

CSCI 491-01

Topics: Internet Programming

Fall 2008

Transport Layer

Derek Leonard
Hendrix College

October 1, 2008

Original slides copyright © 1996-2007 J.F Kurose and K.W. Ross

Chapter 3: Roadmap

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

UDP: User Datagram Protocol [RFC 768]

- “No frills,” “bare bones” Internet transport protocol
- “Best effort” service, UDP segments may be:
 - Lost, corrupted
 - Delivered out of order to the application
- **Connectionless:**
 - No handshaking between UDP sender and receiver
 - Each UDP segment handled independently of others

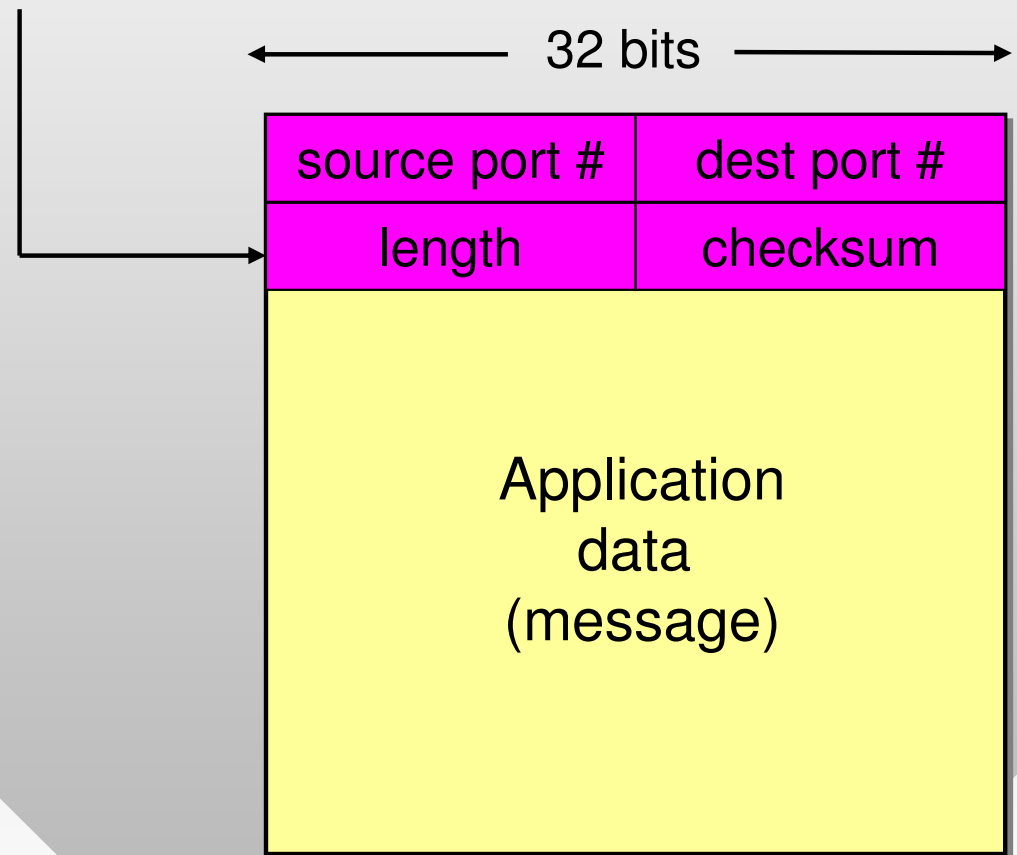
Why is there a UDP?

- 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: More

- Often used for streaming multimedia apps
 - Loss tolerant
 - Rate sensitive
- Other UDP uses
 - DNS
 - SNMP
 - NFS
- Reliable transfer over UDP: add reliability at application layer
 - Application-specific error recovery!

Length (in bytes) of
UDP segment,
including header



UDP segment format

UDP Checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment (packet)

Sender:

- Treat segment contents as a sequence of 16-bit integers
- Checksum: add all integers, then XOR with 0xffff (1's complement)
- Sender puts checksum value into UDP checksum field

Receiver:

- Compute checksum of received segment with the checksum field in it
- Check if result = 0xffff
 - NO - error detected
 - YES - no error detected
- Idea: $(x \text{ XOR } 0xffff) + x = 0xffff$
- *Are errors possible nonetheless?*

UDP Checksum Example

- Note
 - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
<hr/>																	
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
<hr/>																	
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0	
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1	

UDP Checksum (Cont)

- How many corrupted bits can a UDP checksum detect?
- Can we construct an example of a single-bit corruption that is not detected by the checksum?
- Example of a two-bit corruption?
- Suppose the data has only two 16-bit words:
 - (0, 5) which result in a checksum $5 \text{ XOR } 0\text{xffff}$
 - Suppose 0 is corrupted to become 1 and 5 is corrupted to become 4, then the checksum is the same
- What if the transmitted words are 0 and 12?
 - Can two-bit corruption produce the same checksum?
 - If yes, how many ways can (0,12) be corrupted so as to avoid detection?

