Transport layer services, multiplexing/ demultiplexing, reliable data transfer

CE 352, Computer Networks

Salem Al-Agtash

Lecture 8

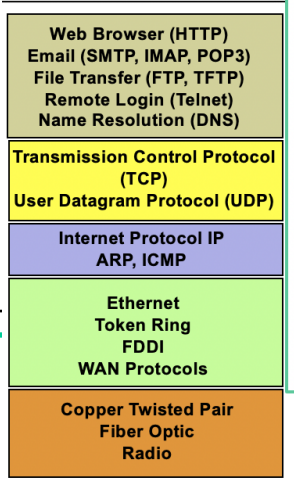
Recap

- Application Layer
- WWW
- Socket programming
- DNS
- Email
- P2P: File transfer, Video streaming, and CDN

Today:

- Transport Layer (multiplexing, demultiplexing, reliable data transfer)

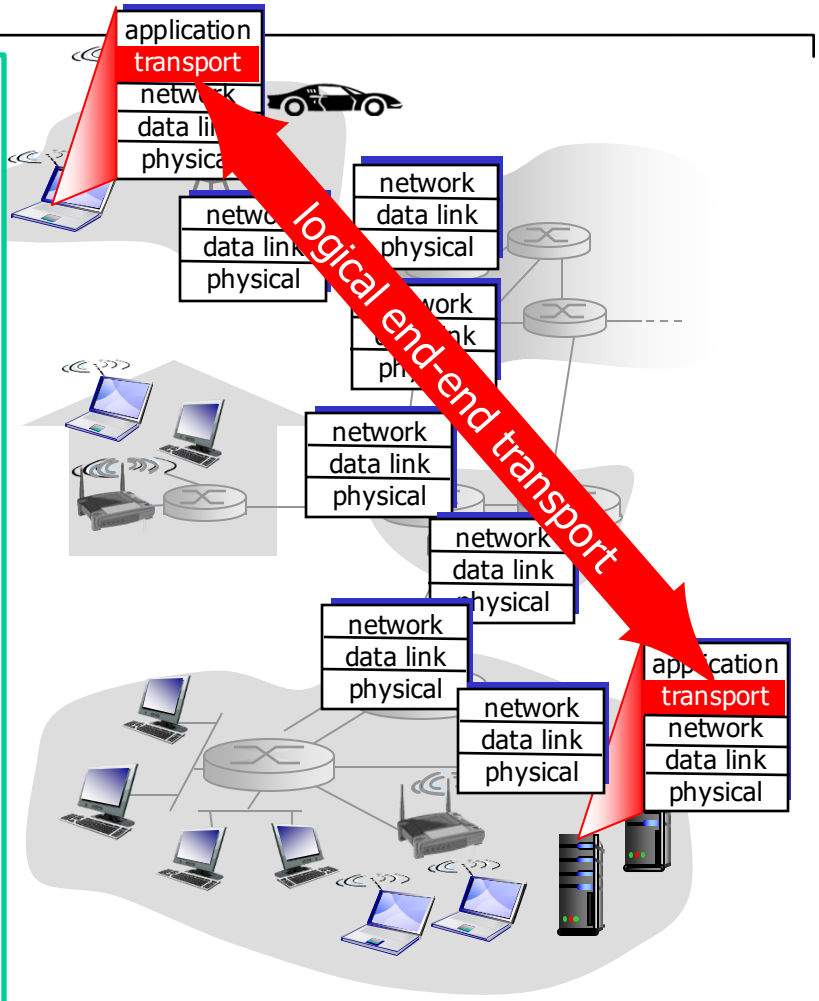
Course outcome - mapping



1. Describe the layered network architecture, the services provided by each layer, and the respective protocols (Application,..) that deliver these services, including wireless protocols.
2. Compare and contrast mechanisms used to improve network performance.
3. Describe security threats and mechanisms and protocols used to protect network communications.
4. Explain how network mechanisms and protocols (Application,..) have evolved to accommodate the growth of the internet and the use of new applications and technology.
5. Write network software using sockets.

