Transport Layer
Part 2

Mark Allman
mallman@case.edu

EECS 325/425
Fall 2018

"Well, we busted outa class, had to get away from those fools,
We learned more from a three minute record than we ever learned in school."

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.

Connectionless demultiplexing

Connectionless demultiplexing

- create UDP sockets with host-local port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

Connectionless demultiplexing

- create UDP sockets with host-local port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

- when creating datagram to send into UDP socket, must specify: (dest IP address, dest port number)

Connectionless demultiplexing

- create UDP sockets with host-local port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

- when creating datagram to send into UDP socket, must specify: (dest IP address, dest port number)
- when host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

Connectionless demultiplexing

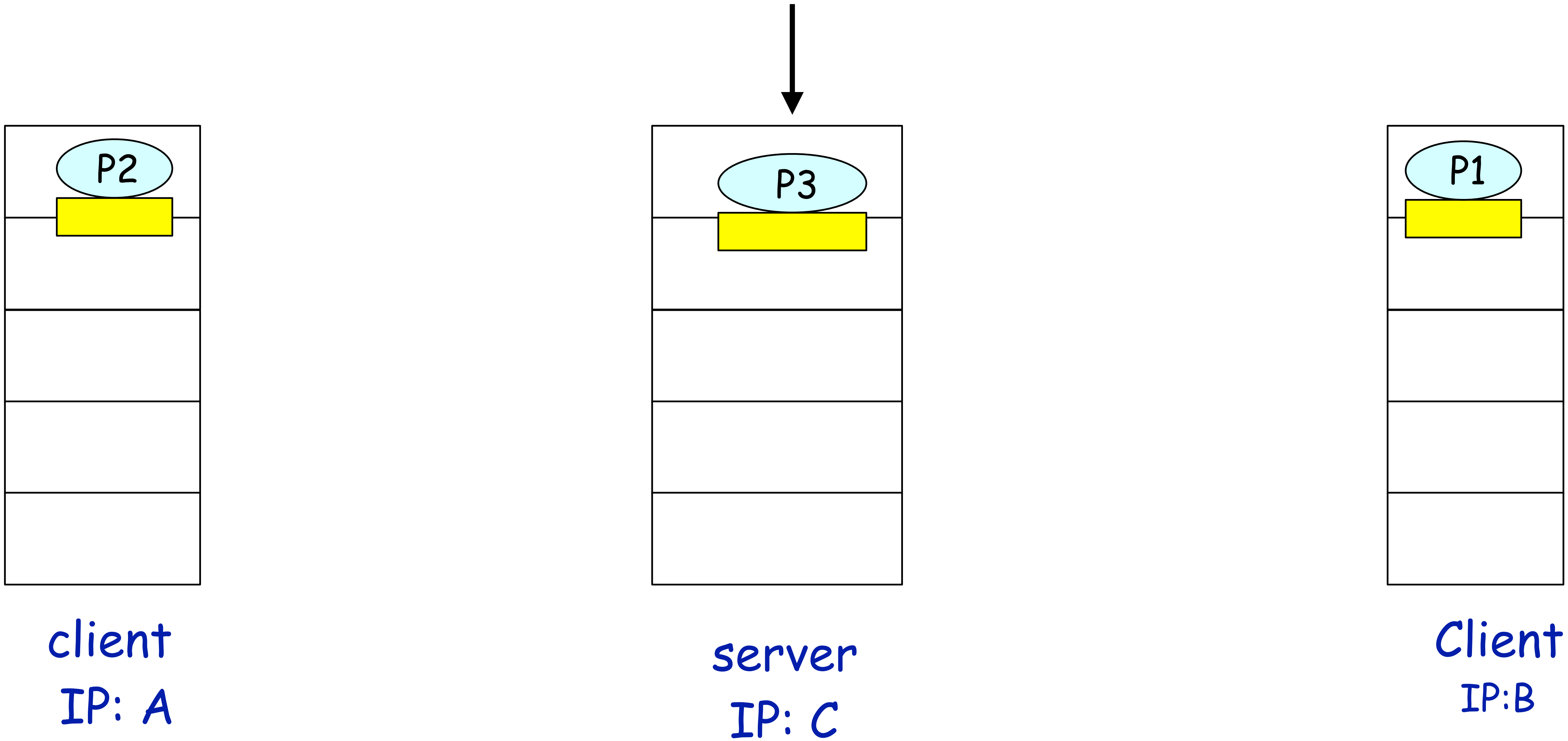
- create UDP sockets with host-local port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

- when creating datagram to send into UDP socket, must specify: (dest IP address, dest port number)
- when host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
  - i.e., demuxing based on destination port only

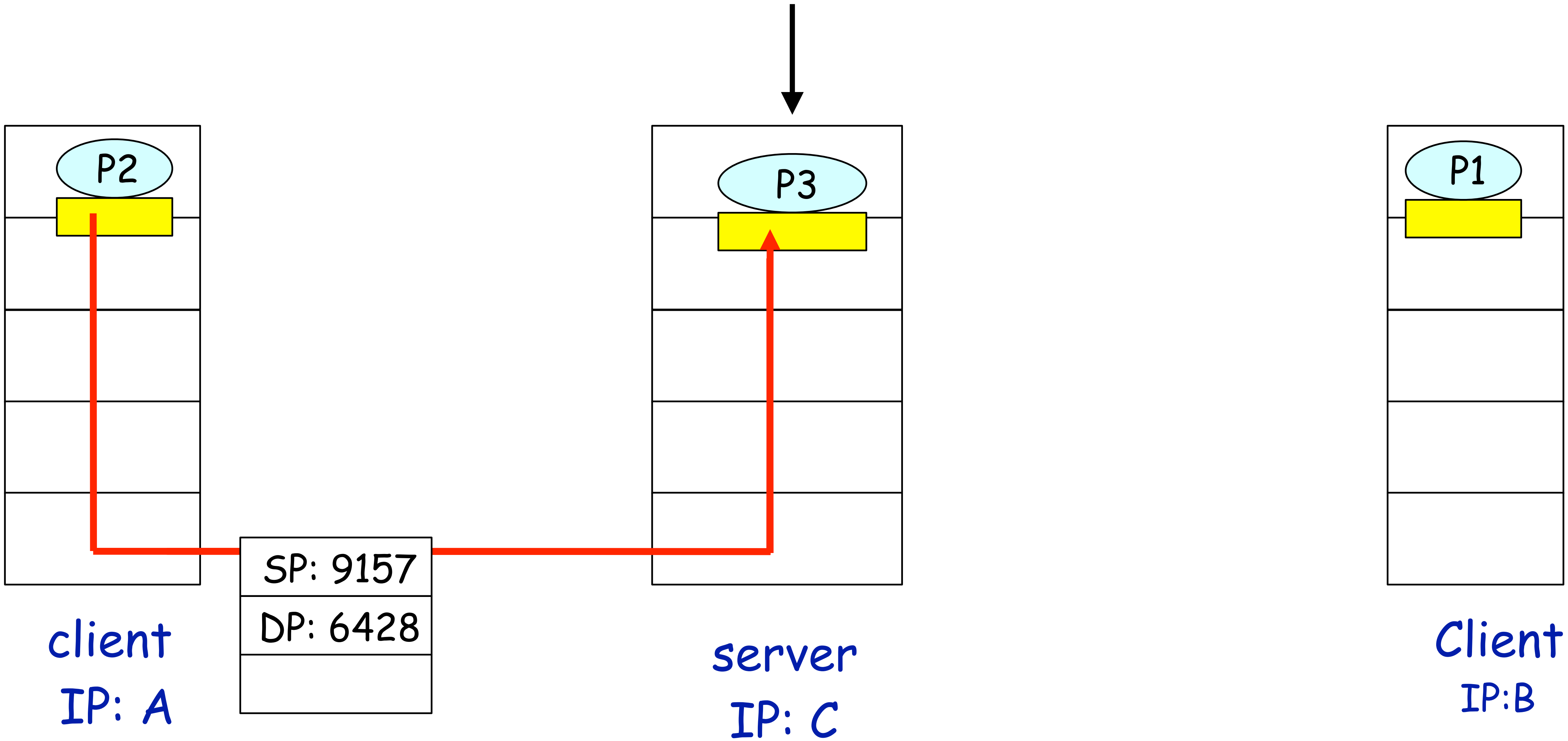
Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



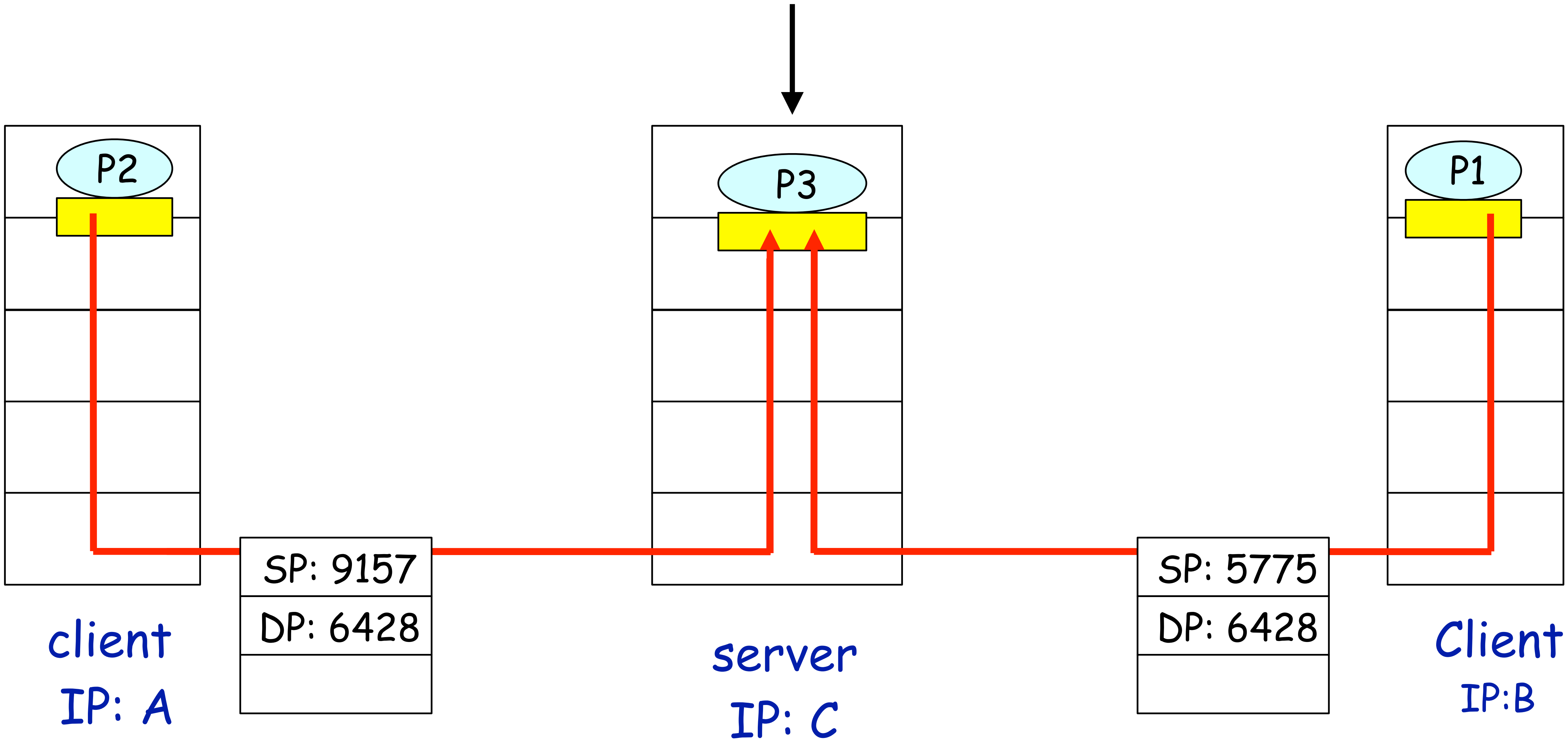
Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



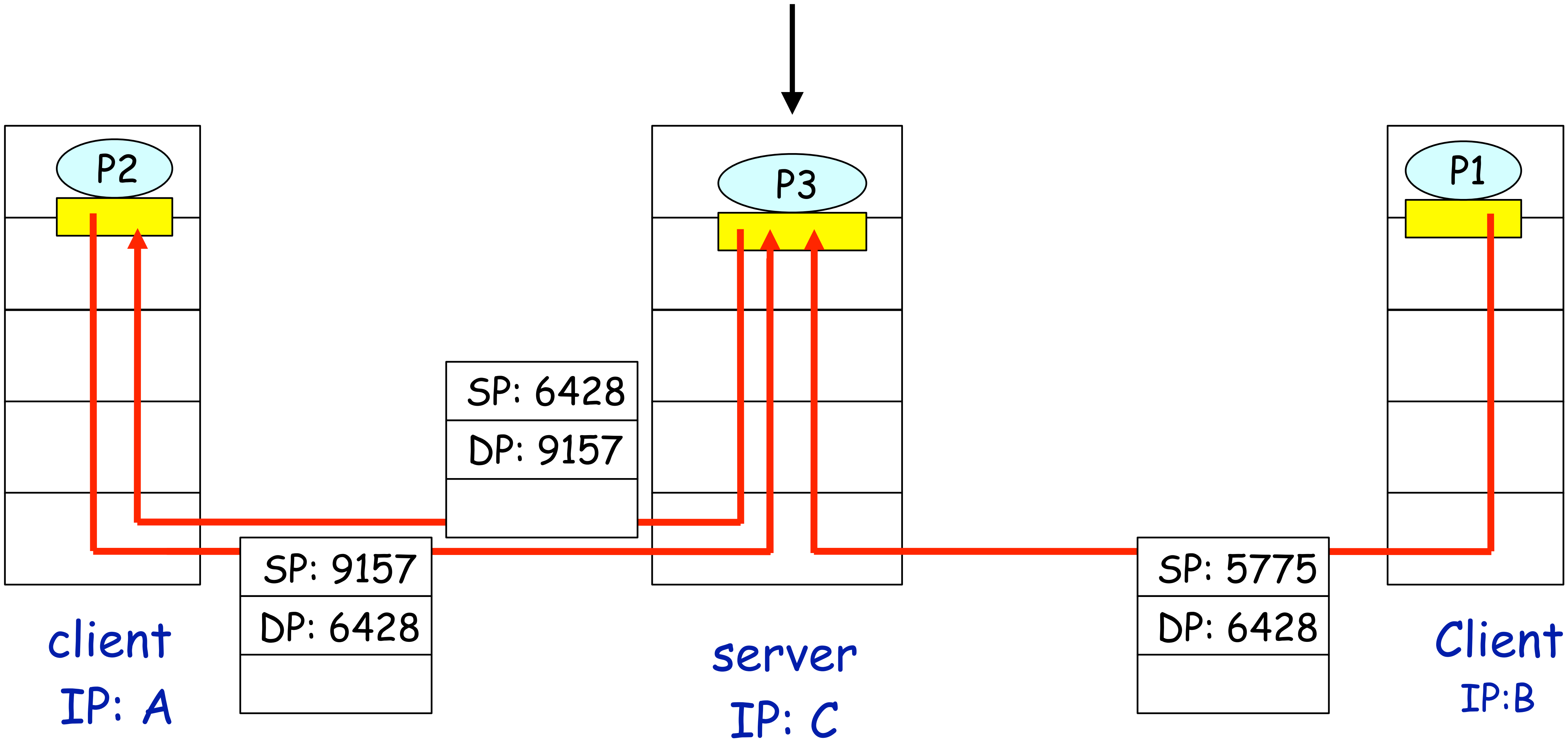
Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



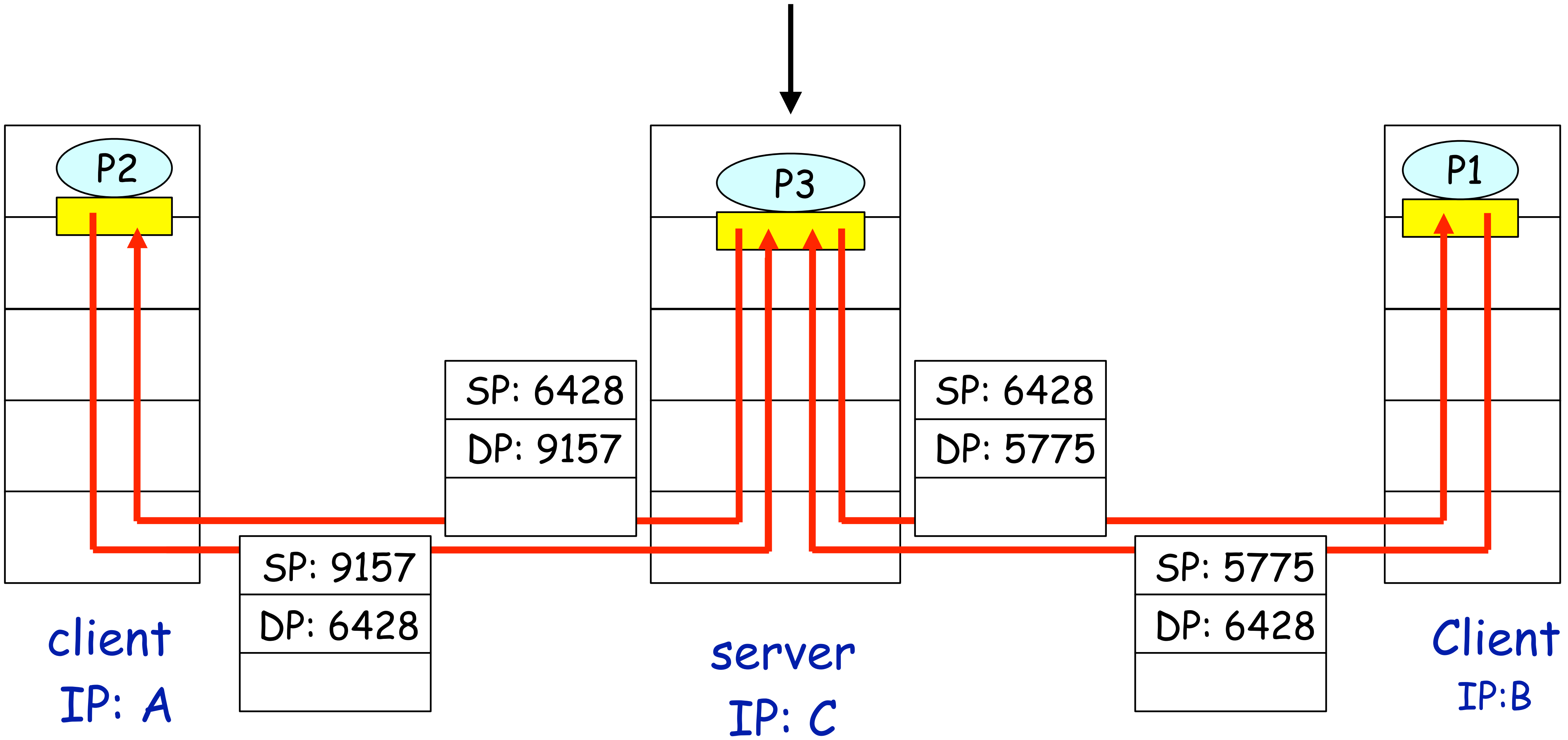
Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