Chapter 3: Roadmap

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

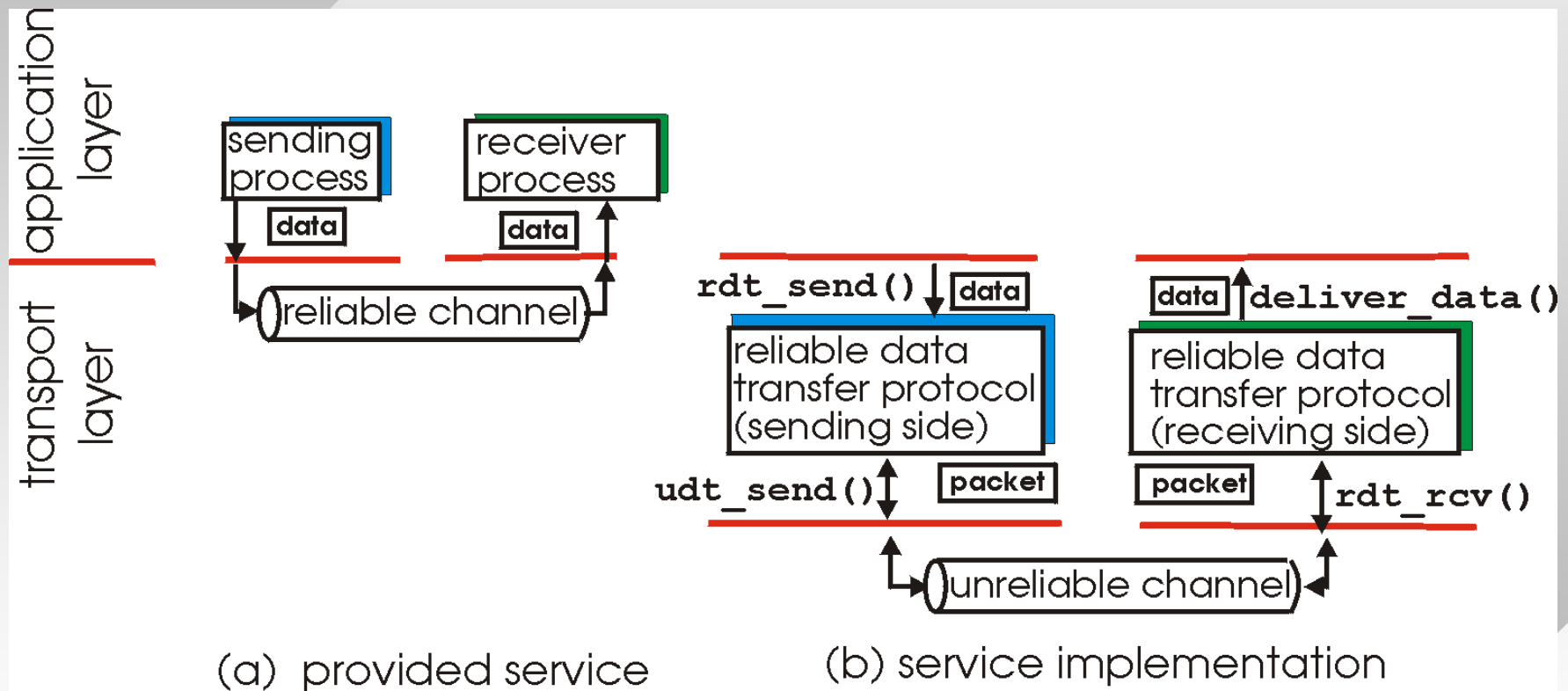
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Principles of Reliable Data Transfer

- Important in application, transport, link layers

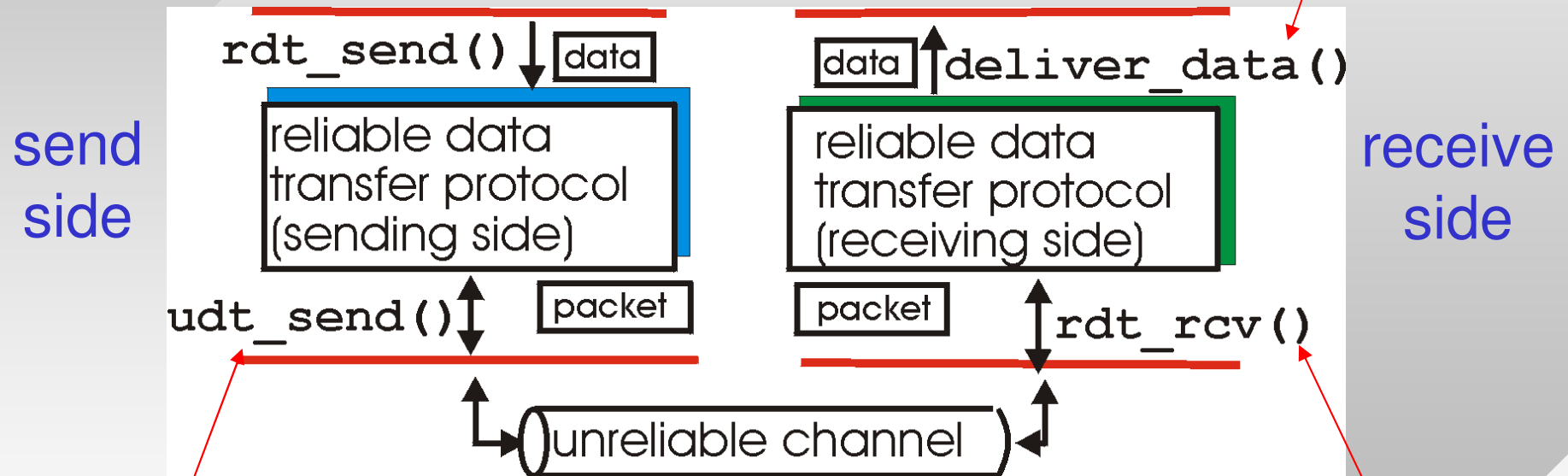


- Characteristics of unreliable channel will determine complexity of **reliable data transfer** (rdt) protocol