Transport Layer

- Provide *logical communication* between app processes running on different hosts
- Transport protocols run in end systems
  - TCP or UDP
  - Sending host: breaks app messages into *segments*, passes to network layer
  - Receiving host: reassembles segments into messages, passes to app layer
- Port numbers (16 bits):
  - Destination IP selects the Server
  - Destination port number selects process
  - ICANN ranges:
    - 0-1023 are well known (/etc/services)
    - 1024 – 49151 registered
    - 49,152 – 65,535 dynamic or private



Transport Layer services

- Port numbers as addressing mechanisms at the transport layer
- Encapsulation/ decapsulation versus multiplexing/ demultiplexing
- Flow control and Error control at the transport layer
- Connection-oriented versus connectionless-oriented services and their FSM
- Generic transport layer protocols:
  - Simple protocol
  - Stop-and-Wait protocol
  - Go-Back-N protocol
  - Selective-Repeat protocol
- Process-to-process communication (transport layer) versus host-to-host communication (network layer)

Multiplexing/demultiplexing

Encapsulation

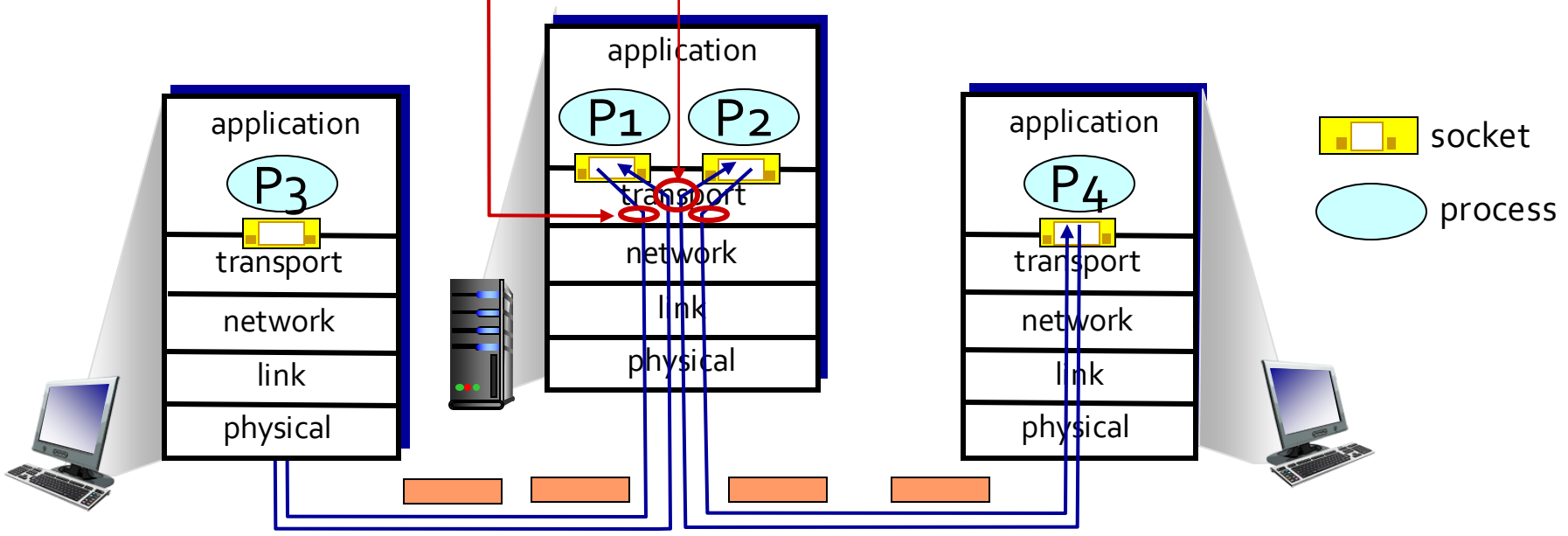
Decapsulation

multiplexing at sender:

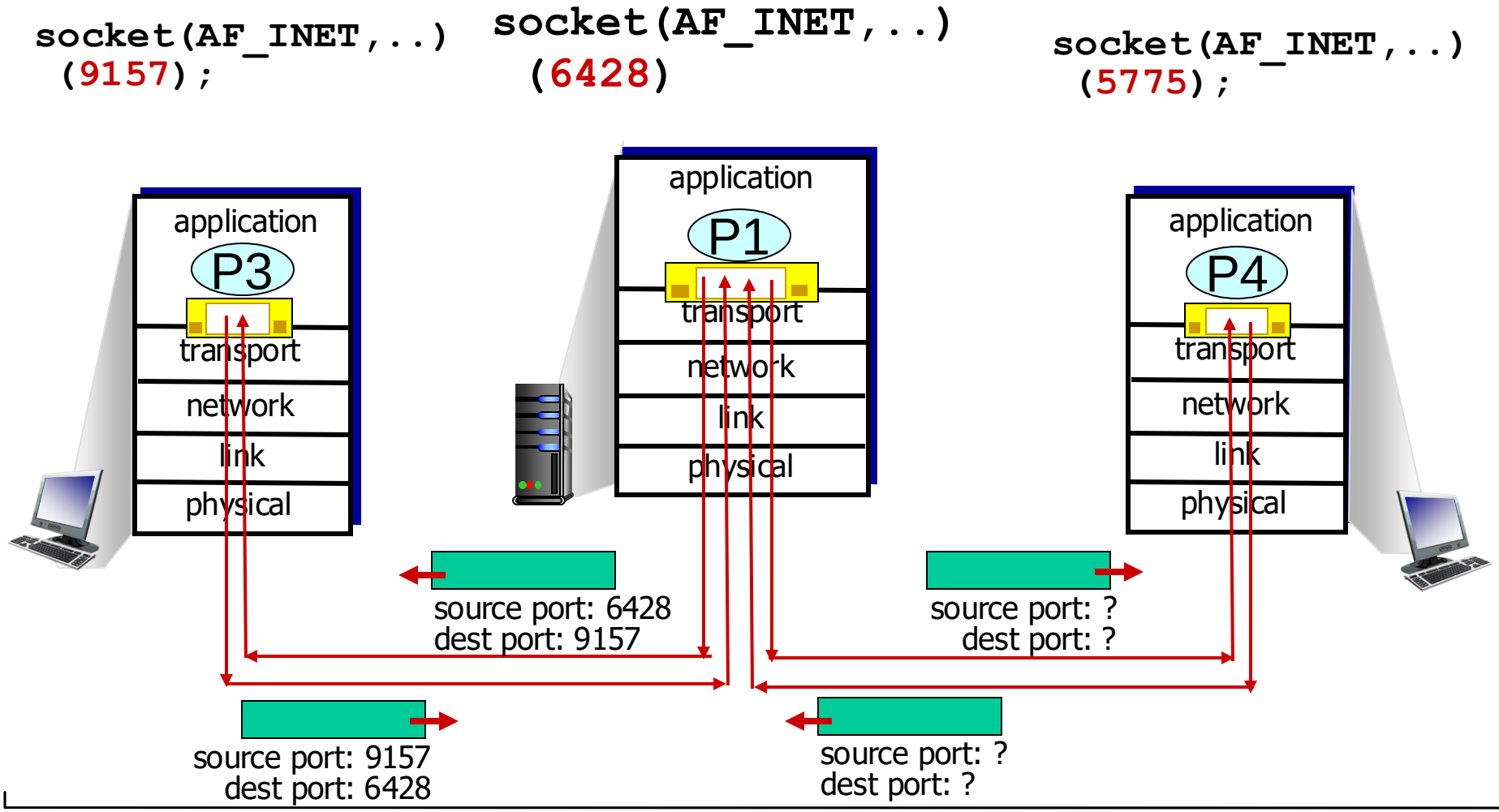
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