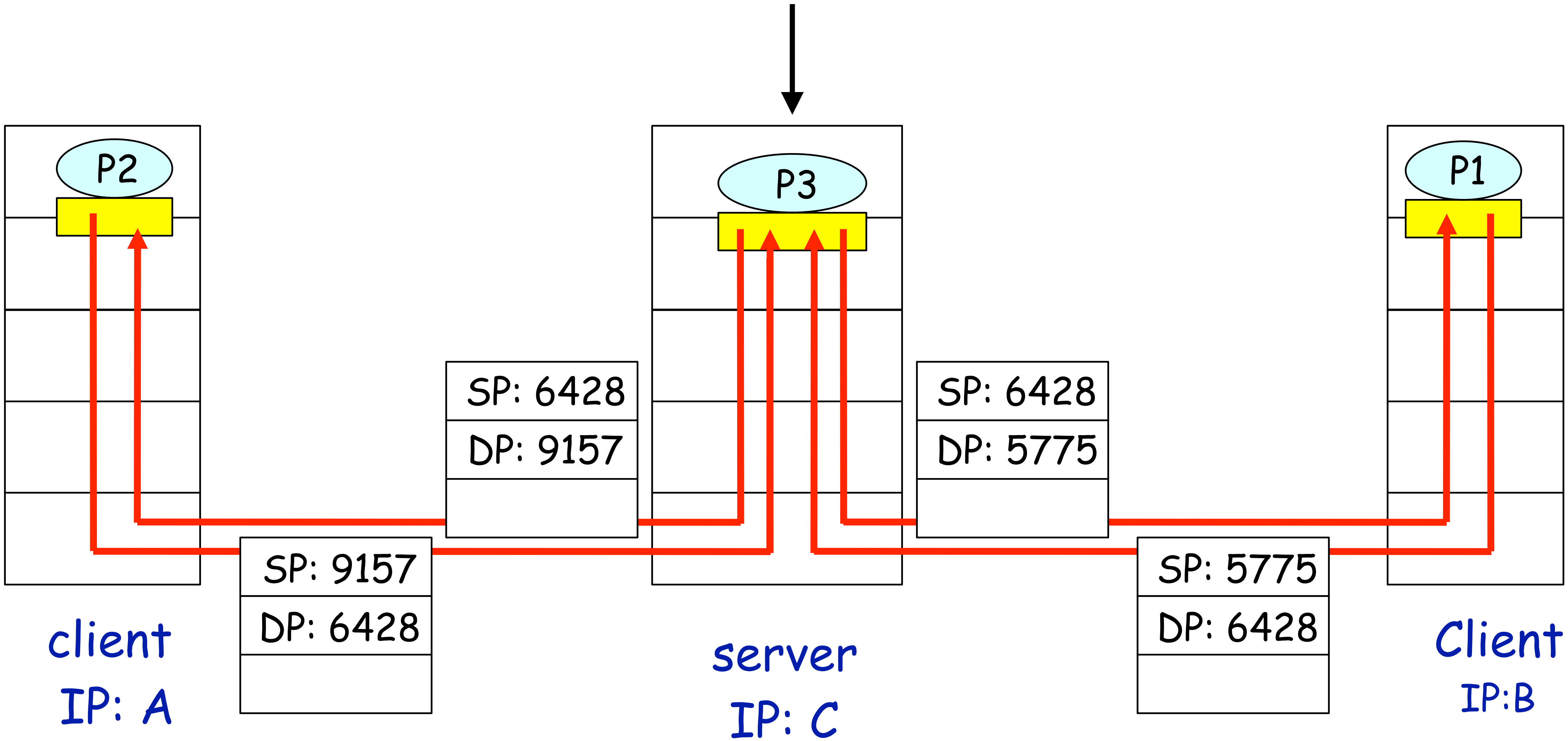
Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



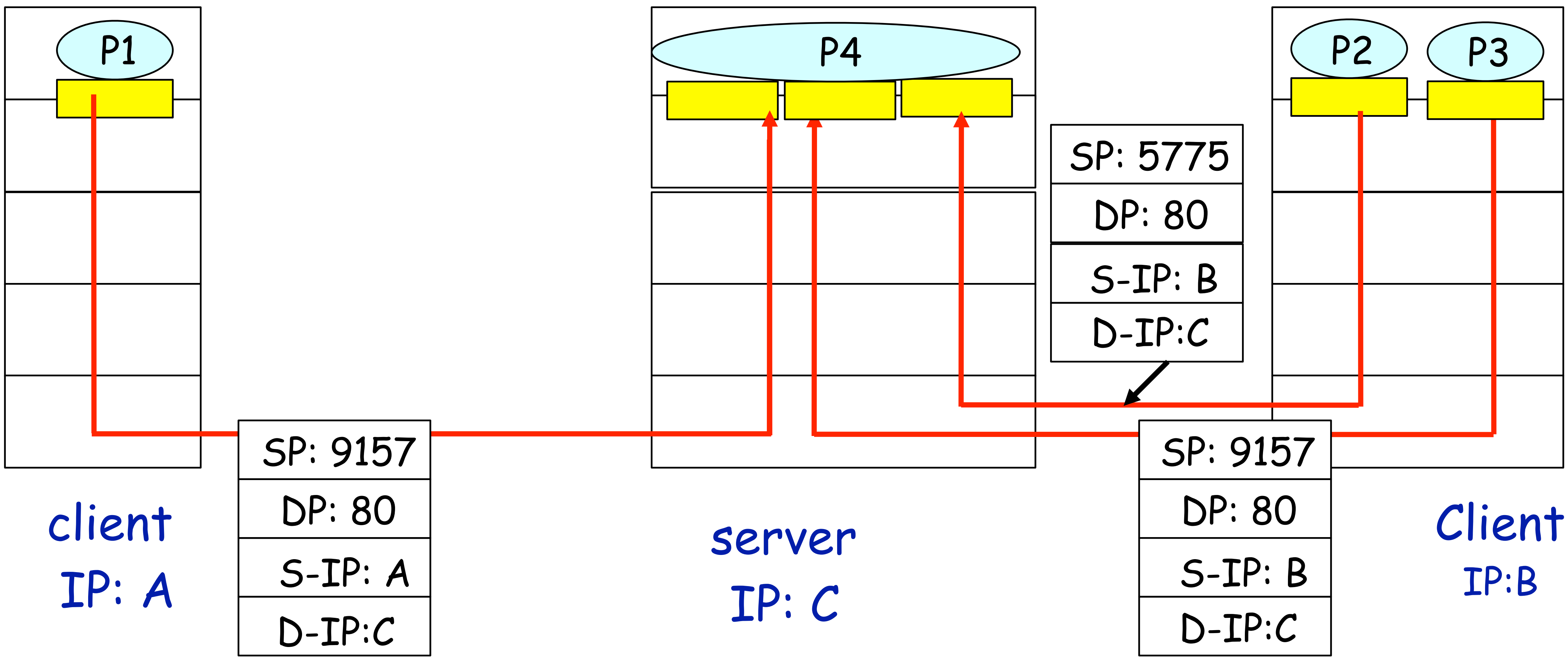
Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

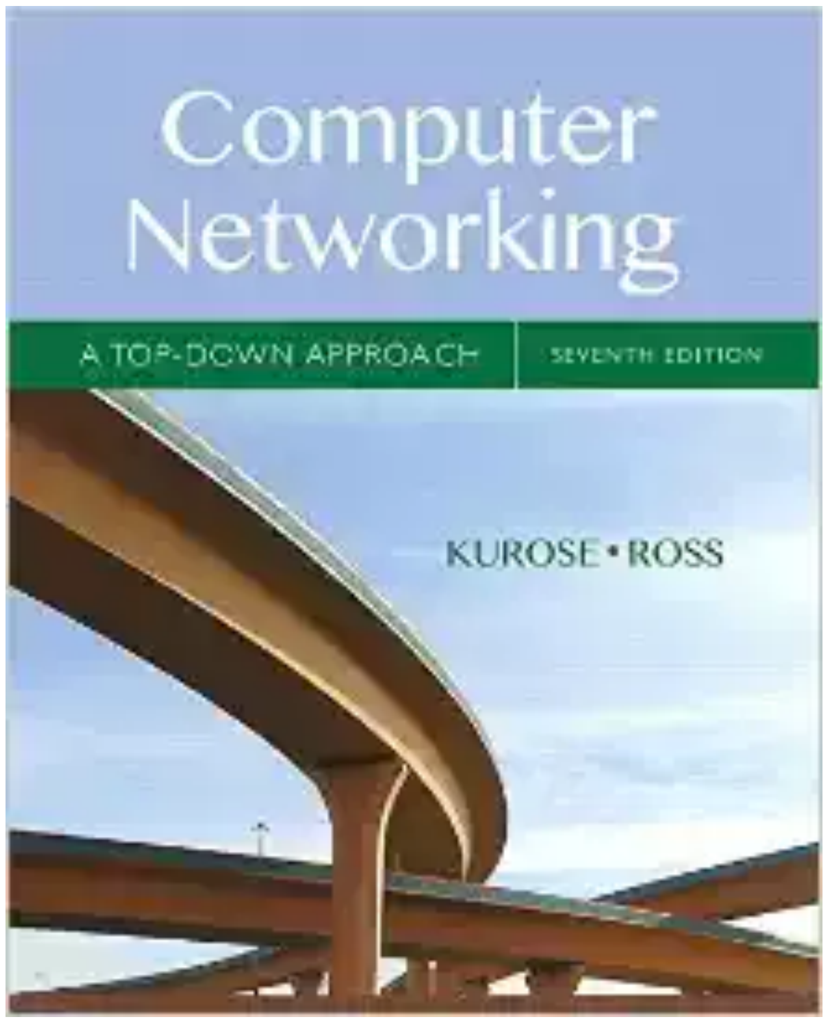


SP provides "return address"

More Complex Demux-ing



Reading Along ...



- 3.3: Connectionless Transport: UDP

UDP: User Datagram Protocol [RFC 768]

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

UDP: User Datagram Protocol [RFC 768]

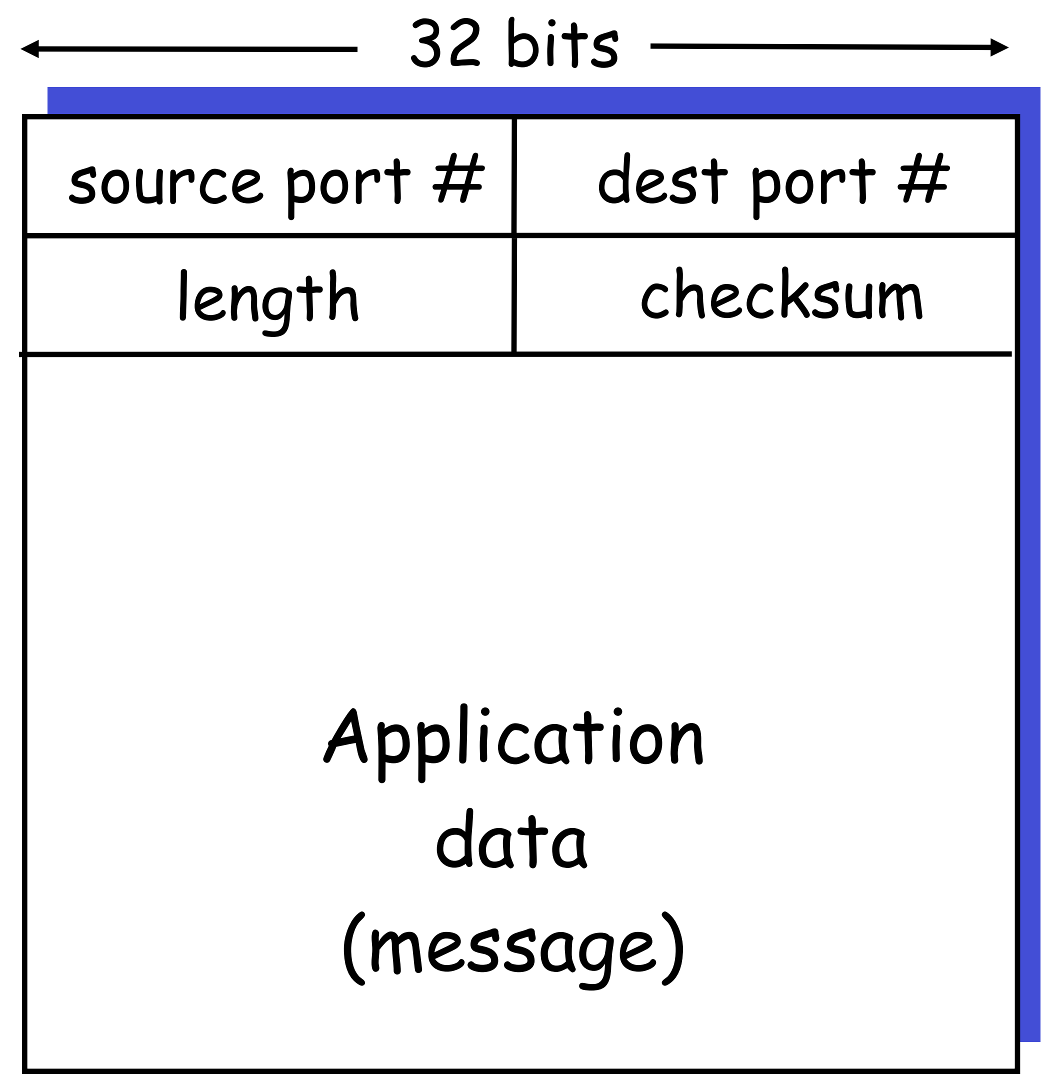
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP

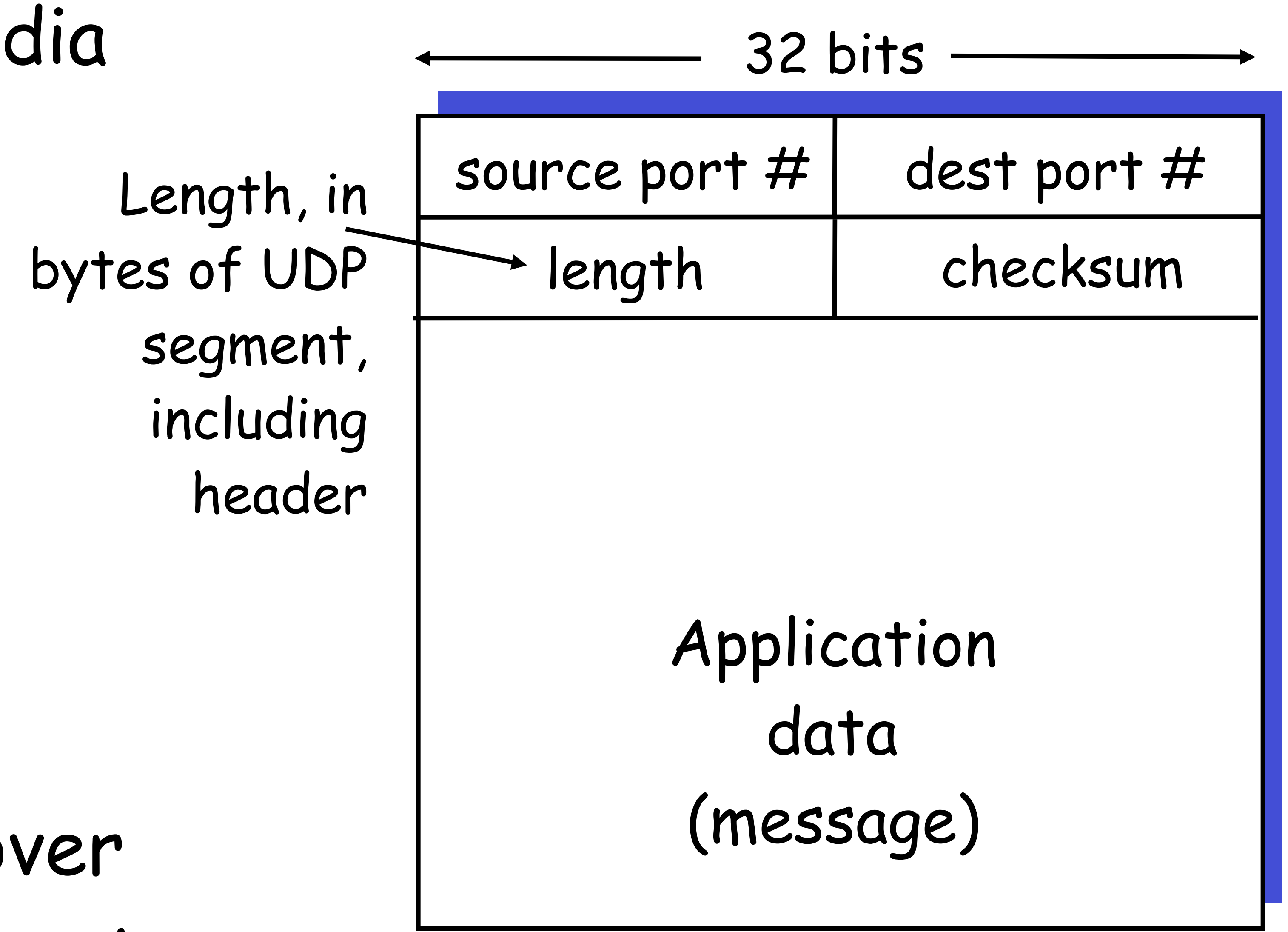
- often used for streaming multimedia apps
  - loss tolerant
  - rate controllable
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format

UDP

- often used for streaming multimedia apps
  - loss tolerant
  - rate controllable
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format

UDP checksum

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.
    - But maybe errors nonetheless?