Reliable Data Transfer: Getting Started

rdt_send(): called from above (e.g., by the application): passes data to be delivered to the receiver

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



udt_send(): called by rdt to transfer packets over an unreliable channel to receiver

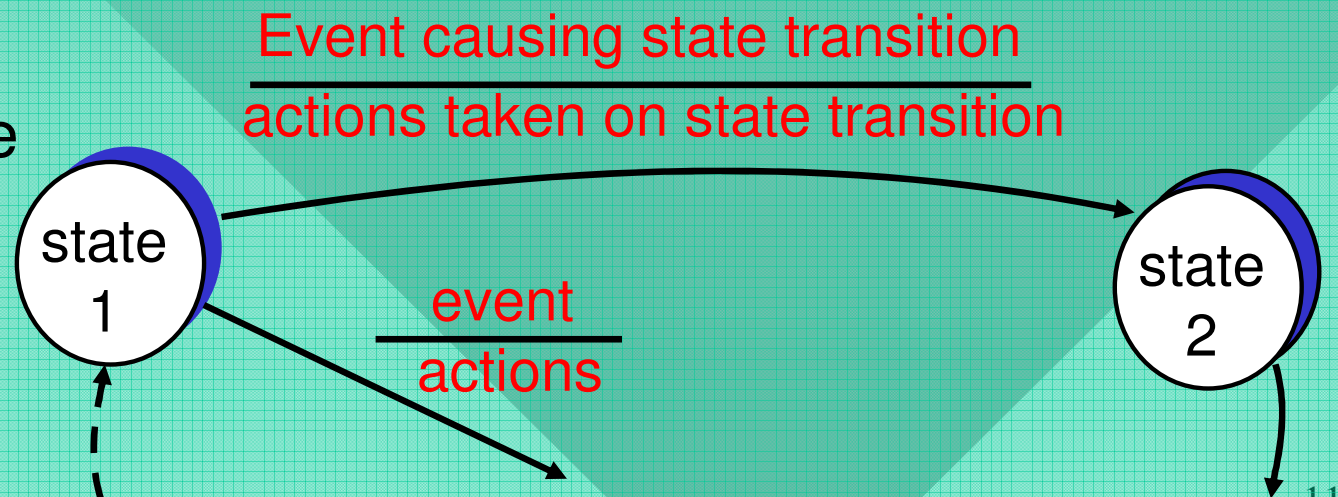
rdt_rcv(): called when packet arrives on receiver-side of channel

Reliable Data Transfer: Getting Started

We will:

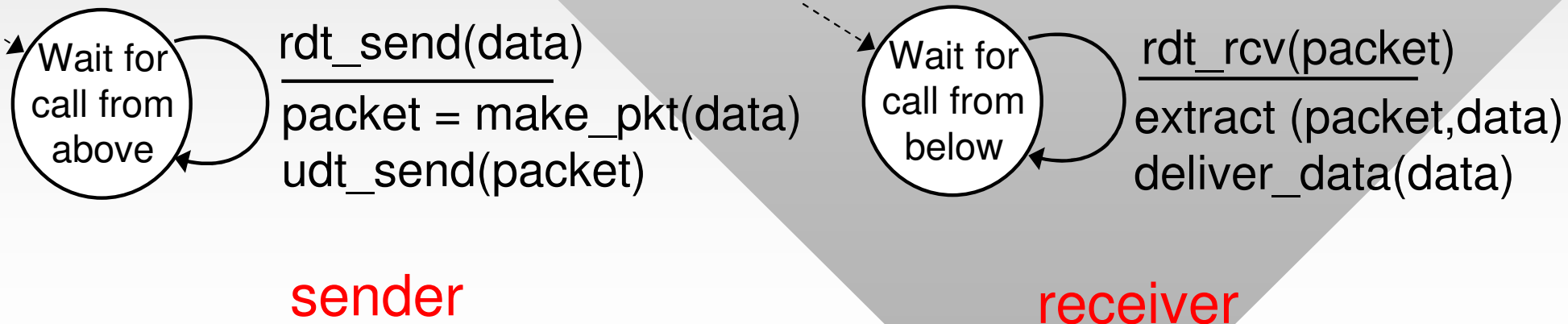
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
 - But control info will flow on both directions!
- Use **finite state machines** (FSM) to specify both sender and receiver

