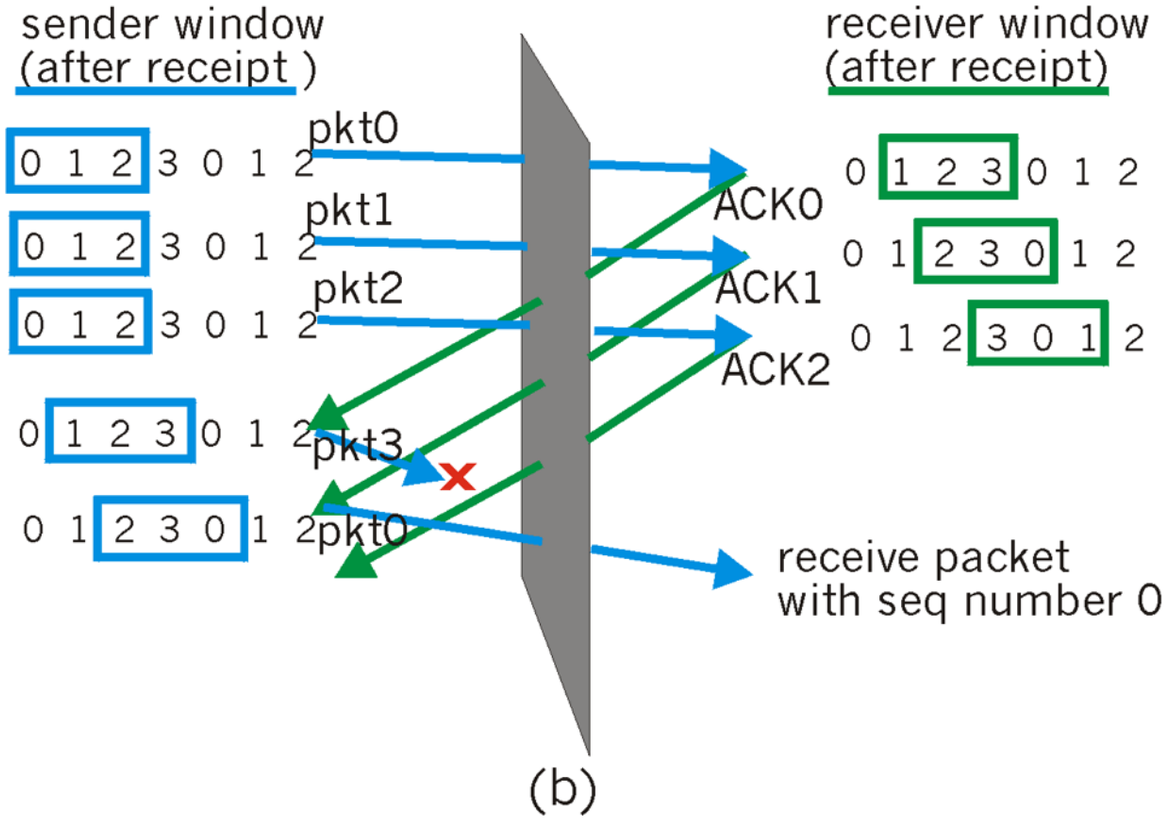
Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)