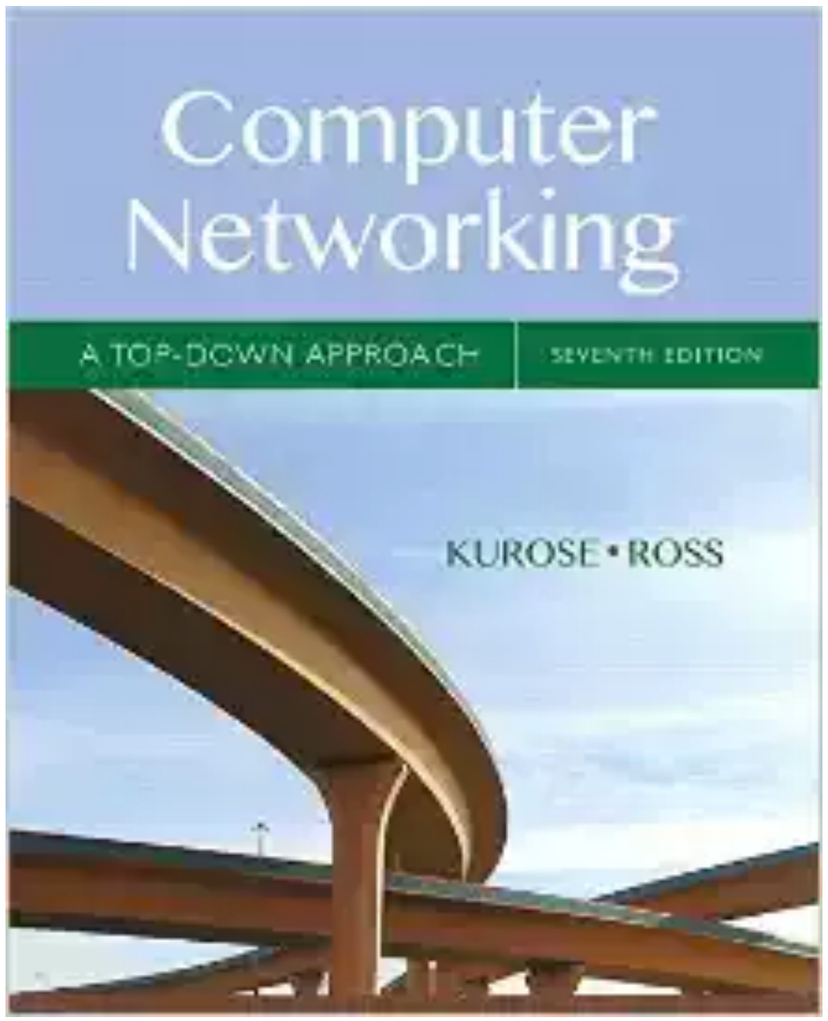
UDP Packet Header

0 ... 15	16 ... 31
Source Port Number(16 bits)	Destination Port Number(16 bits)
Length(UDP Header + Data)16 bits	UDP Checksum(16 bits)
Application Data (Message)	

A Tangent...

- Binary number: 0111

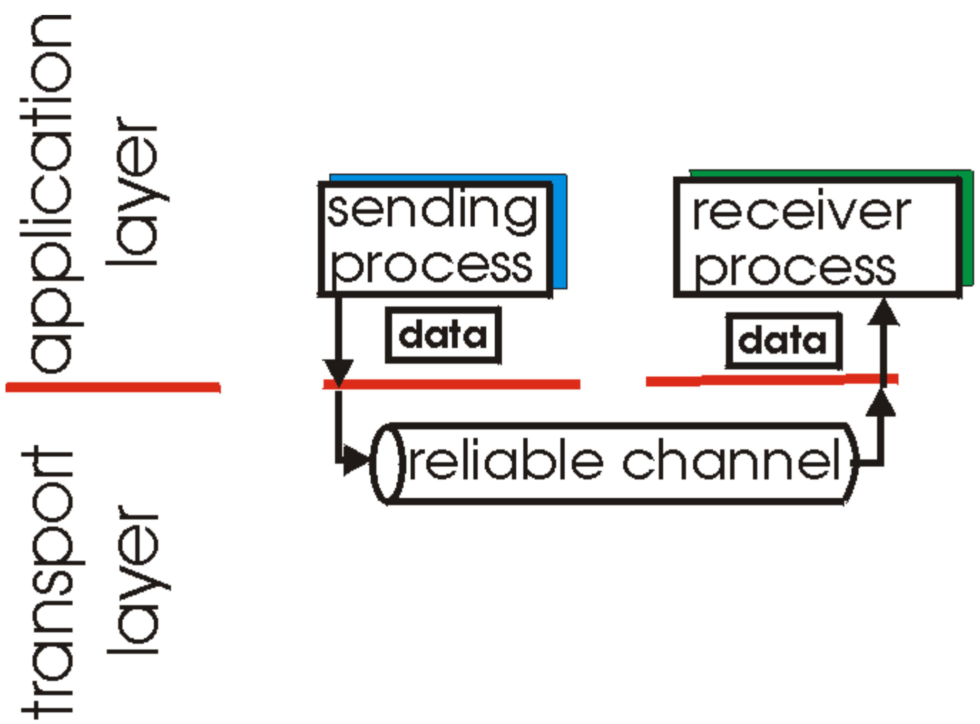
Reading Along ...



- 3.4: Principles of reliable data transfer

Principles of Reliable data transfer

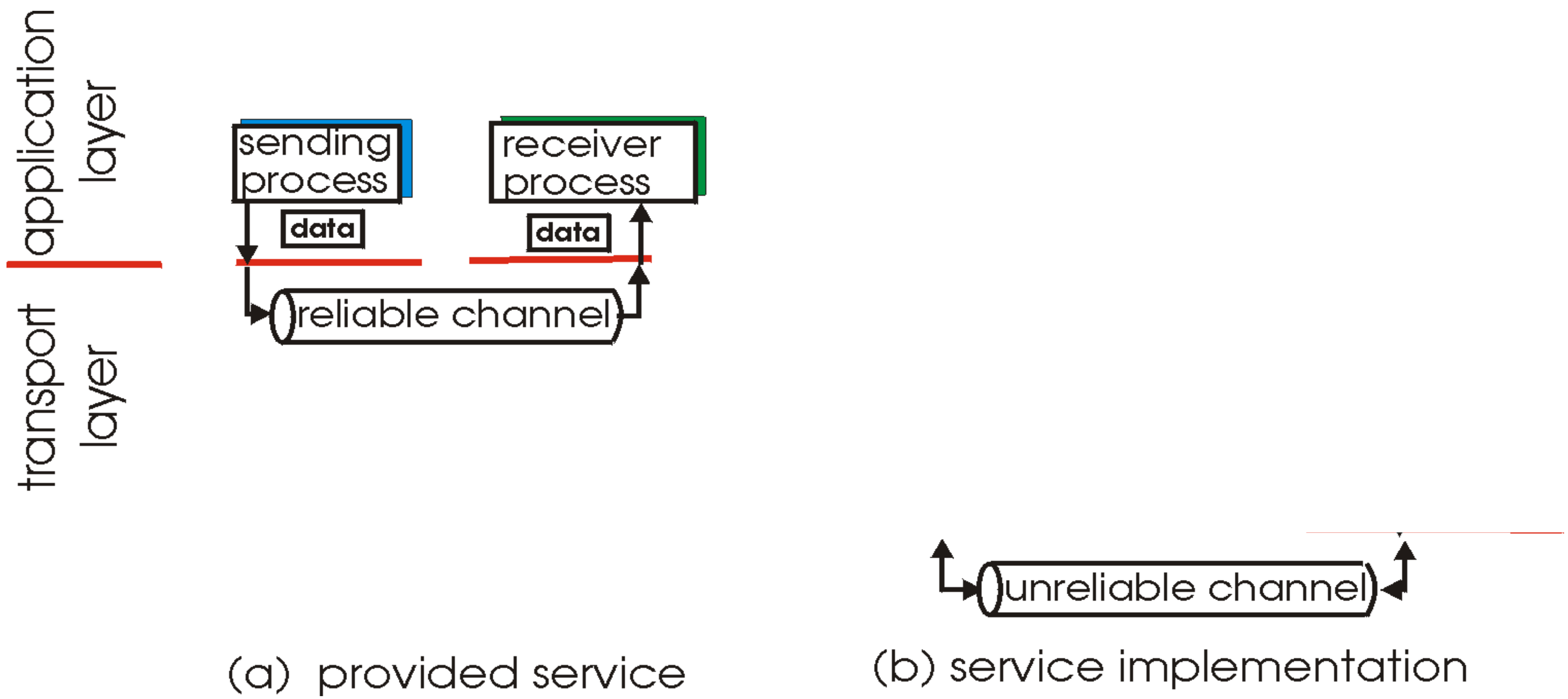
- important in app., transport, link layers
- top-10 list of important networking topics!



(a) provided service

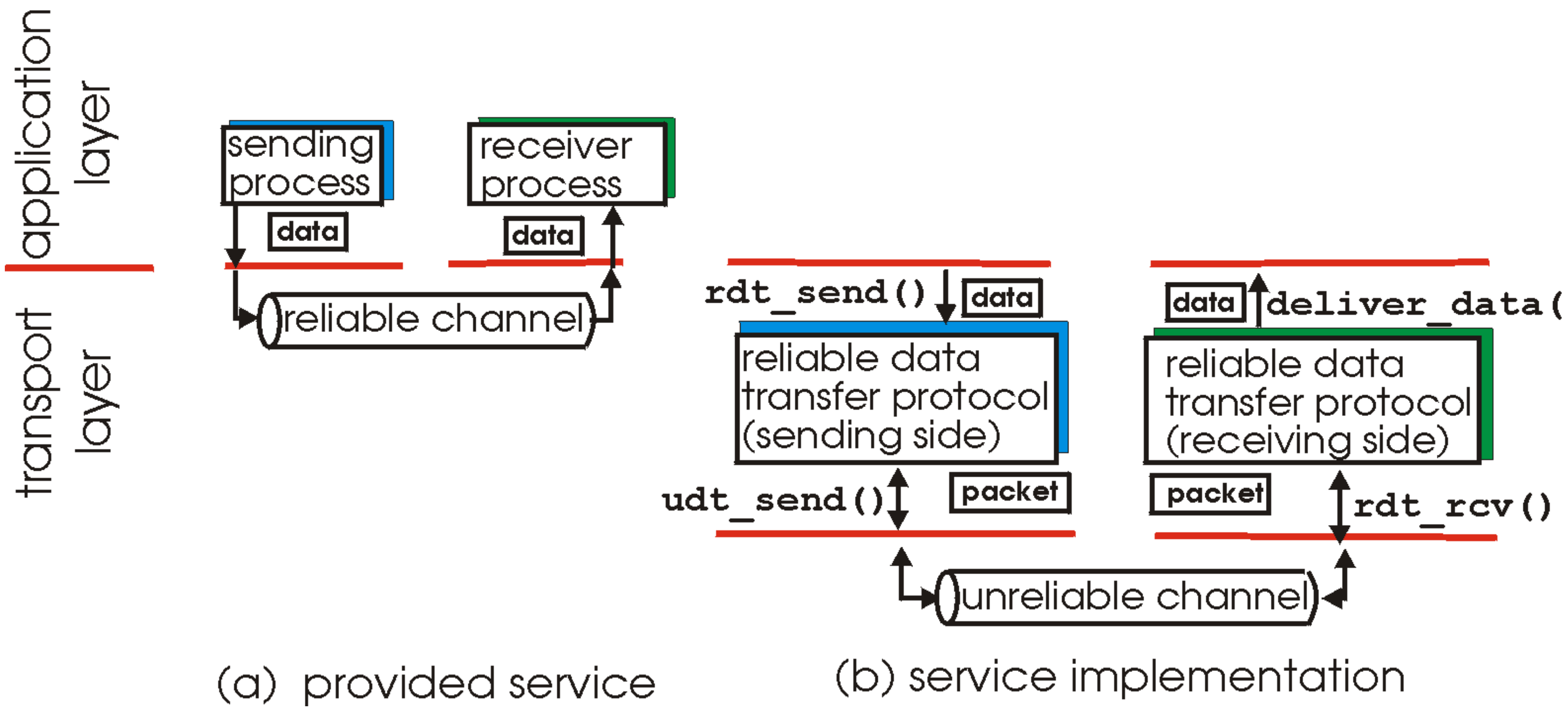
Principles of Reliable data transfer

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

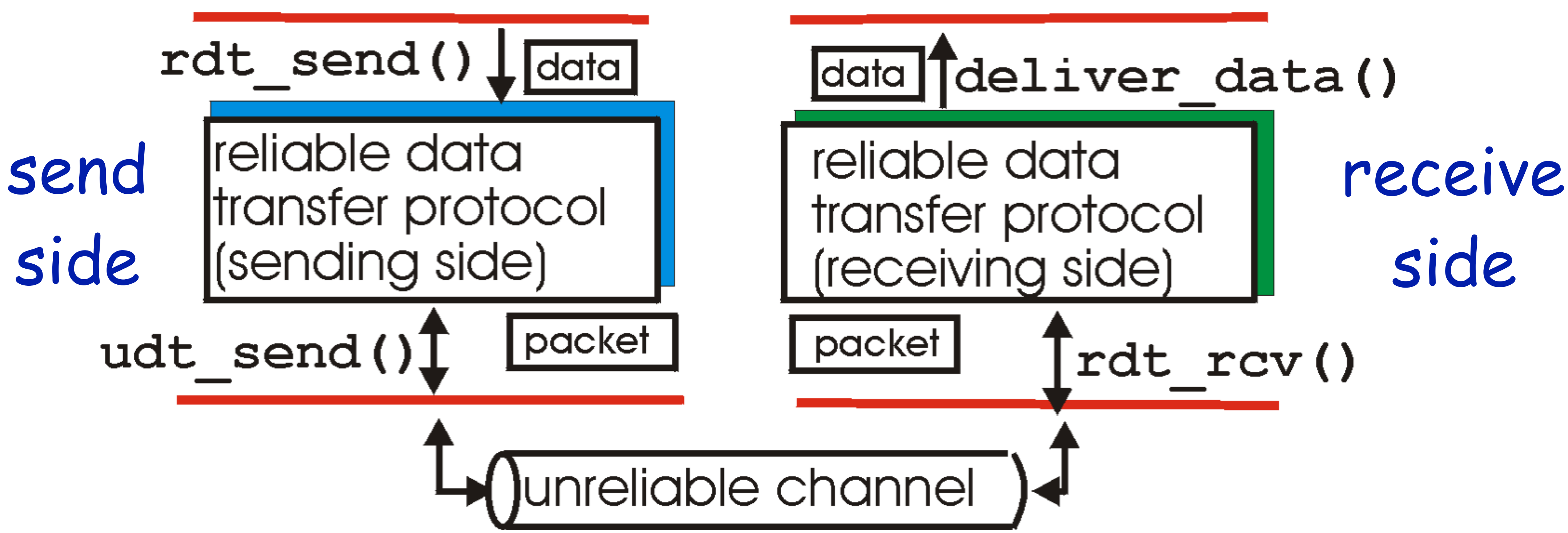


Principles of Reliable data transfer

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

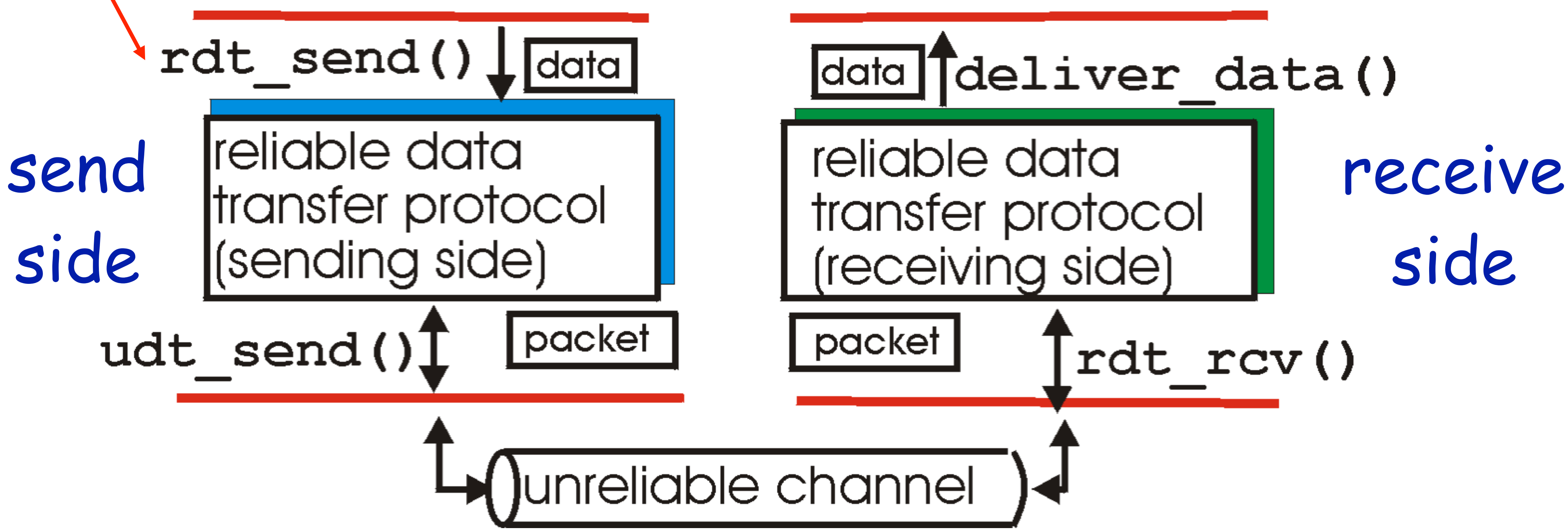


Reliable data transfer: getting started



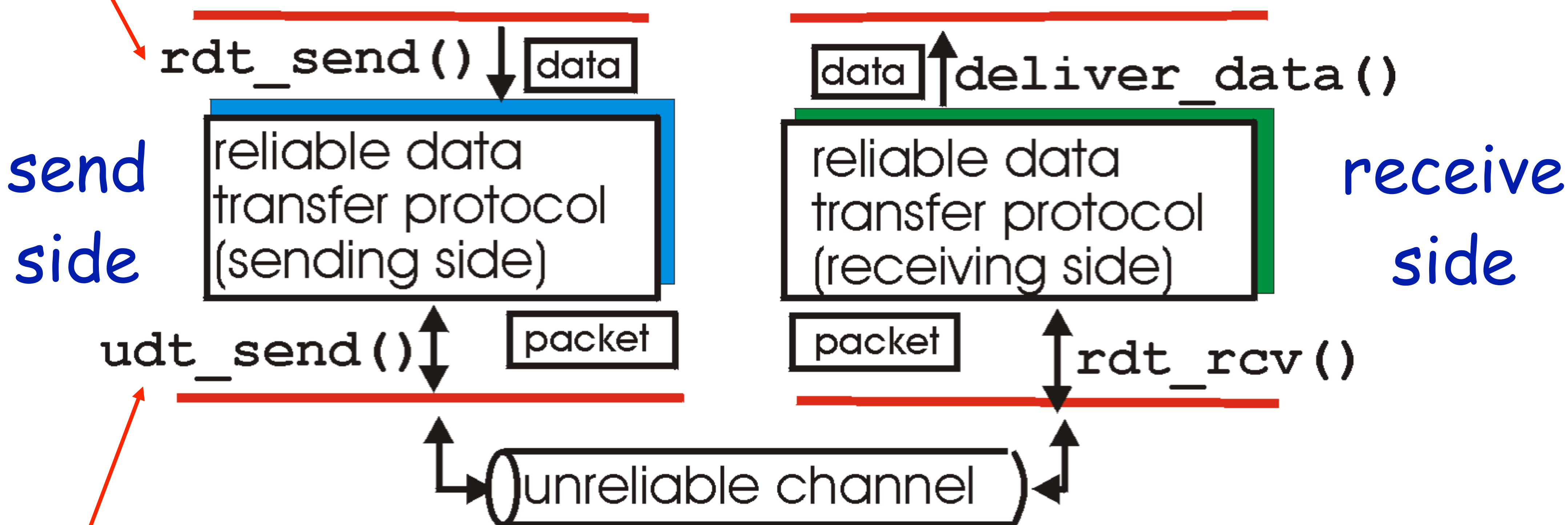
Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