State: when in this “state,” next state uniquely determined by next event



Rdt1.0: Transfer Over a Reliable Channel

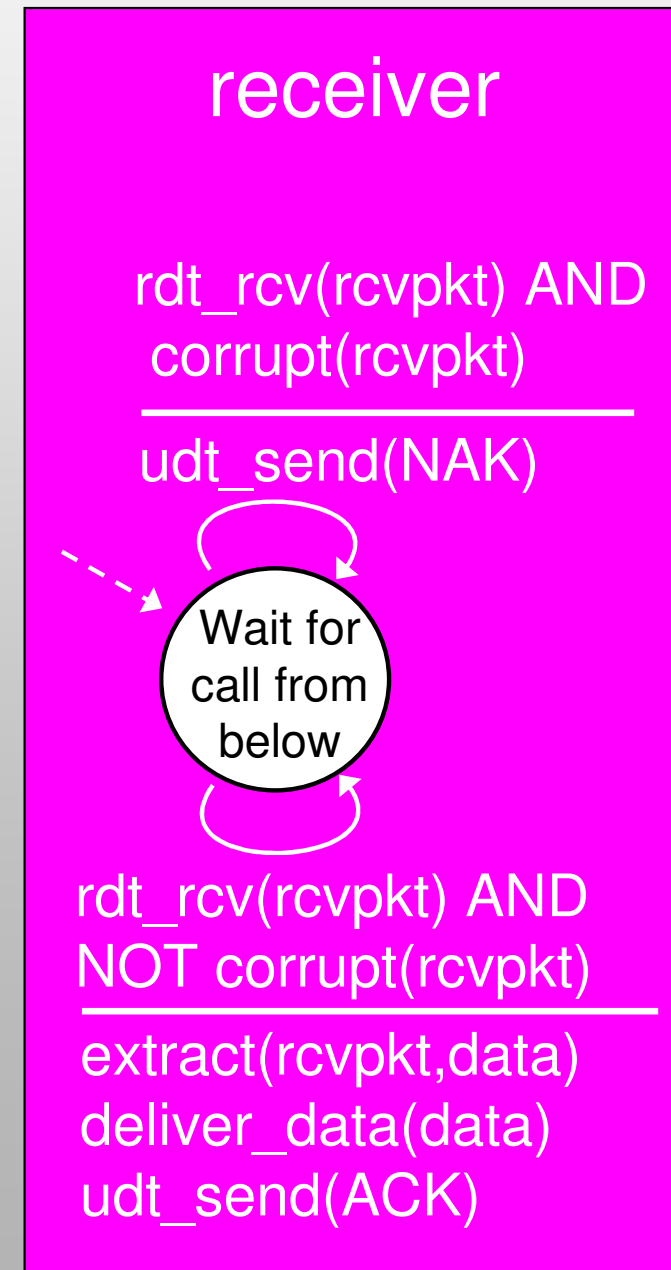
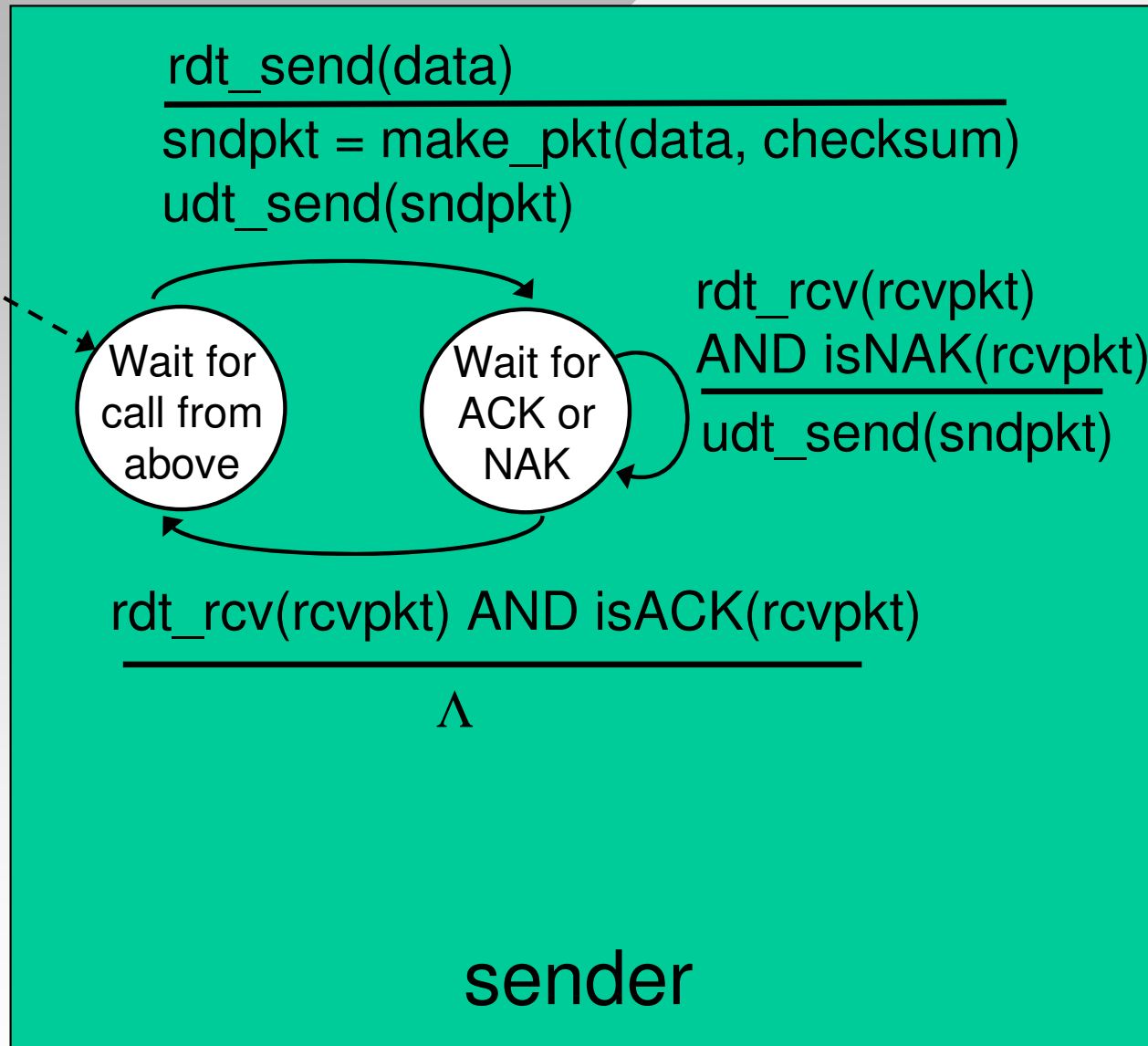
- Underlying channel perfectly reliable
 - No bit errors
 - No loss of packets
- Separate FSMs for sender and receiver:
 - Sender sends data into underlying channel
 - Receiver reads data from underlying channel