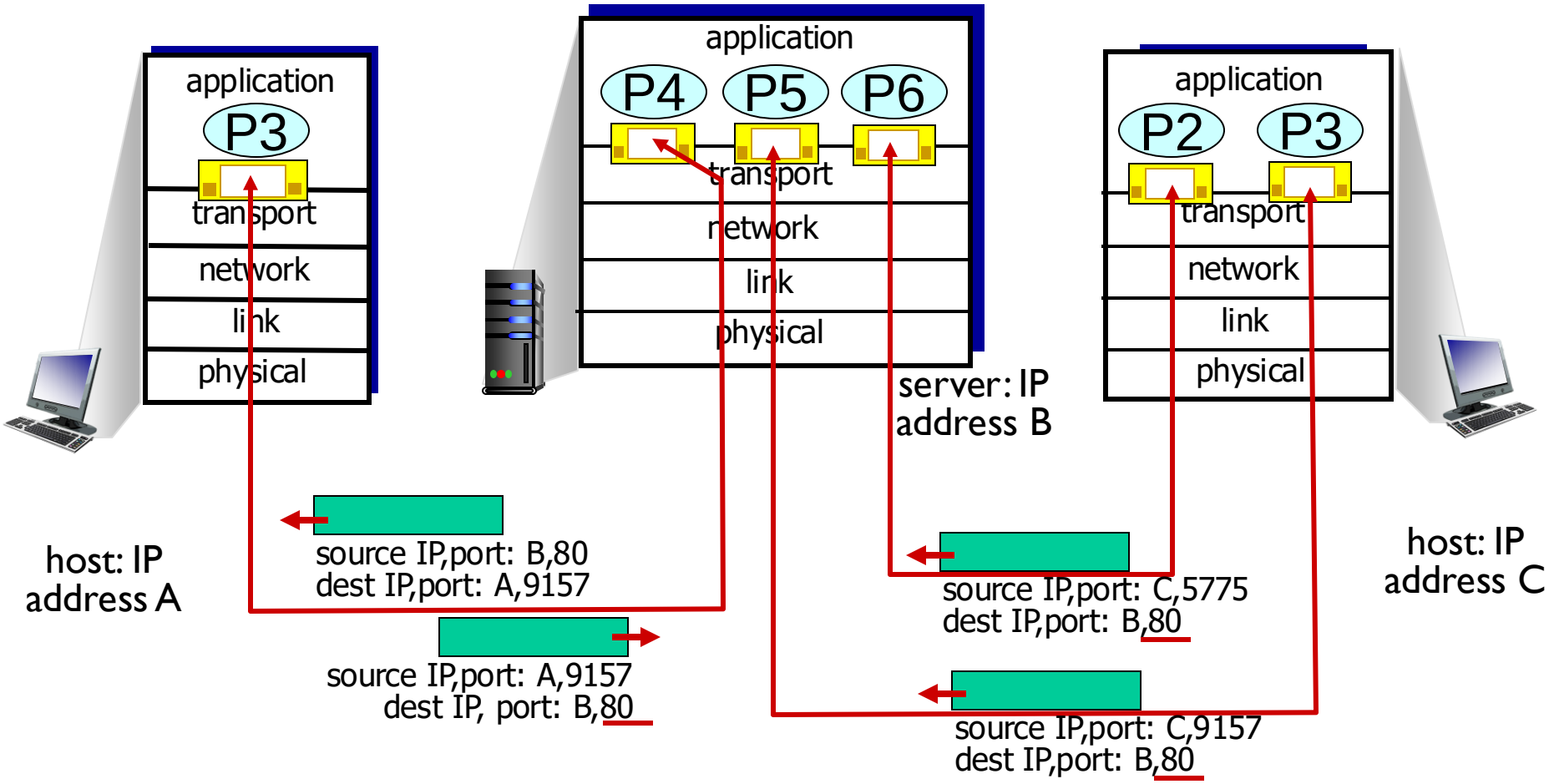
use header info to deliver received segments to correct socket