Reliable data transfer: getting started

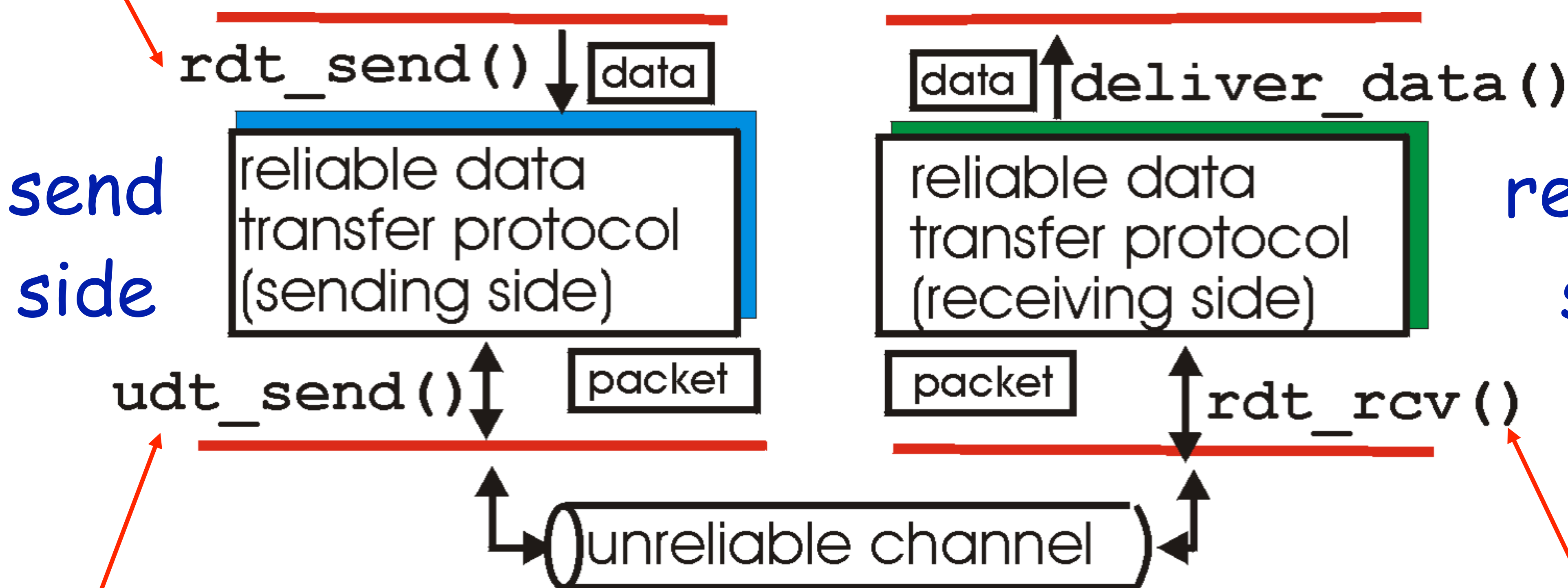
rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



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

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



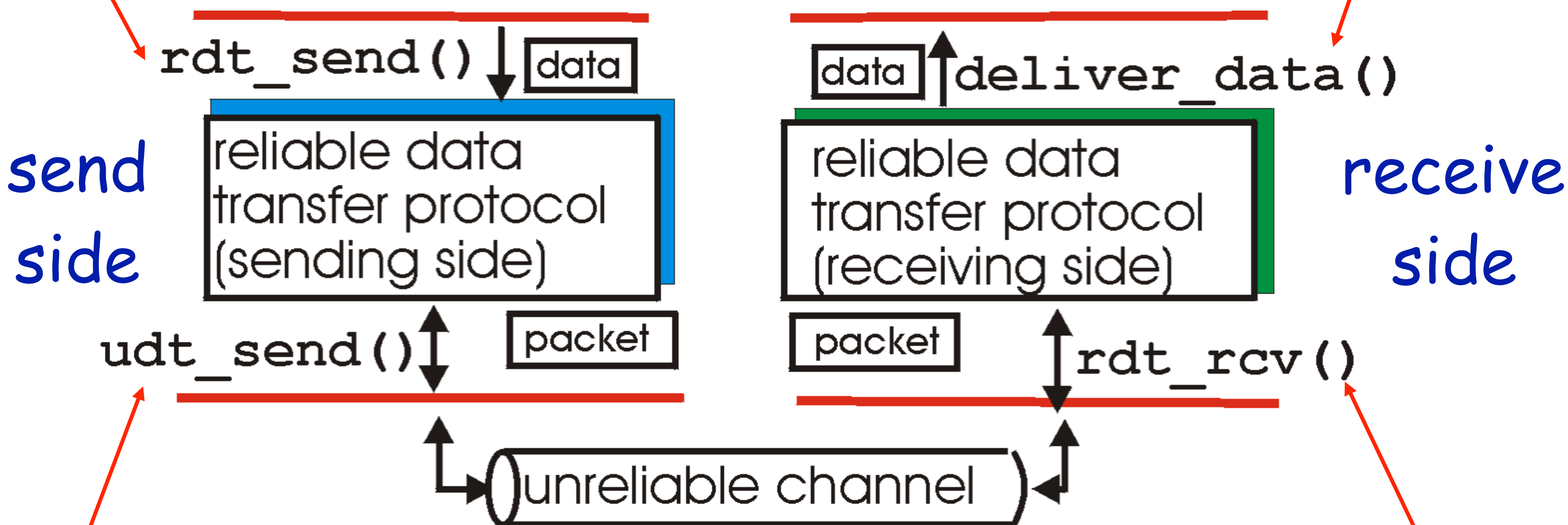
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

`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



`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

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 in both directions!
- use finite state machines (FSM) to specify sender, receiver

Reliable data transfer: getting started

We'll:

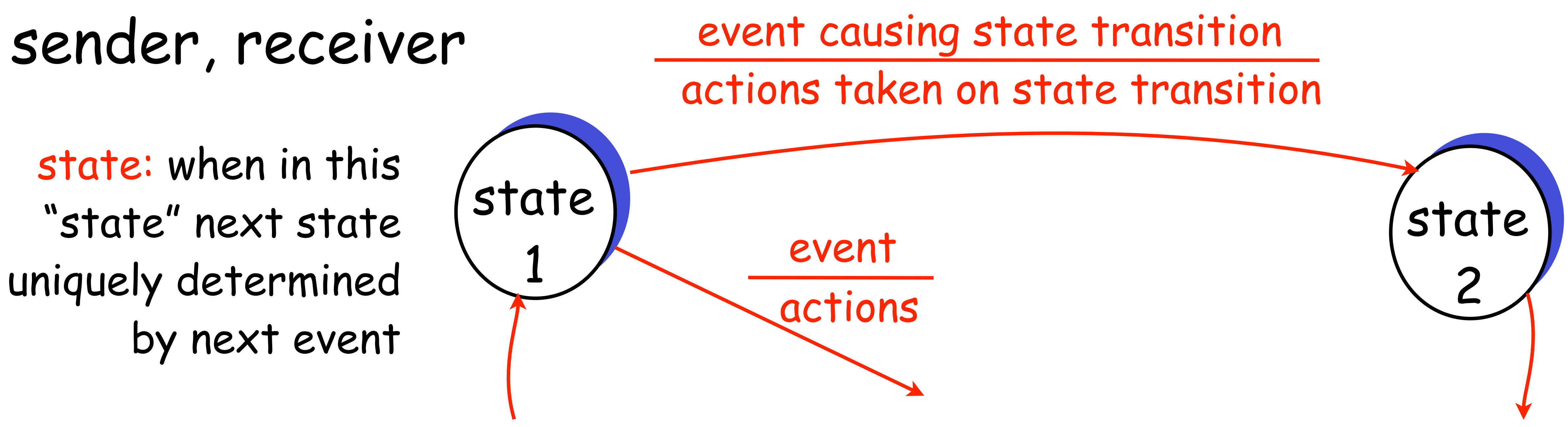
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver

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

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 in both directions!
- use finite state machines (FSM) to specify sender, receiver



Rdt1.0: reliable transfer over a reliable channel

Rdt1.0: reliable transfer over a reliable channel

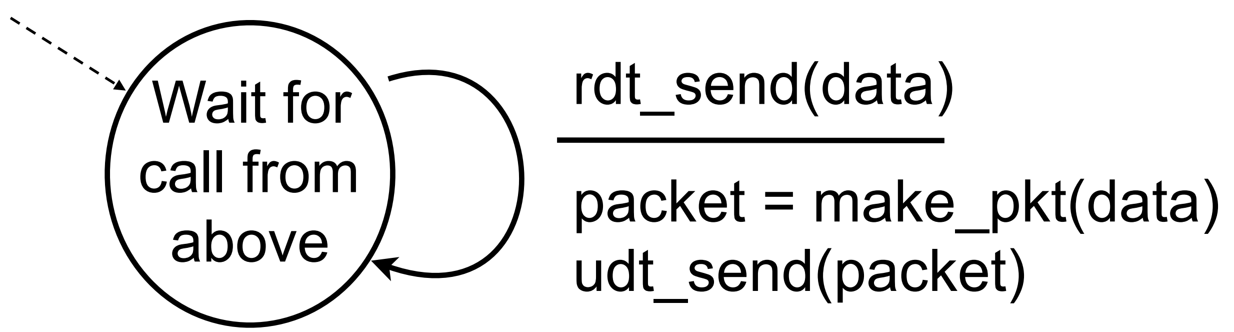
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

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 read data from underlying channel

Rdt1.0: reliable transfer over a reliable channel

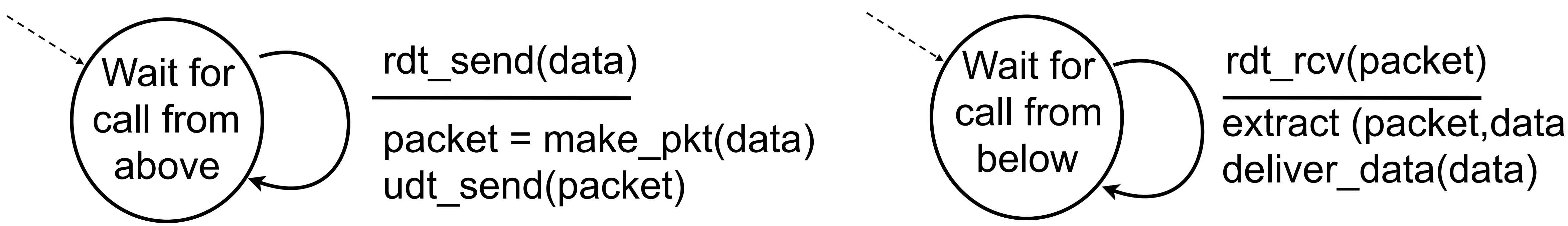
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



sender

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 read data from underlying channel



sender　　　　receiver

Rdt 1.0

Rdt 1.0

- Obviously unrealistic assumptions!
- If that was all there was to it, there'd be little reason to even bring it up!

Rdt2.0: channel with bit errors

Rdt2.0: channel with bit errors

How do humans recover
from "errors"
during conversation?

Rdt2.0: channel with bit errors

Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors

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:

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

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

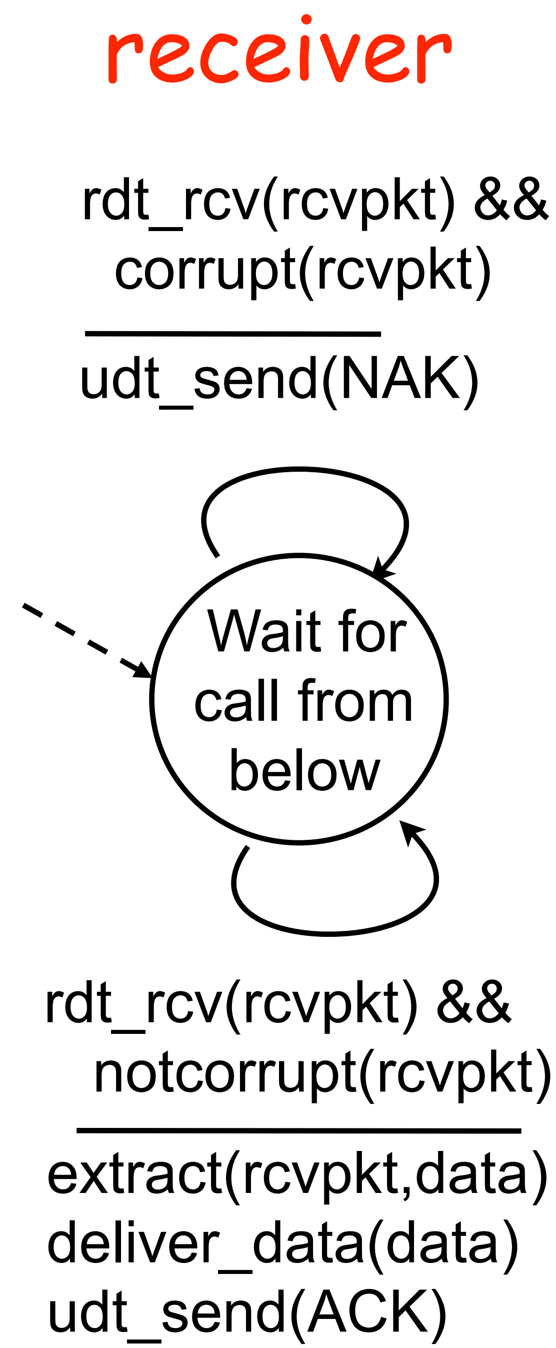
Rdt2.0: channel with bit errors

- 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

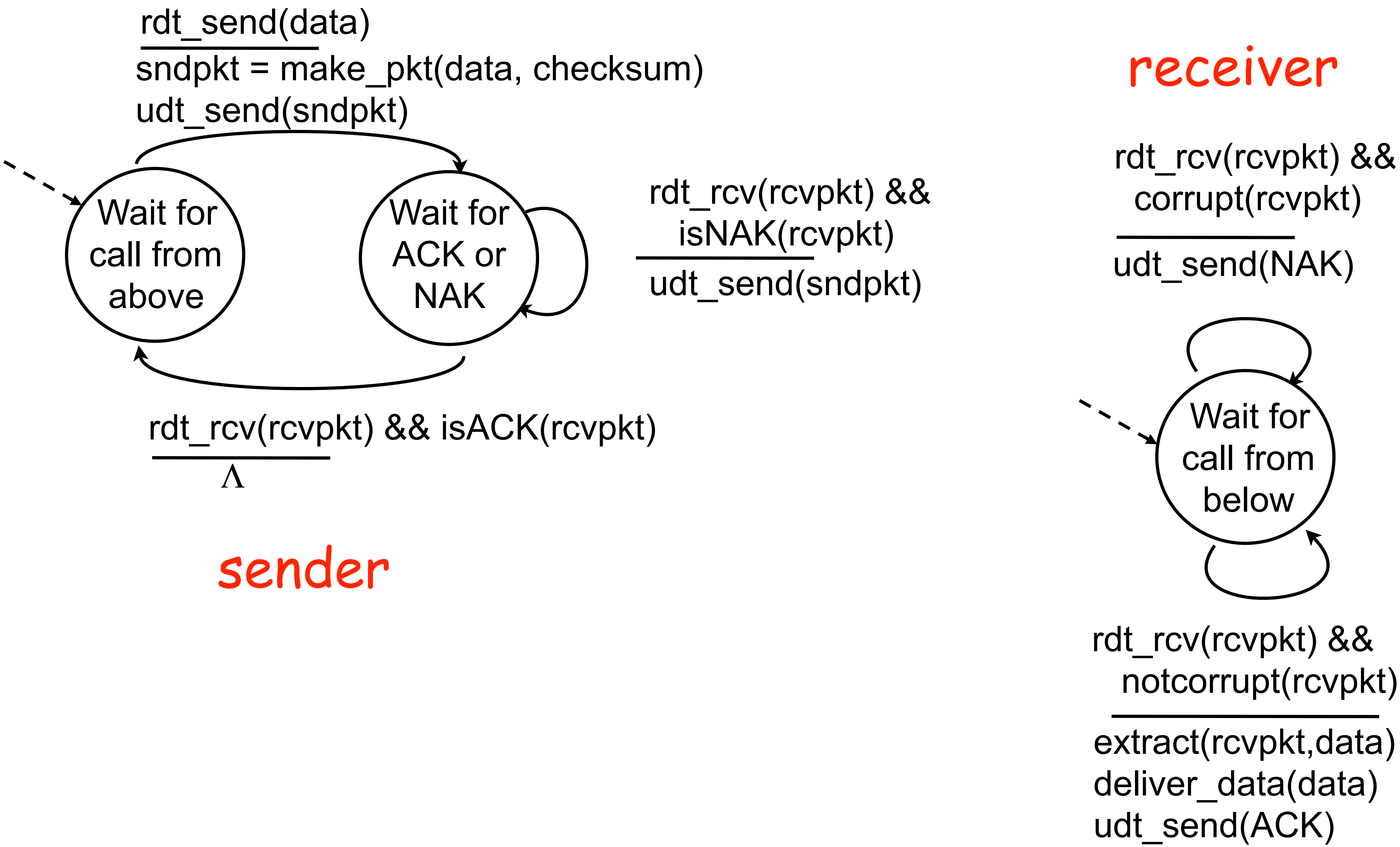
Rdt2.0: channel with bit errors

- 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
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