Rdt2.0: Channel With Bit Errors

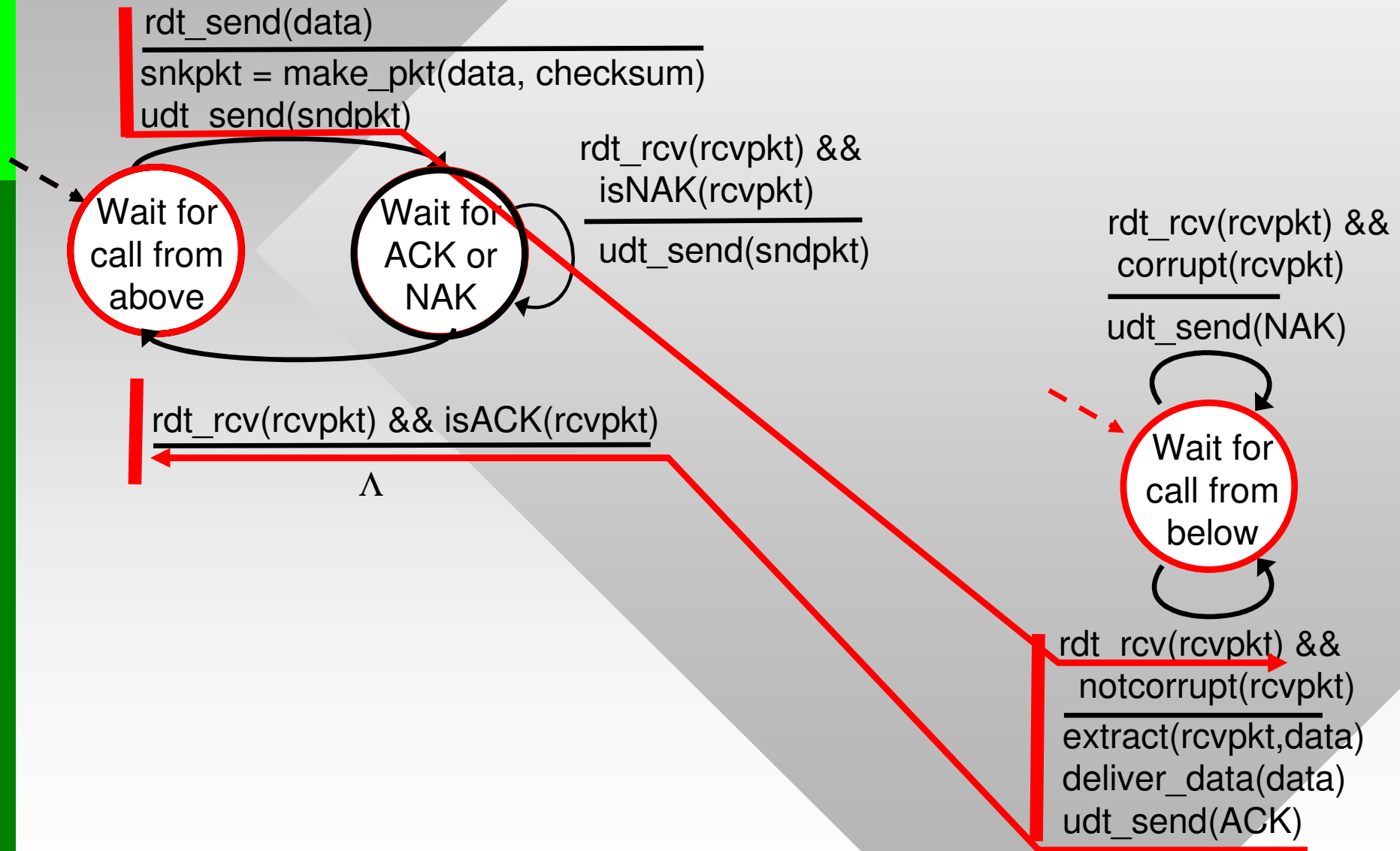
- Underlying channel may flip bits in packet (no loss)
 - Checksum to detect bit errors
- Question: how to recover from errors?
- One possible approach to use two feedback msgs:
 - *Acknowledgements (ACKs)*: receiver explicitly tells sender that packet was received OK
 - *Negative acknowledgements (NAKs)*: receiver explicitly tells sender that packet had errors
 - Sender retransmits packet on receipt of NAK
- New mechanisms in rdt 2.0 (beyond rdt 1.0):
 - Error detection
 - Receiver feedback: control msgs (ACK,NAK) from receiver to sender