Connectionless demultiplexing



Connection-oriented demultiplexing

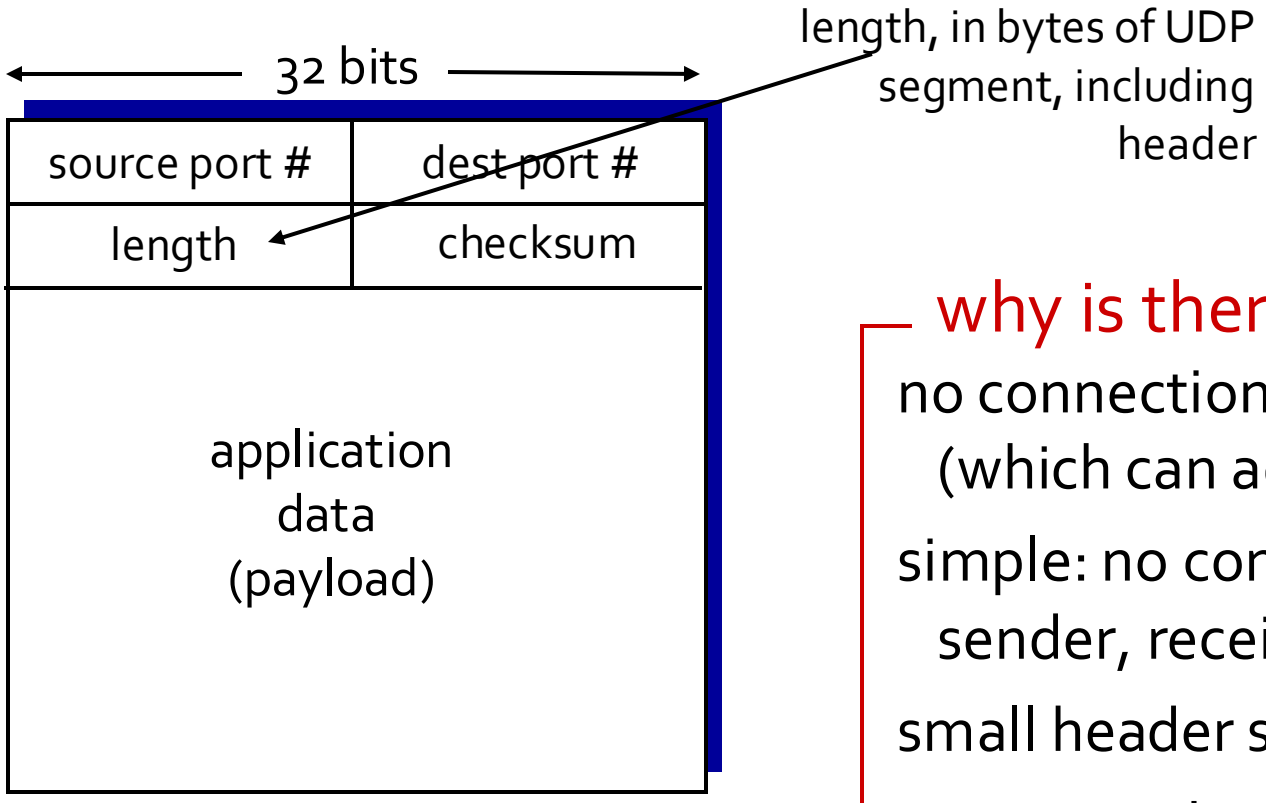


three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* connection descriptors

UDP: User Datagram Protocol [RFC 768]

- "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 use:
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP (Simple Network Management Protocol)
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

why is there a UDP?

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

UDP checksum

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

Sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one'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?* More later ….

Internet checksum

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

	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

wraparound 

wrapped sum

Checksum

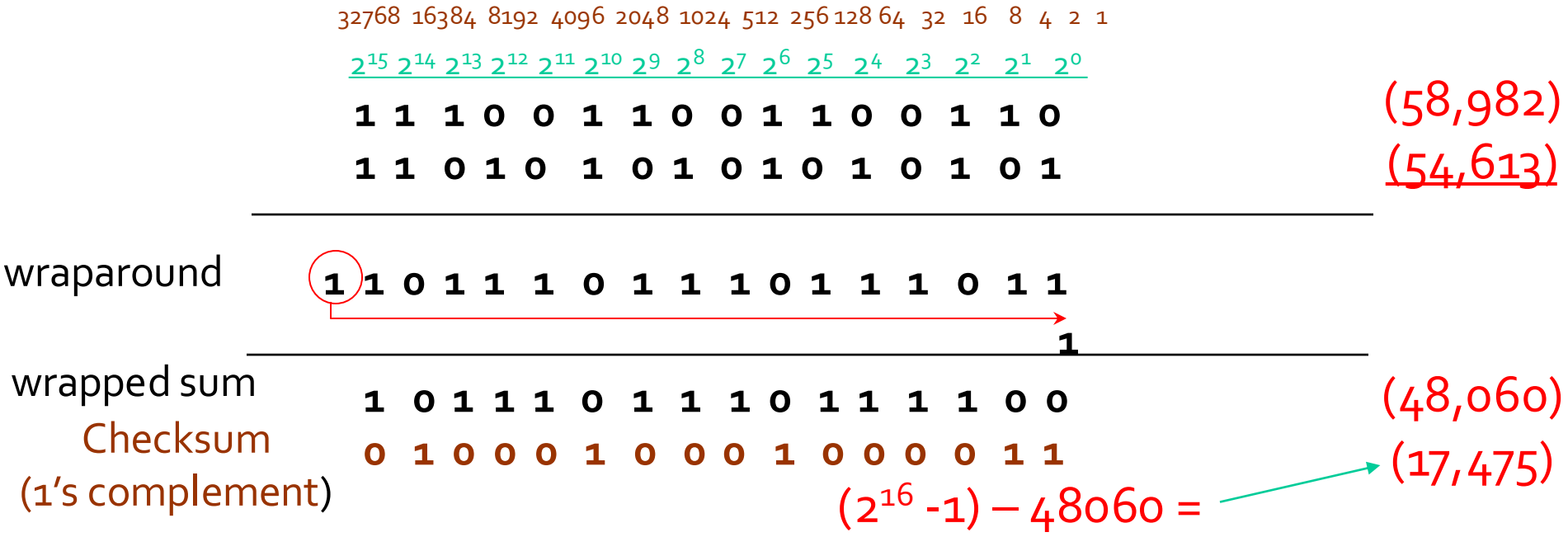
(1's complement)