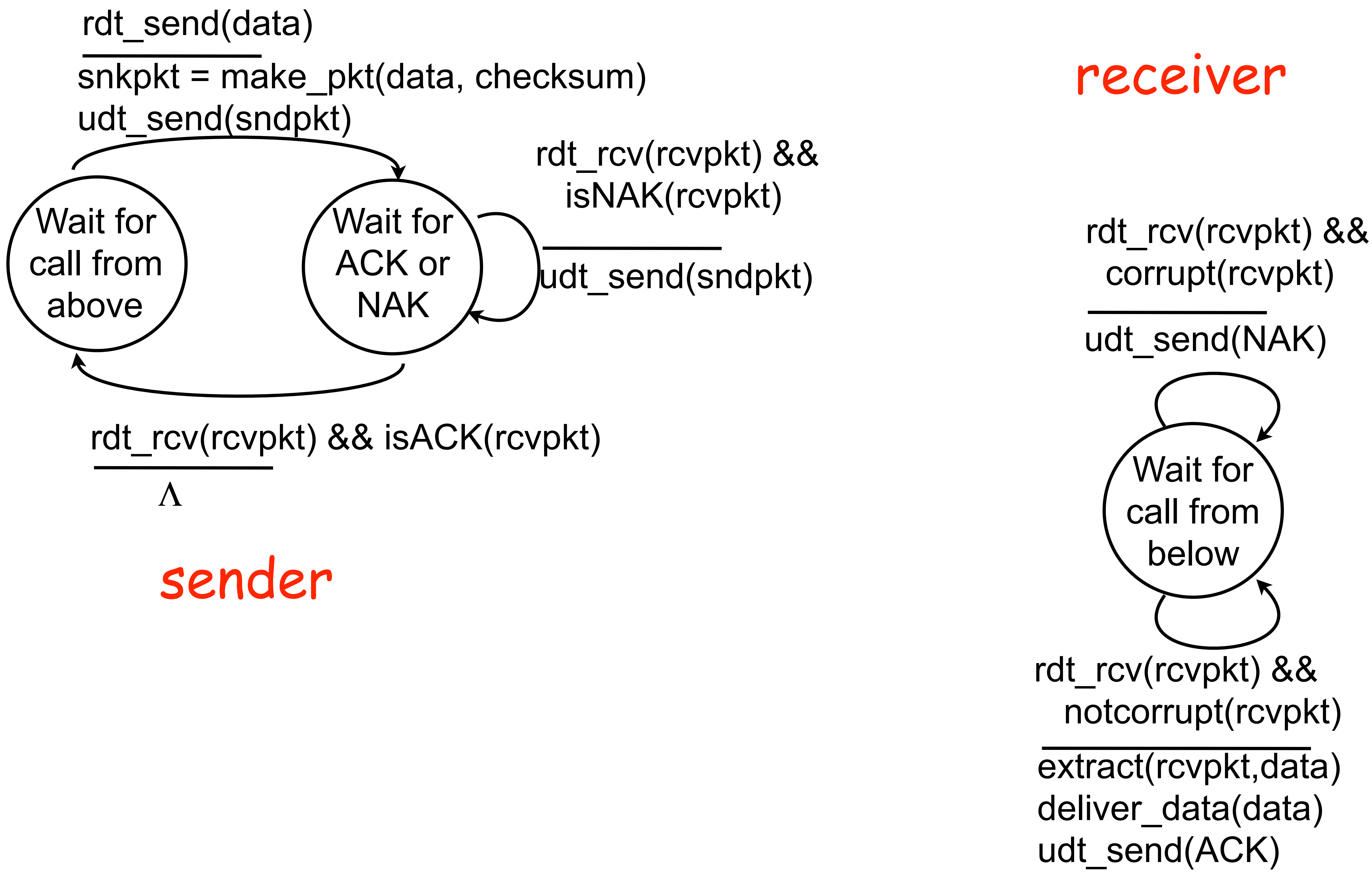
rdt2.0: FSM specification



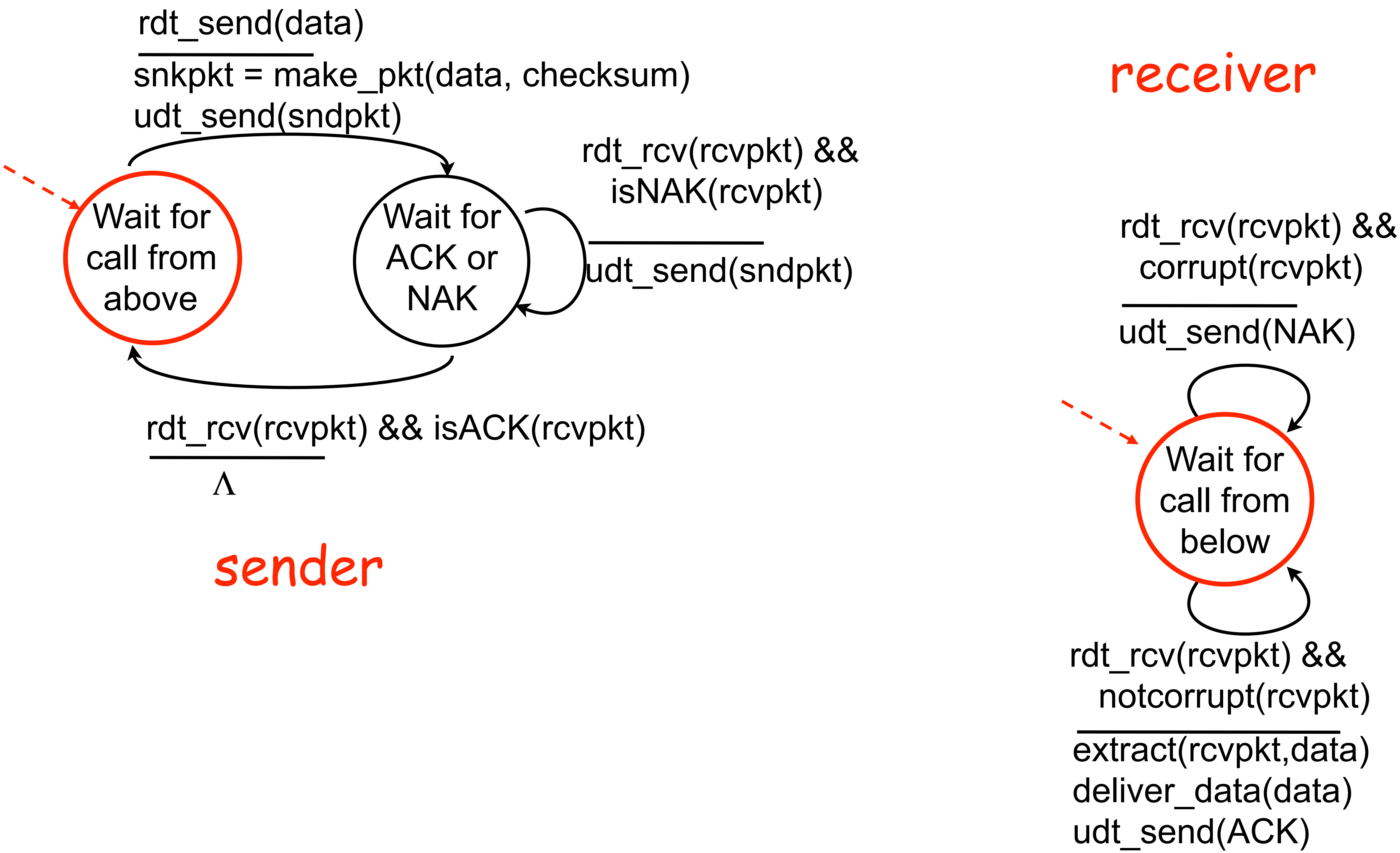
rdt2.0: FSM specification



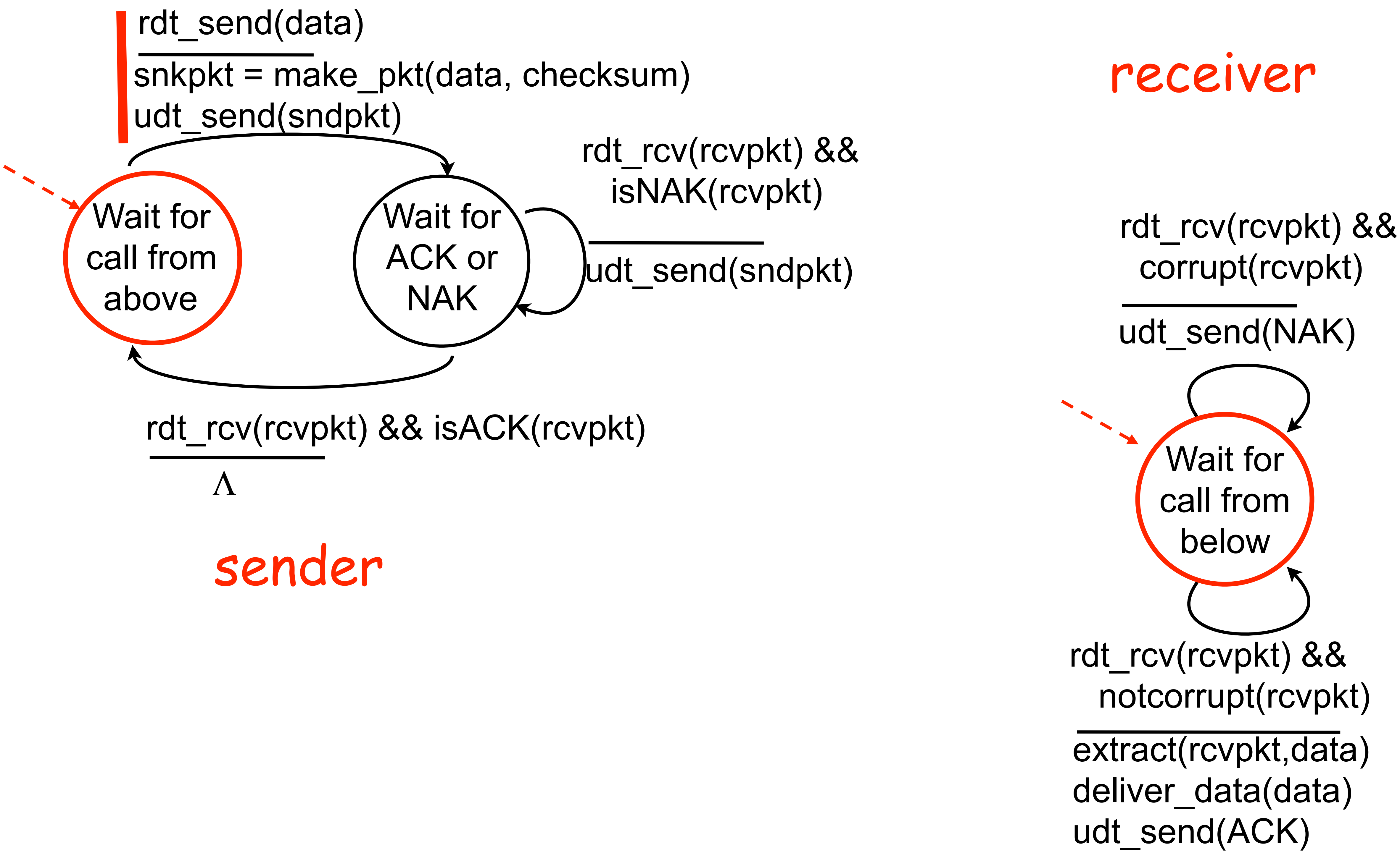
rdt2.0: operation with no errors



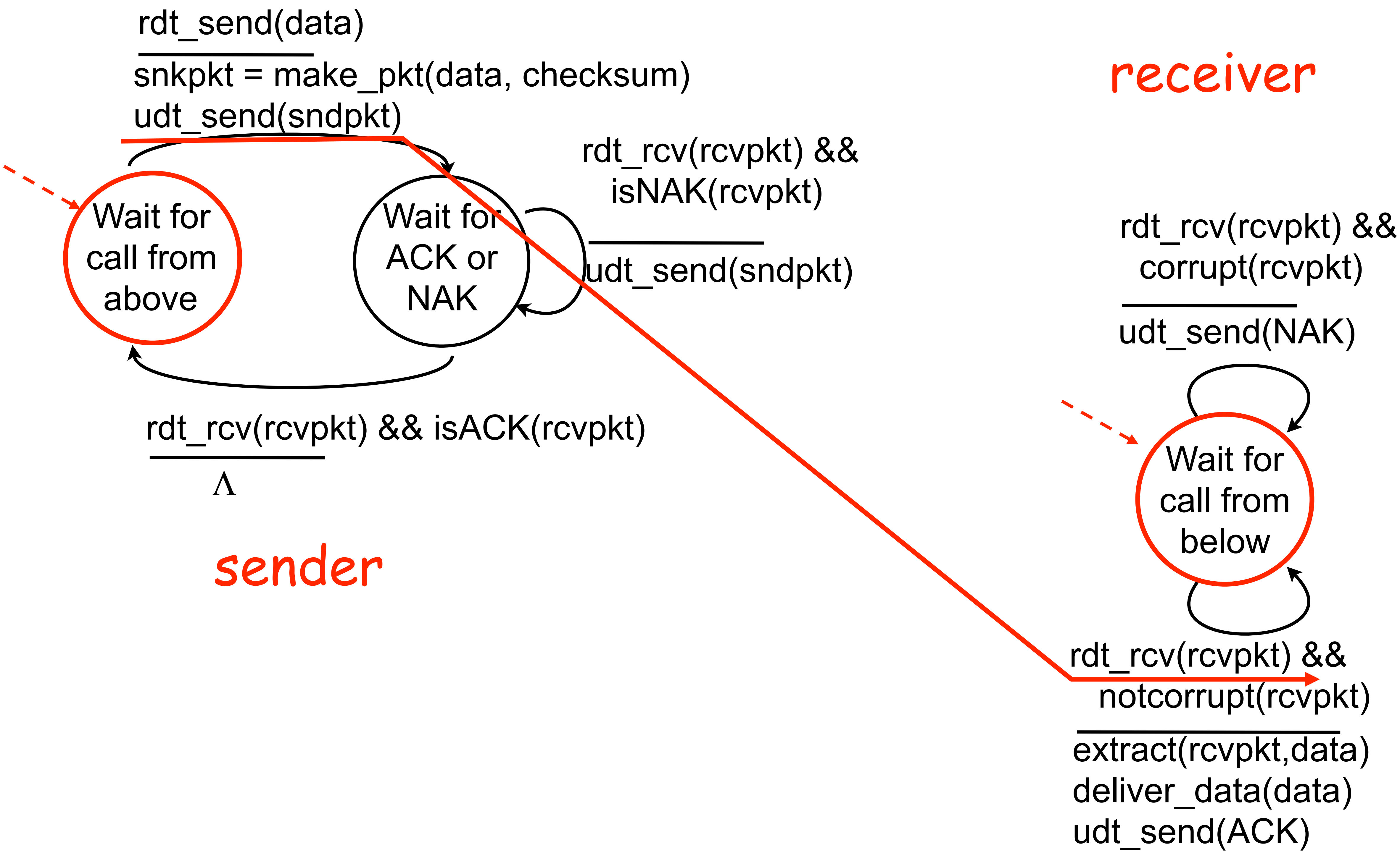
rdt2.0: operation with no errors



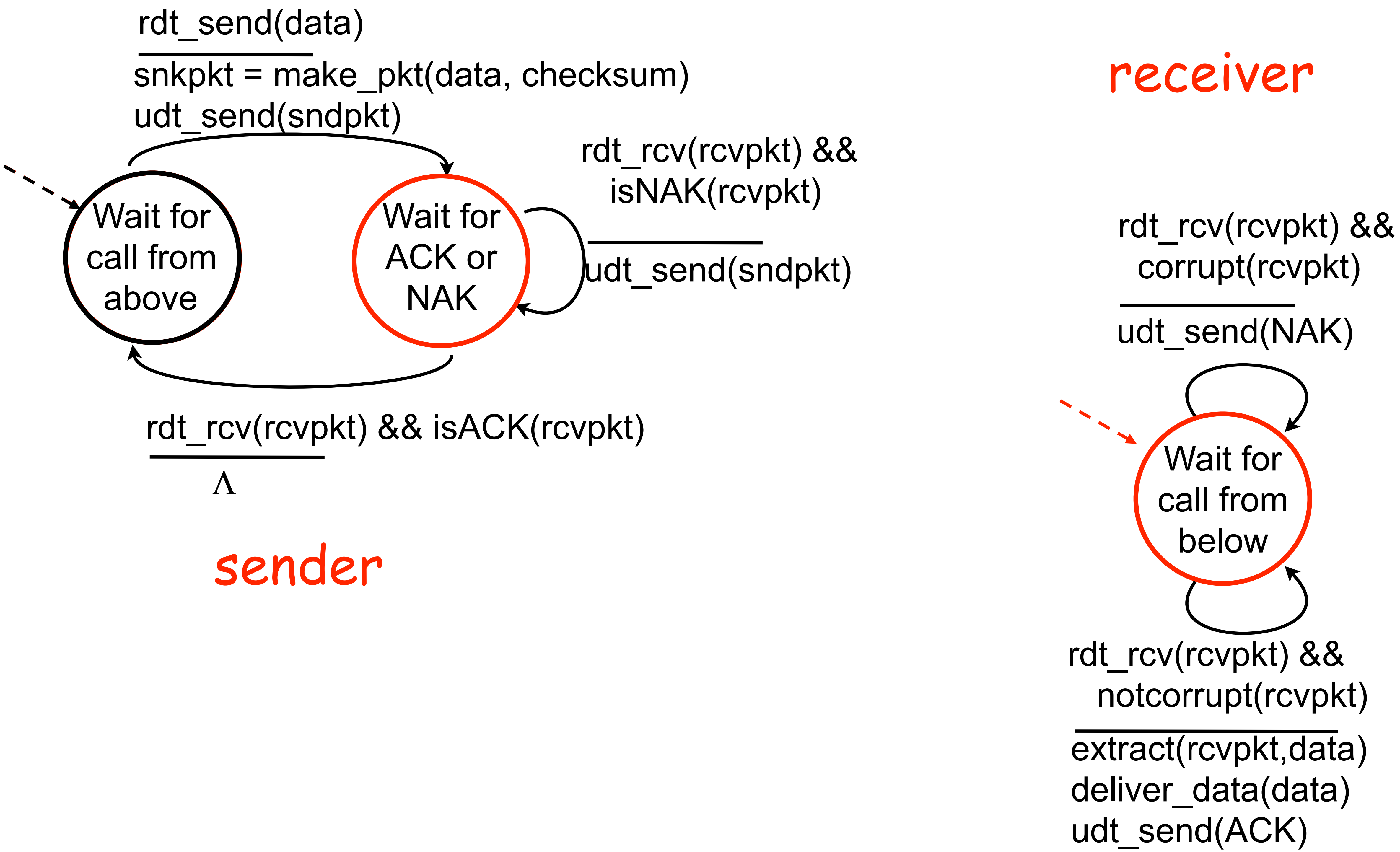
rdt2.0: operation with no errors