Rdt2.0: FSM Specification

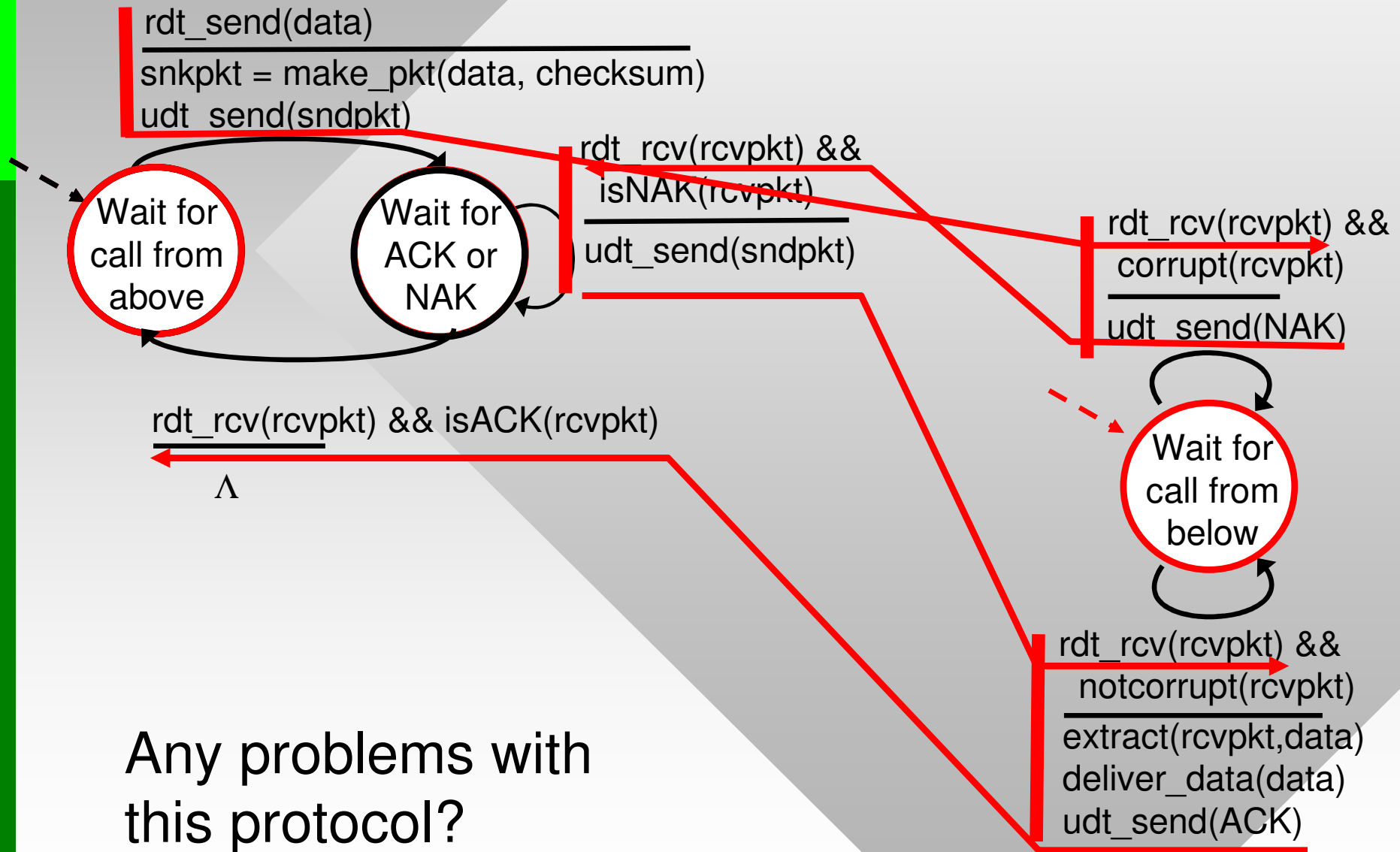


Λ = empty action, i.e., do nothing

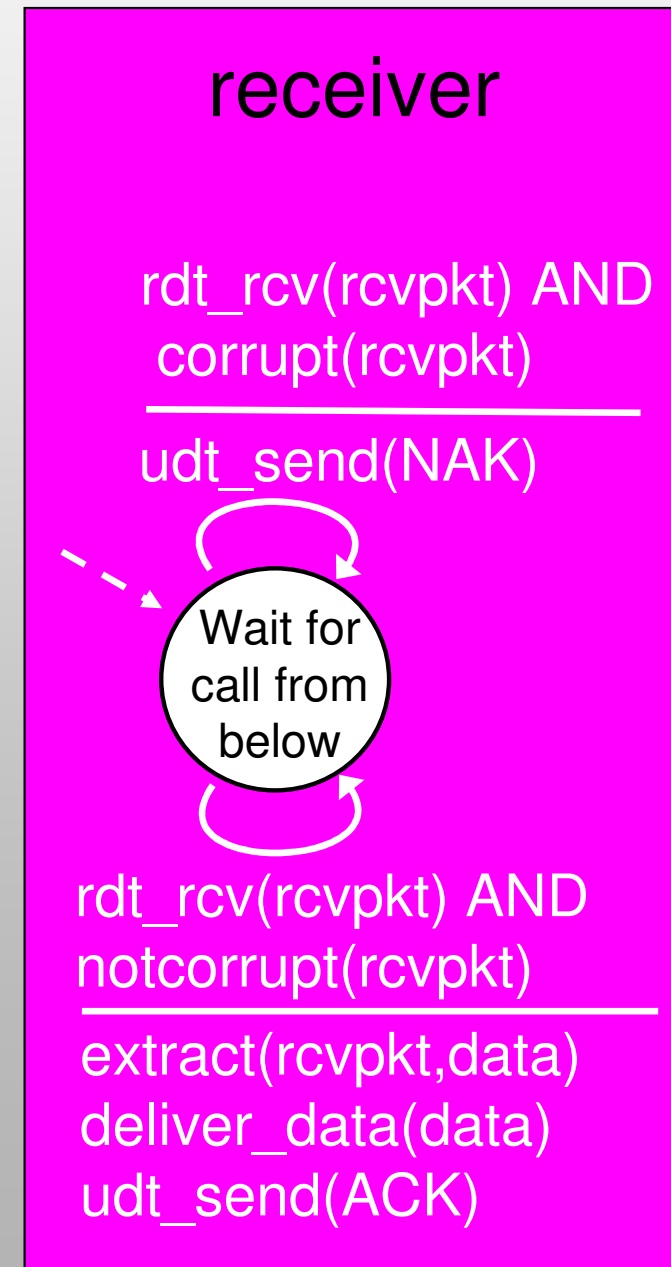
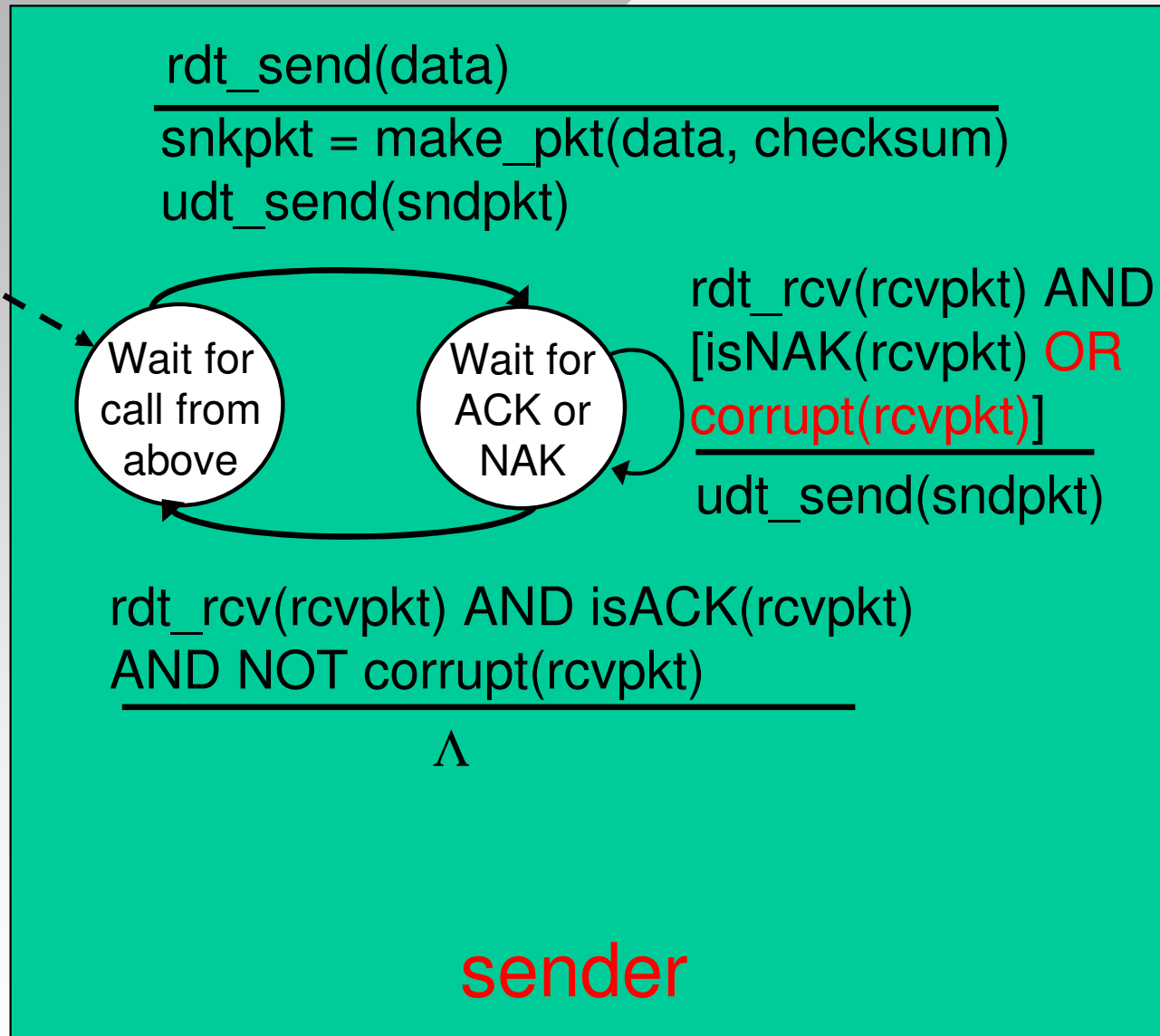
Rdt2.0: Operation With No Errors



Rdt2.0: Error Scenario



Rdt2.0a: Handles Corrupted Feedback



Any problems?

Rdt2.0 and Rdt2.0a Have Fatal Flaws

- Rdt 2.0 does not work when ACK/NAK is corrupted
 - Sender doesn't know what happened at receiver!
- Rdt 2.0a delivers duplicate packets to application

Proper algorithm:

- Sender adds *sequence number* to each pkt
- Sender retransmits current pkt if ACK/NAK is garbled
- Receiver discards (doesn't deliver up) duplicate pkt

Stop-and-Wait protocol: sender sends one packet, then waits for receiver's response

Rdt2.1: Sender, Handles Garbled ACK/NAKs

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)

udt_send(sndpkt)

rdt_rcv(rcvpkt) AND
[corrupt(rcvpkt) OR
isNAK(rcvpkt)]

udt_send(sndpkt)

rdt_rcv(rcvpkt) AND
NOT corrupt(rcvpkt)
AND isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) AND
NOT corrupt(rcvpkt) AND
isACK(rcvpkt)