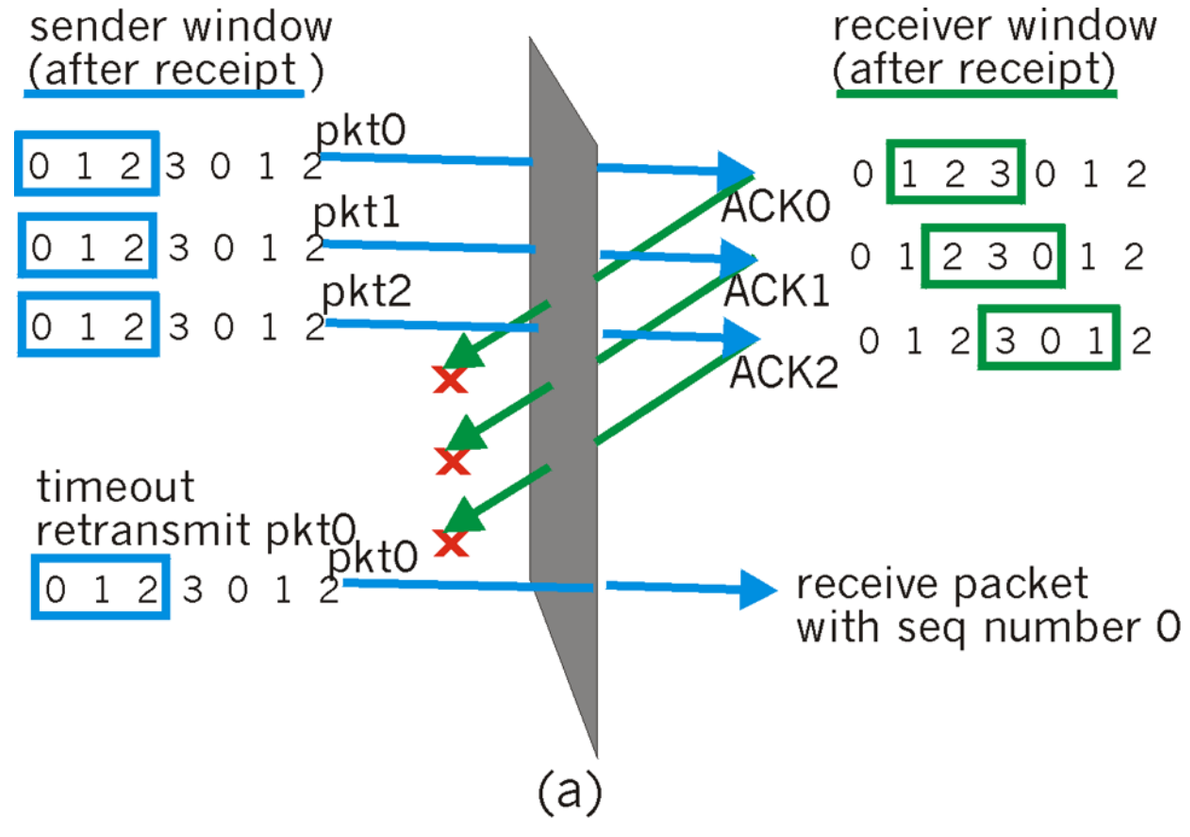
Selective repeat: dilemma

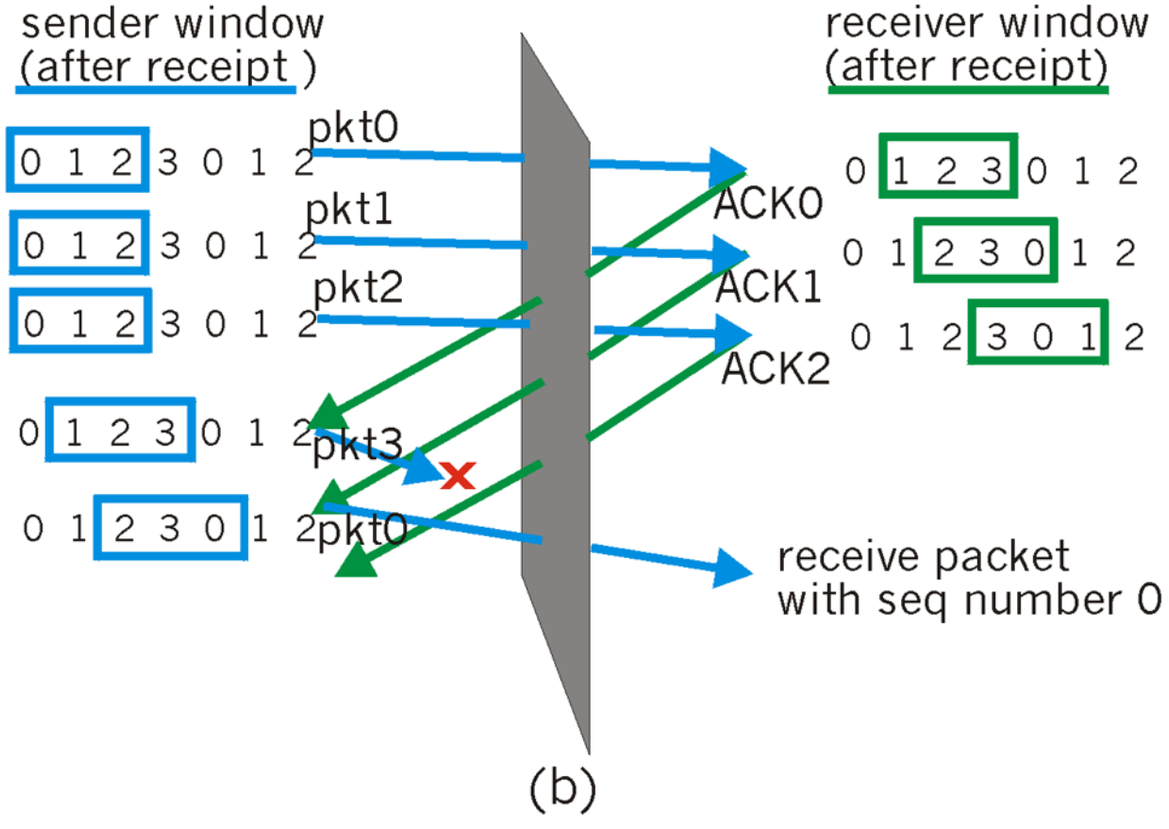
Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?





Reliability Concepts

- Checksums
- Sequence numbers
- Acknowledgments
- Negative ACKs
- Window + pipelining
- Timers