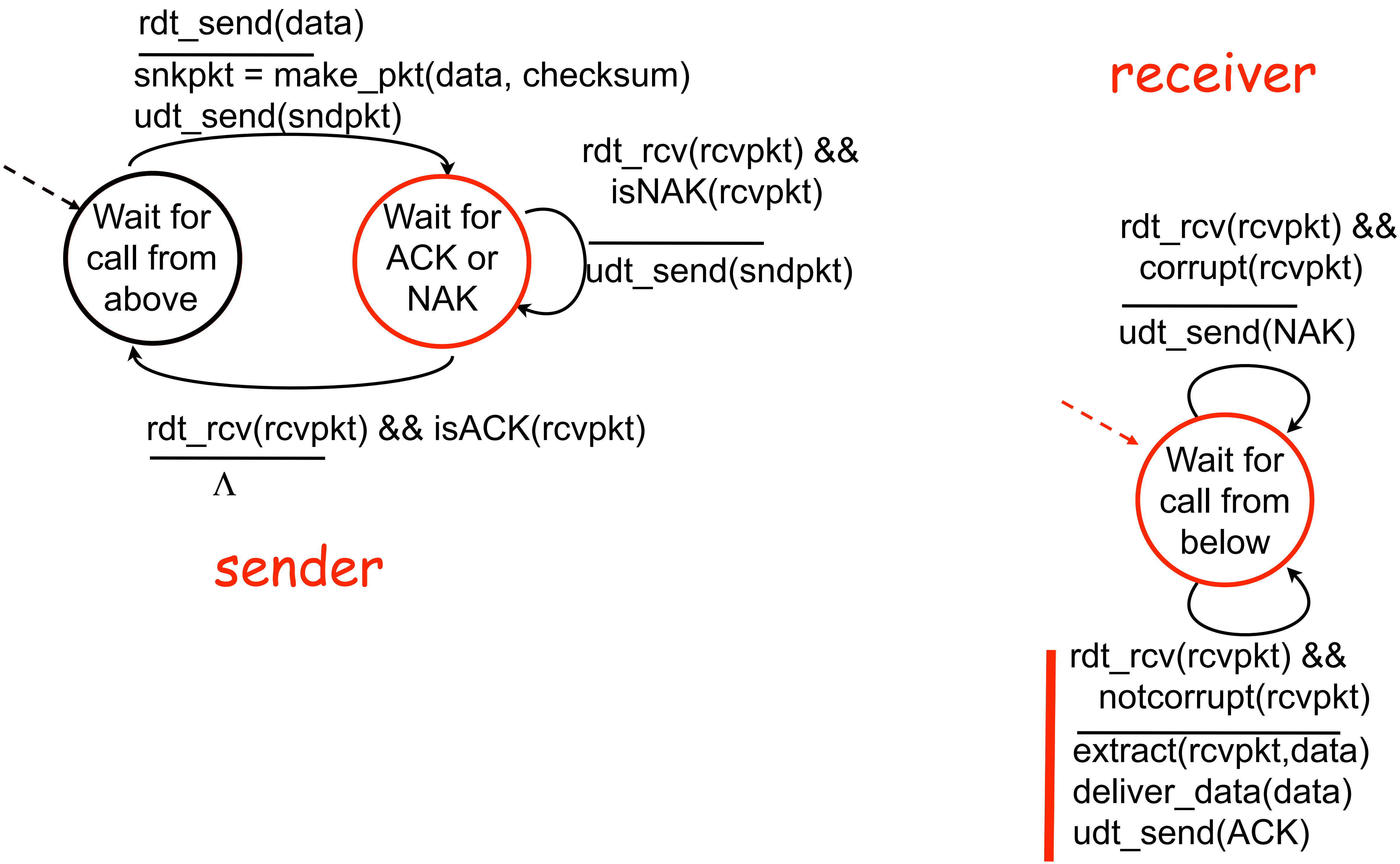
rdt2.0: operation with no errors



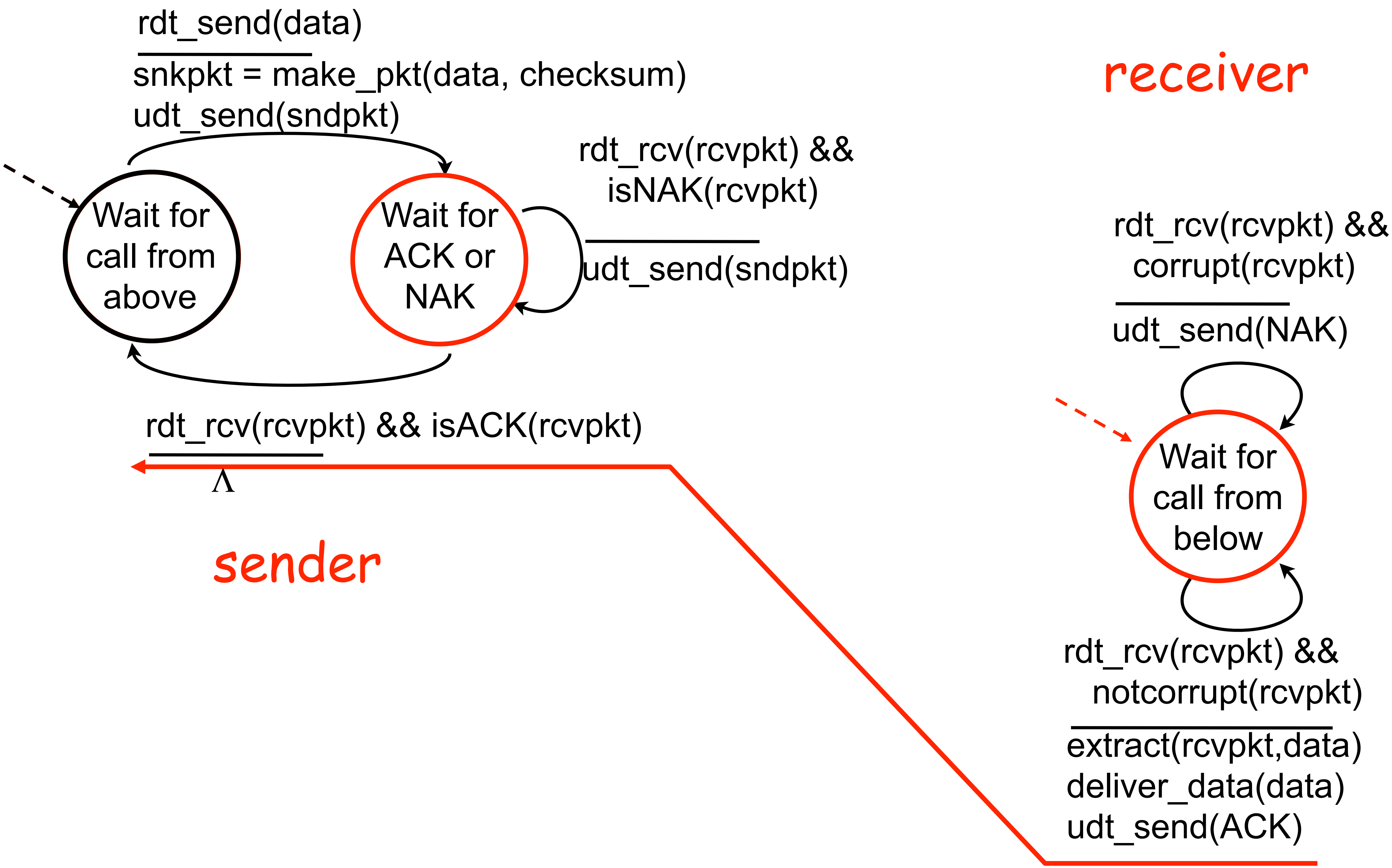
rdt2.0: operation with no errors



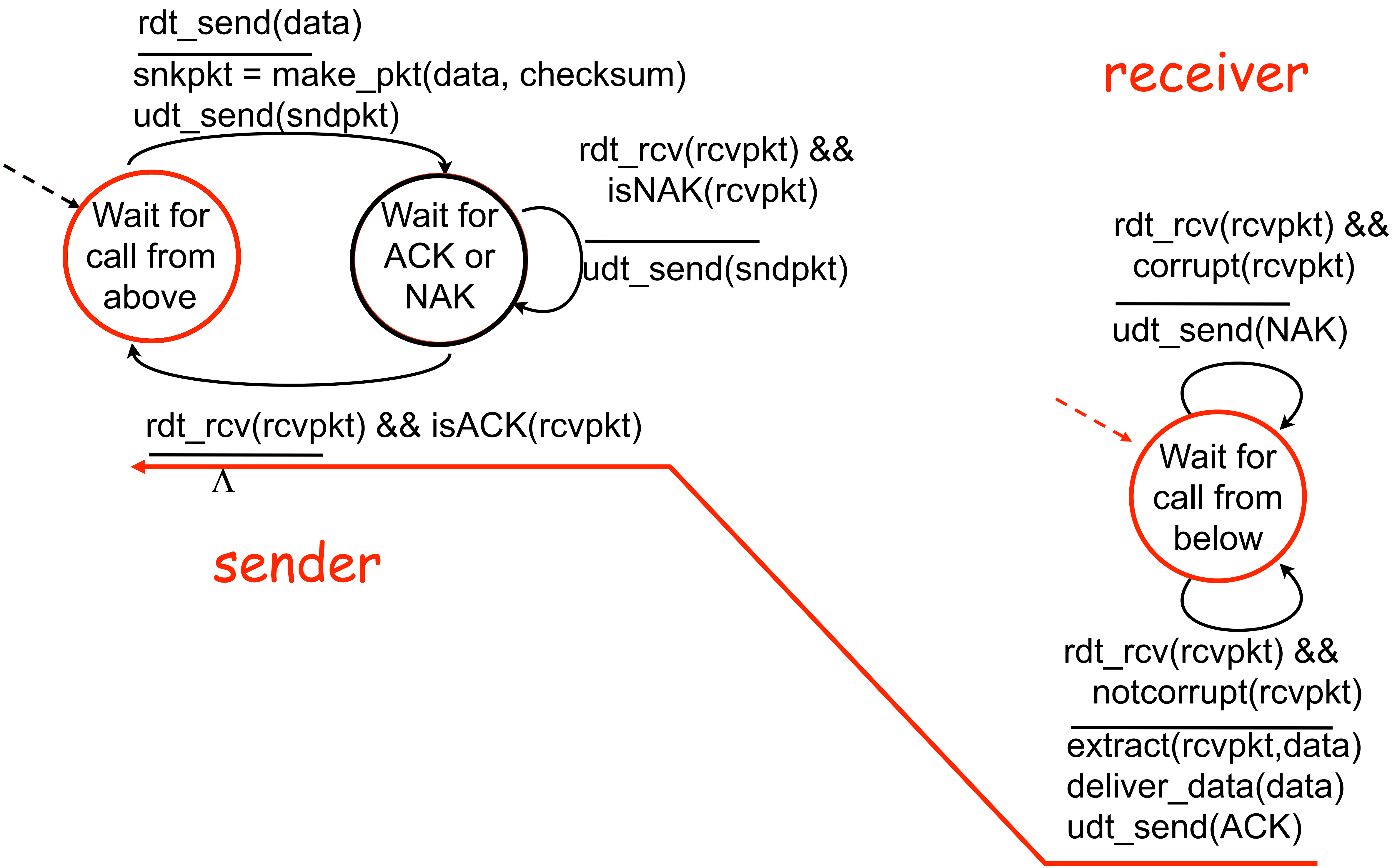
rdt2.0: operation with no errors



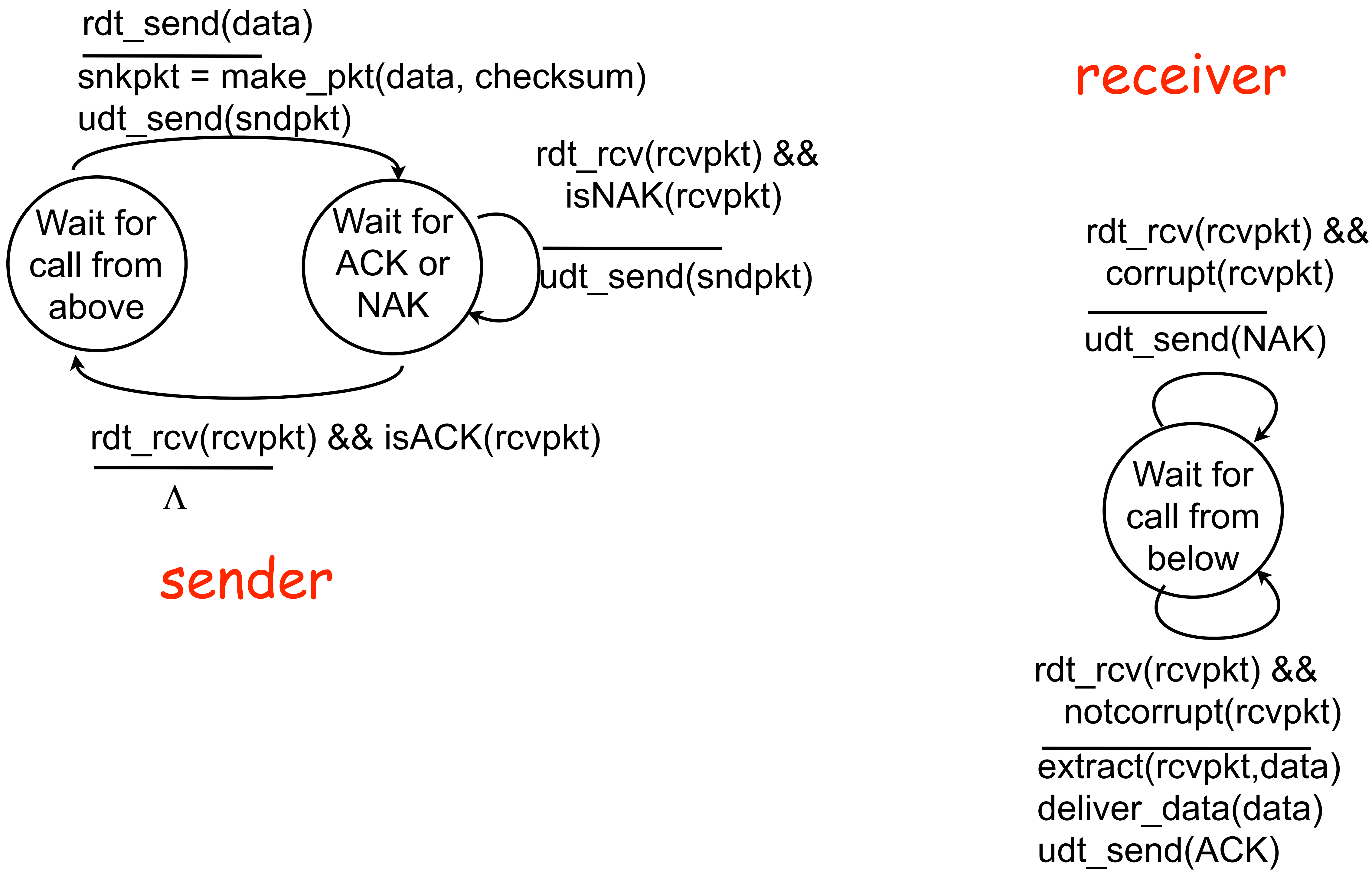
rdt2.0: operation with no errors



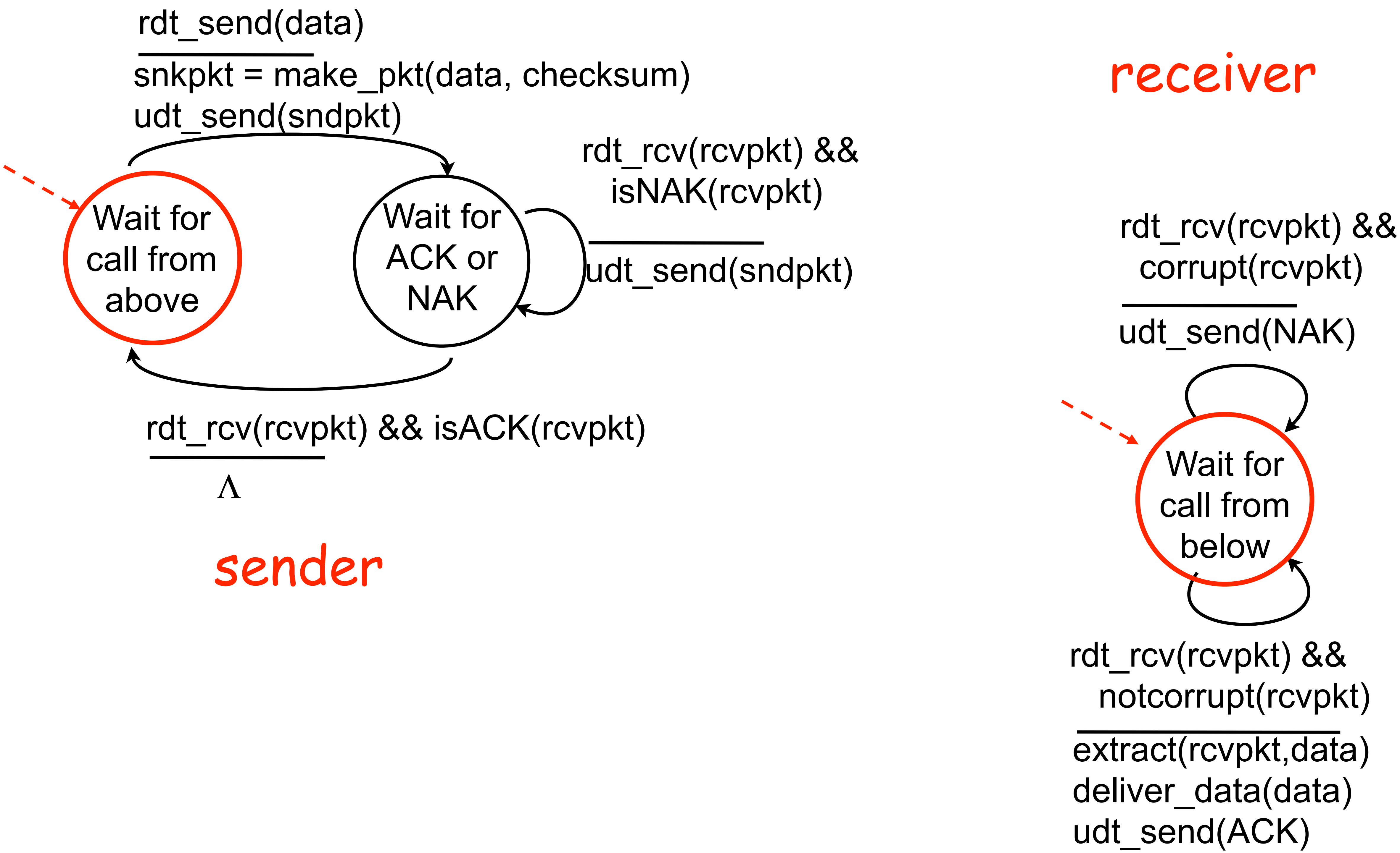
rdt2.0: operation with no errors