Λ

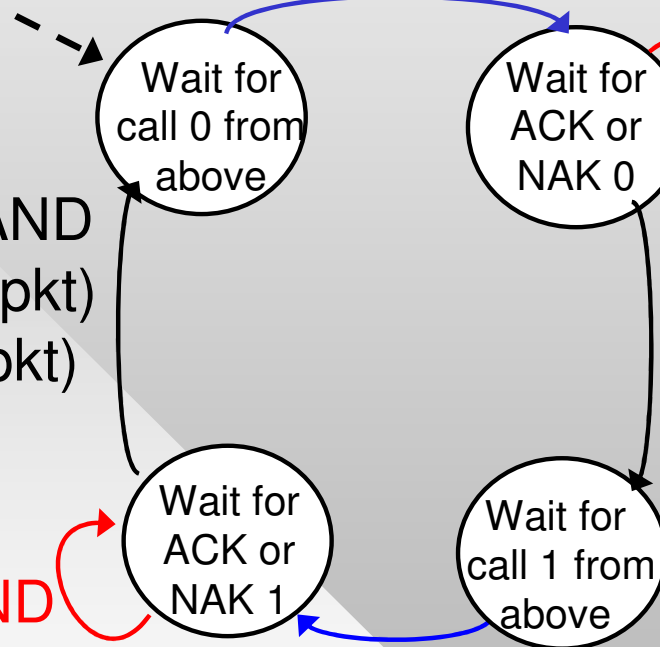
rdt_rcv(rcvpkt) AND
[corrupt(rcvpkt) OR
isNAK(rcvpkt)]

udt_send(sndpkt)

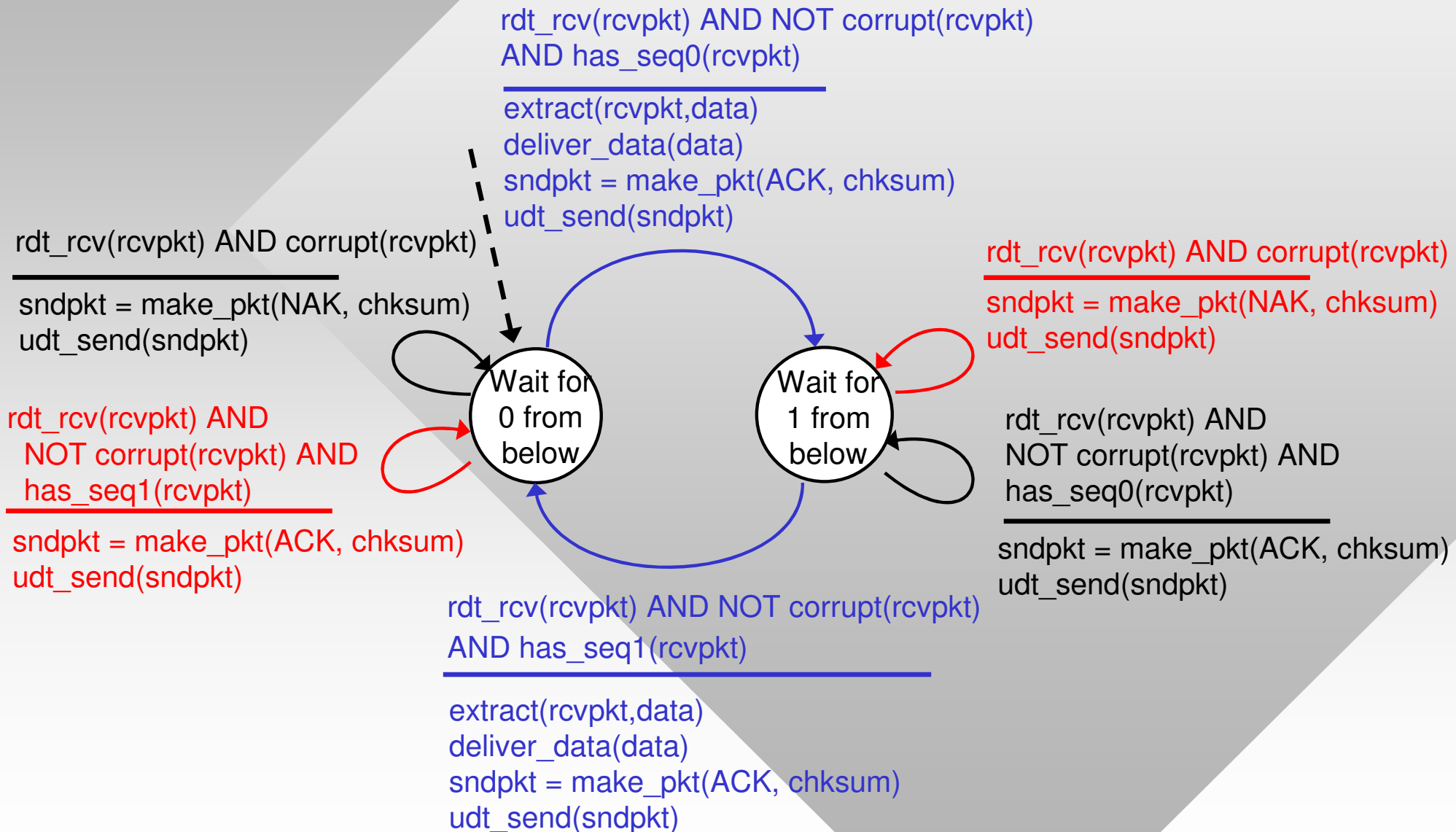
rdt_send(data)

sndpkt = make_pkt(1, data, checksum)

udt_send(sndpkt)



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
 - Protocol must remember whether current pkt has 0 or 1 sequence number

Receiver:

- Must check if received packet is duplicate
 - State indicates whether 0 or 1 is the expected packet seq #
- Note: receiver *cannot* know if its last ACK/NAK was received OK at sender