Internet checksum

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

Internet checksum

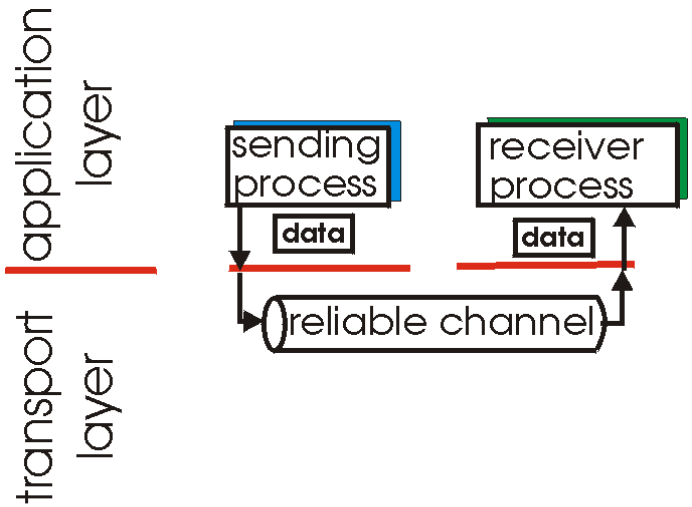
- Example: add two 16-bit integers
- *Note:* when adding numbers, a carryout from the most significant bit needs to be added to the result



58,982 + 54,613 + 17,475 = 131070 → 1111111111111110 → 1111111111111111 (wrapped sum) → 0