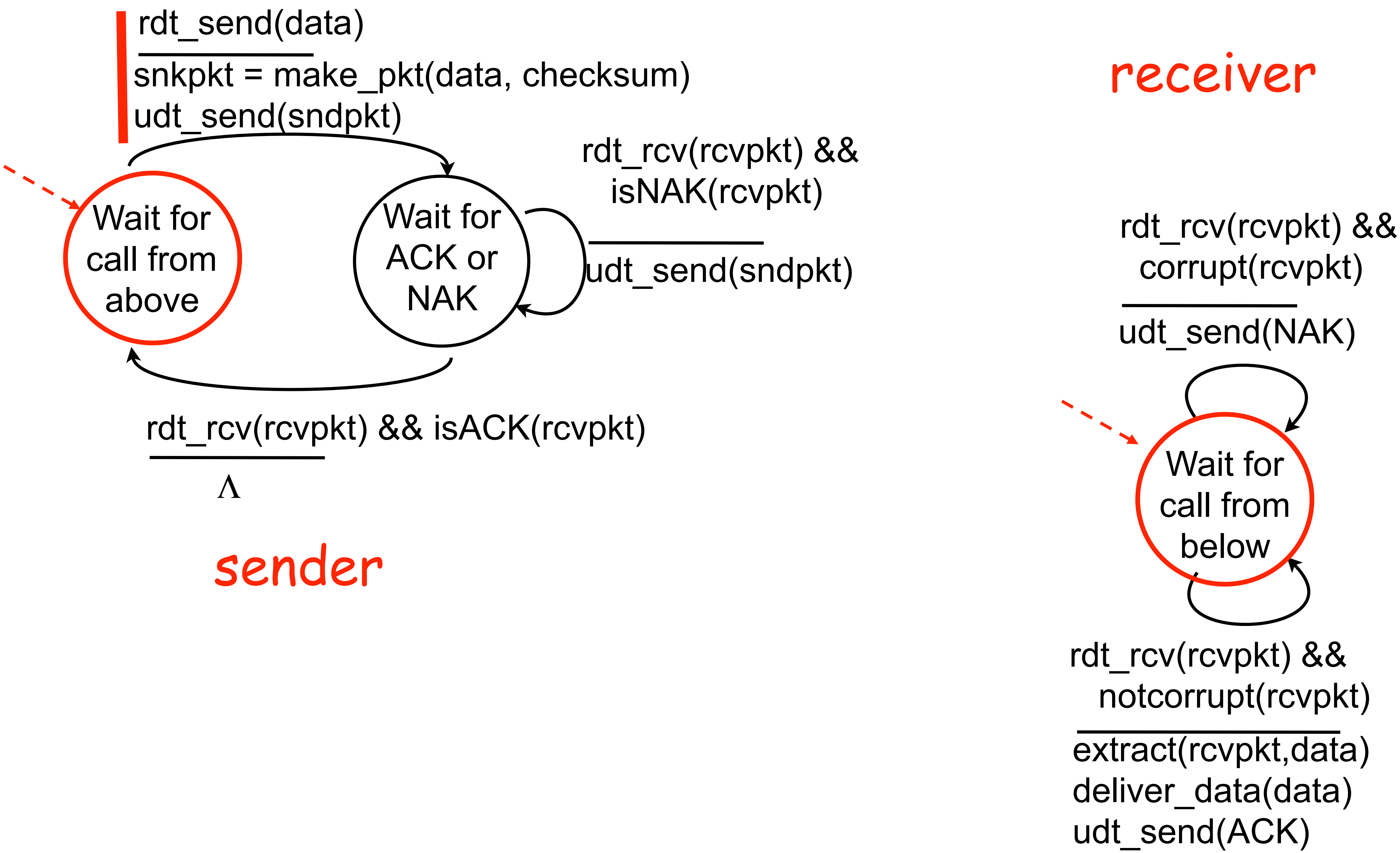
rdt2.0: error scenario



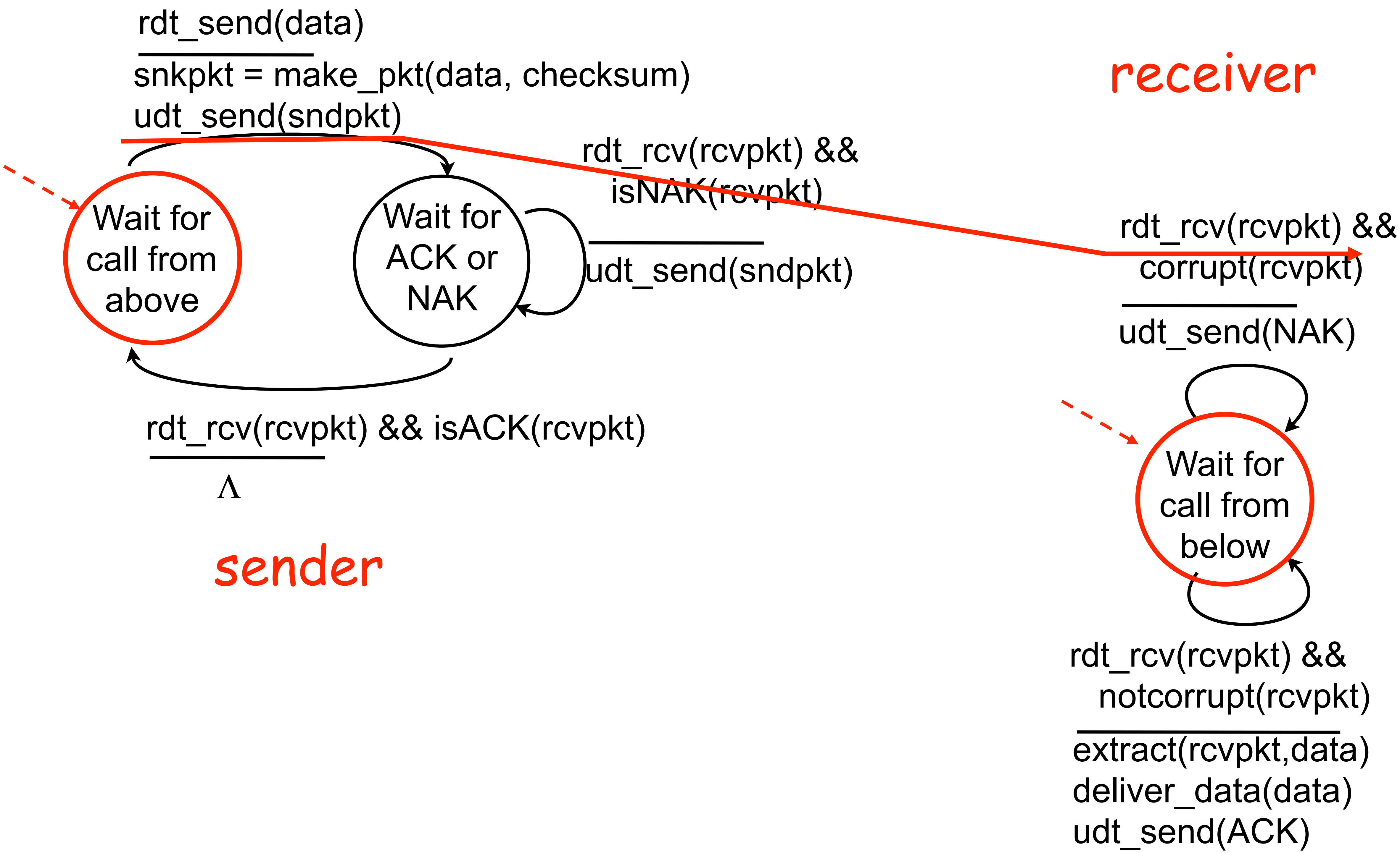
rdt2.0: error scenario



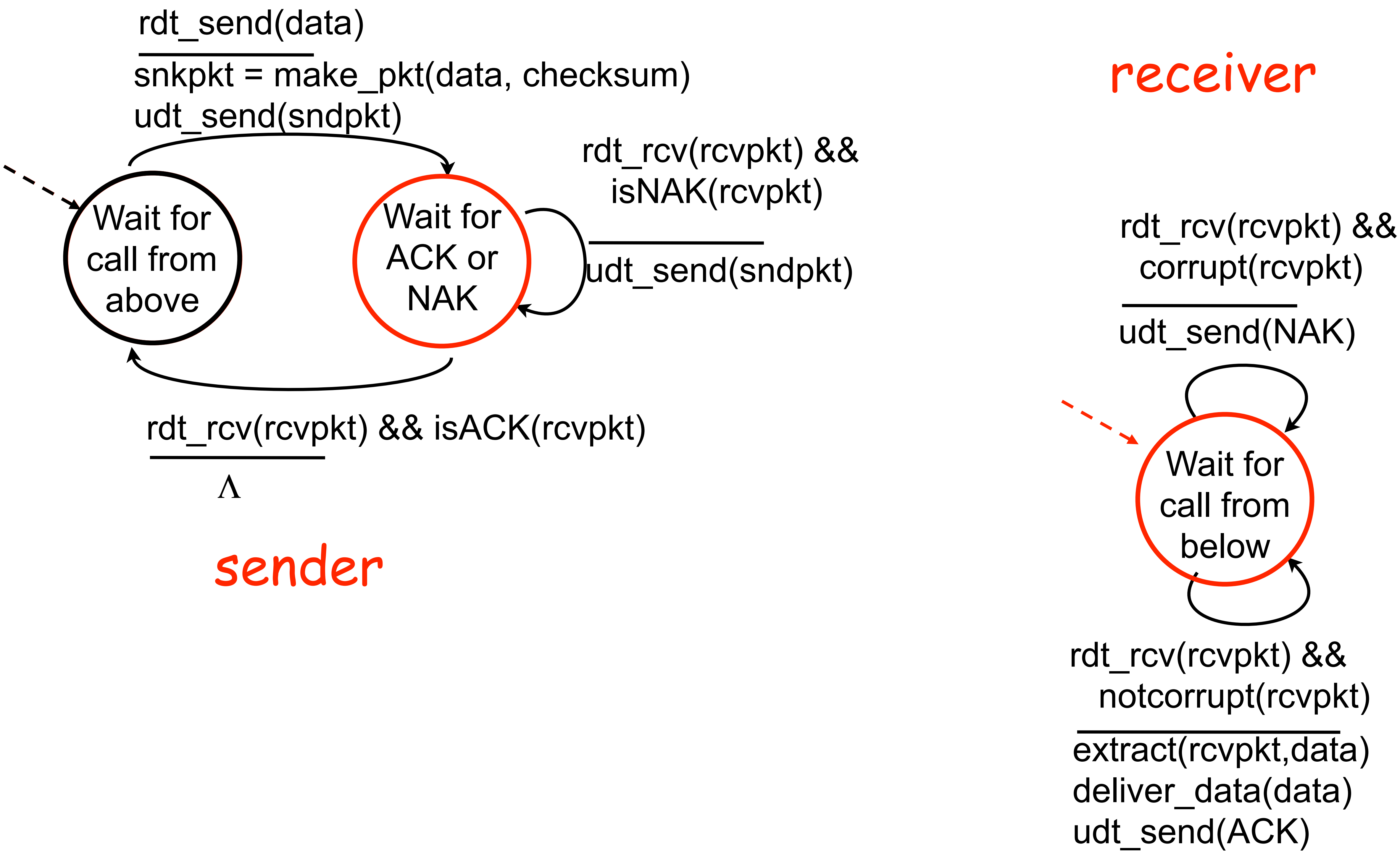
rdt2.0: error scenario



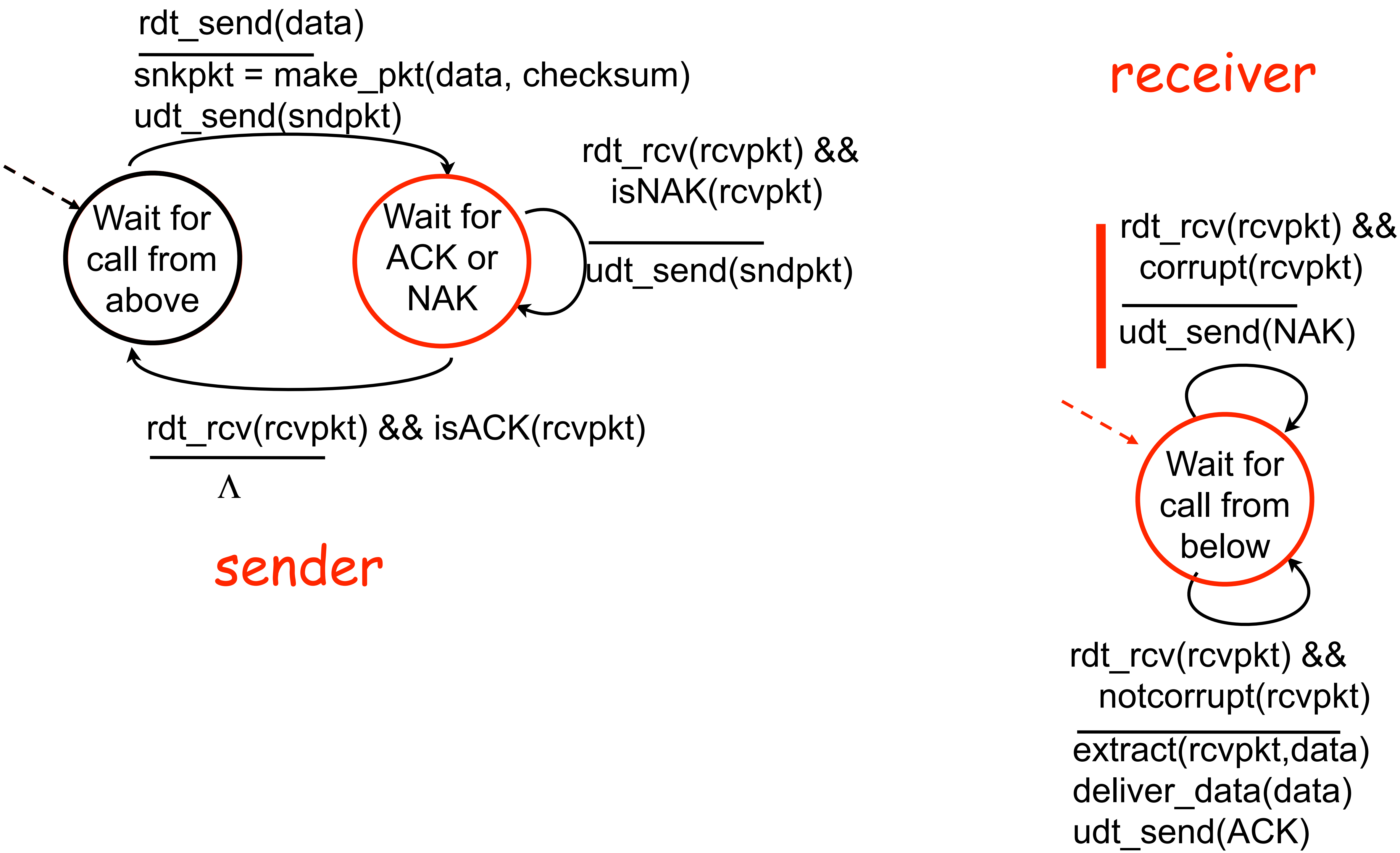
rdt2.0: error scenario



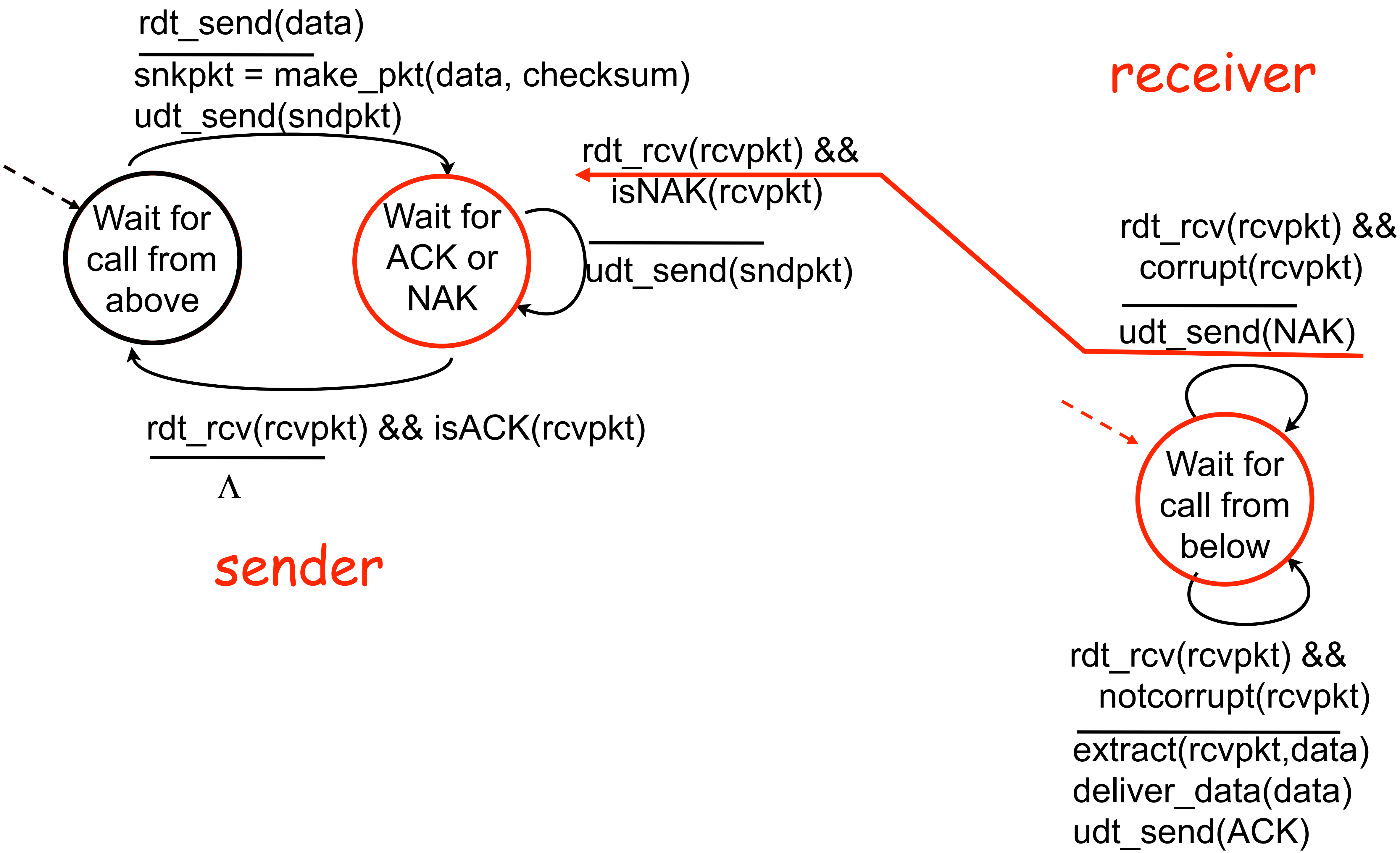
rdt2.0: error scenario



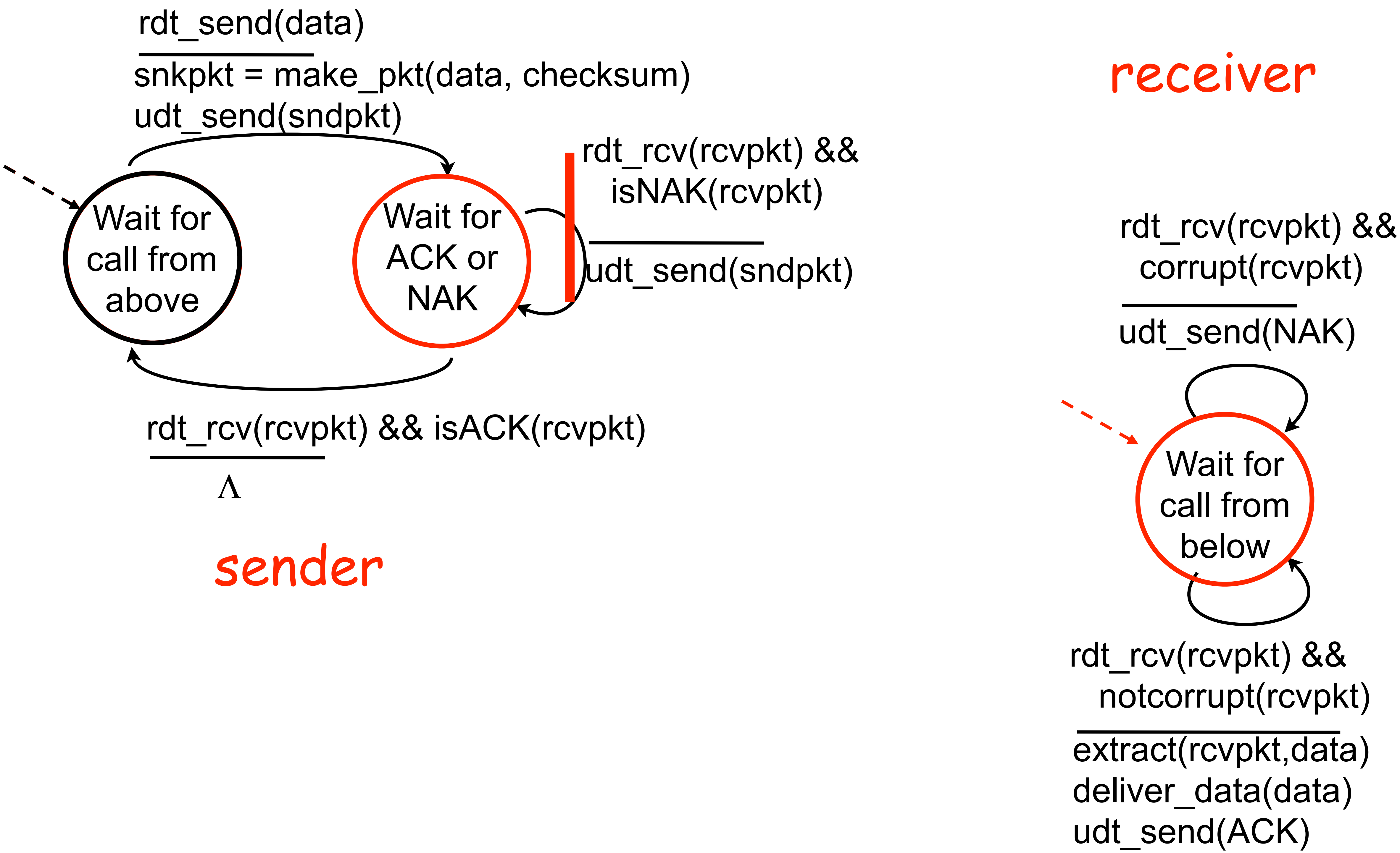
rdt2.0: error scenario



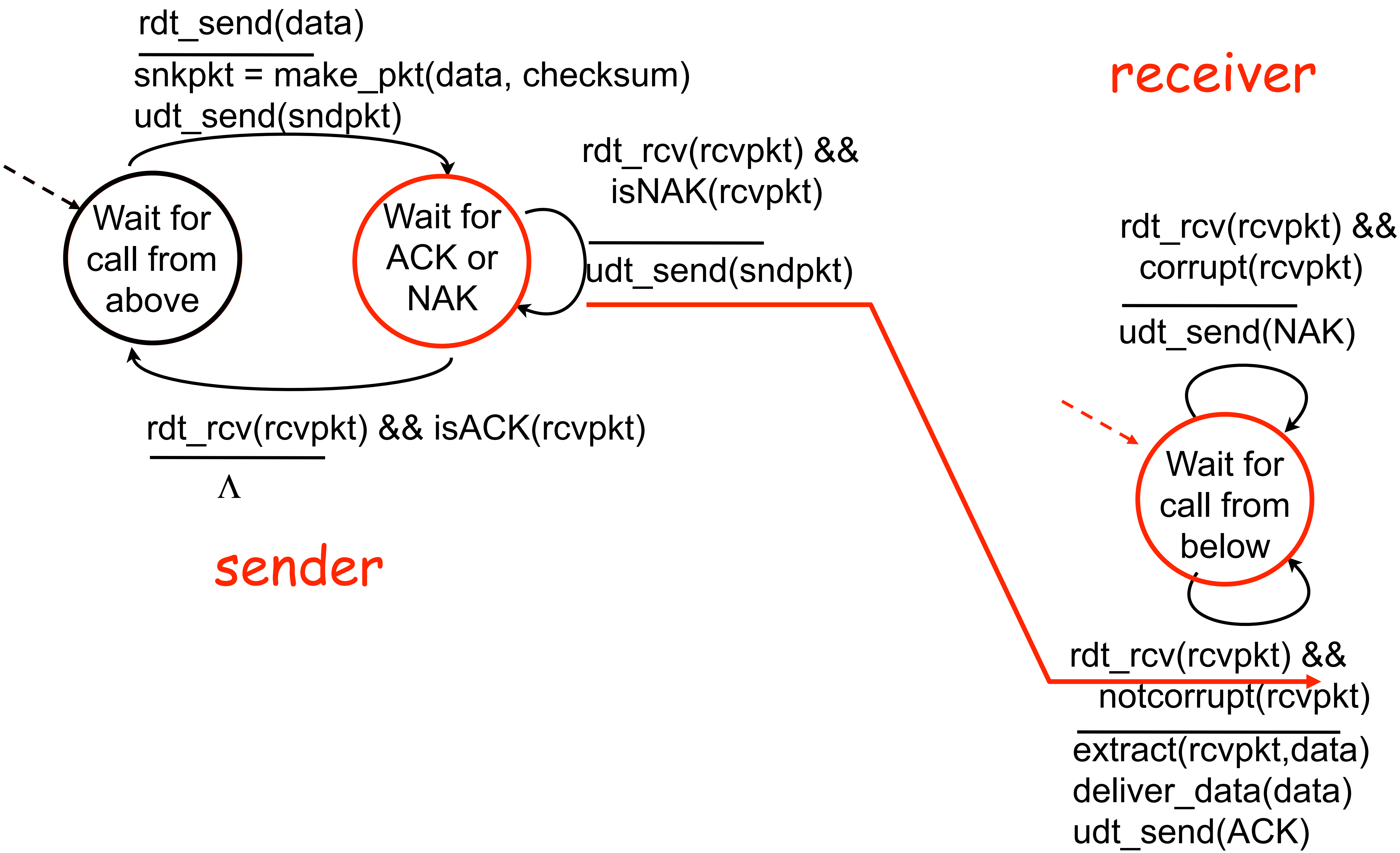
rdt2.0: error scenario



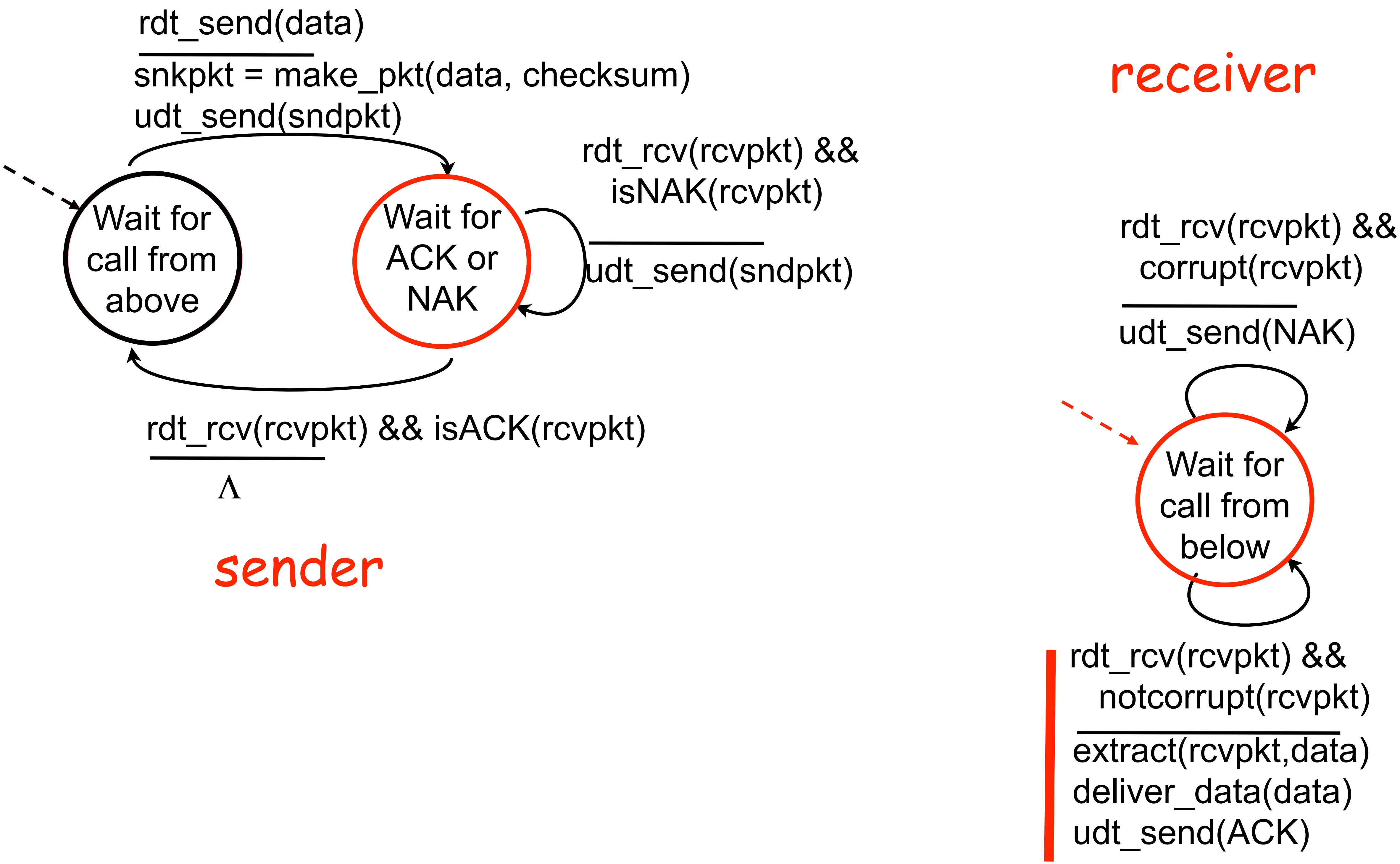
rdt2.0: error scenario



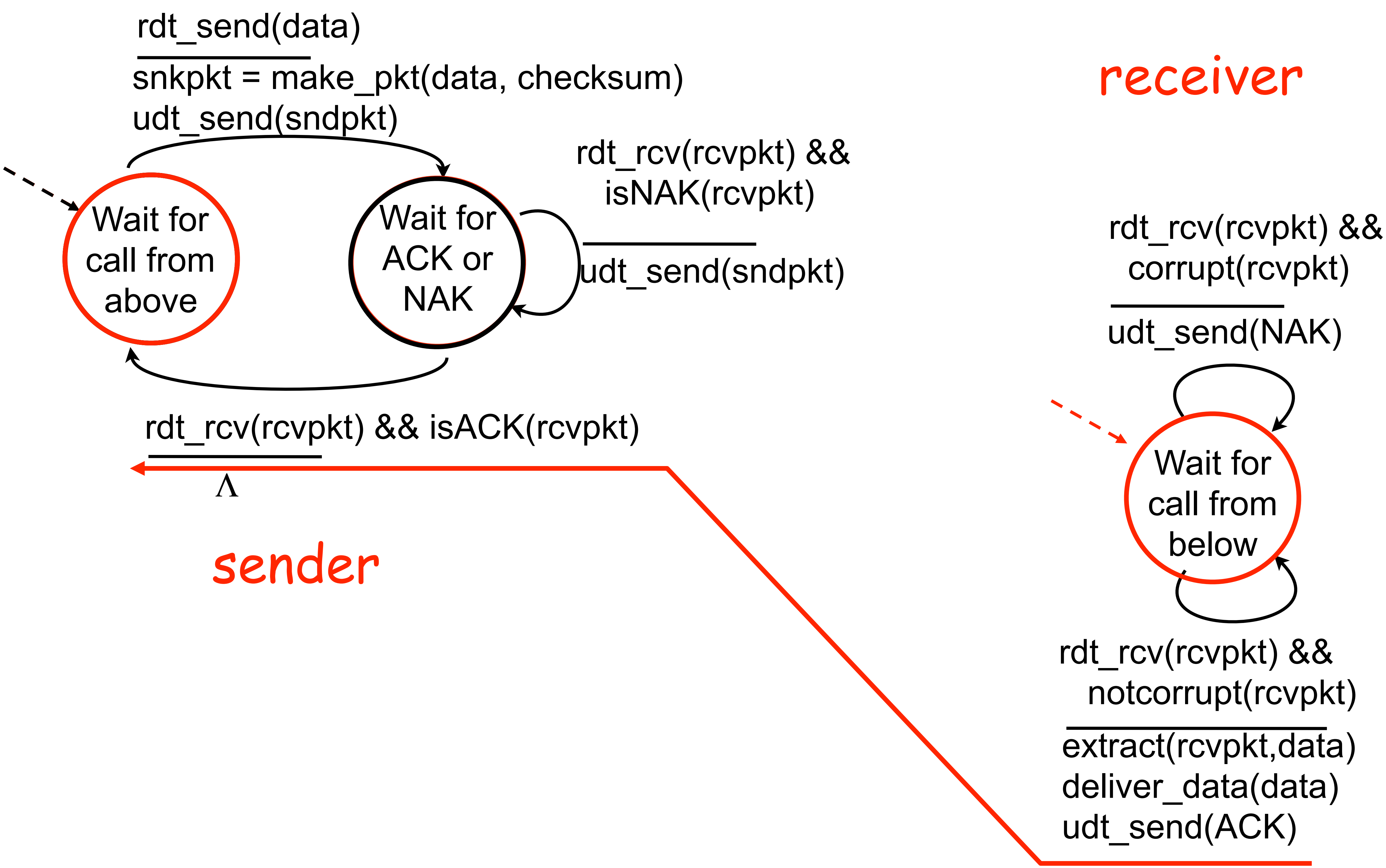
rdt2.0: error scenario



rdt2.0: error scenario



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!

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