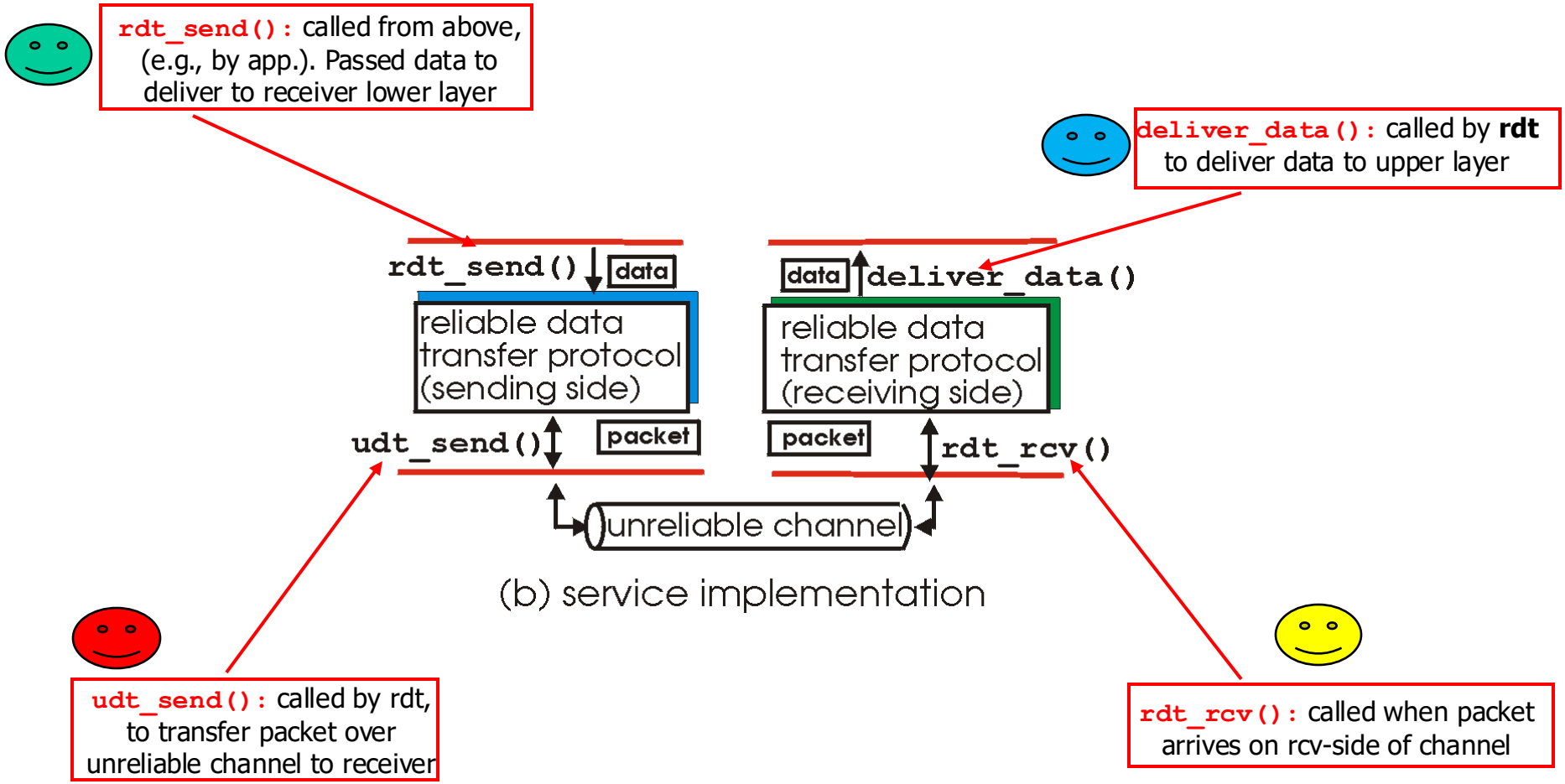
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Data transfer - reliable



(a) provided service

Data transfer - unreliable

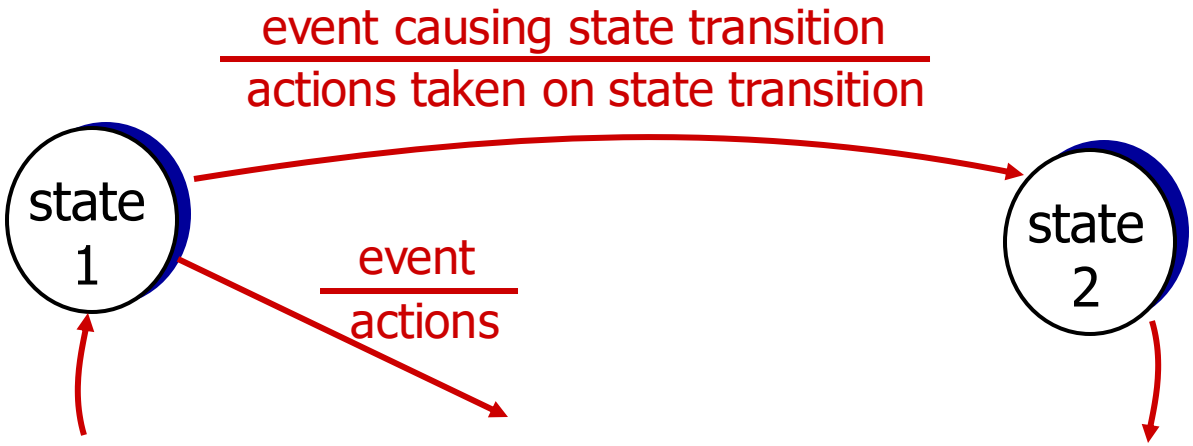


Reliable data transfer

- 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

Computation model that can be used to simulate sequential logic

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

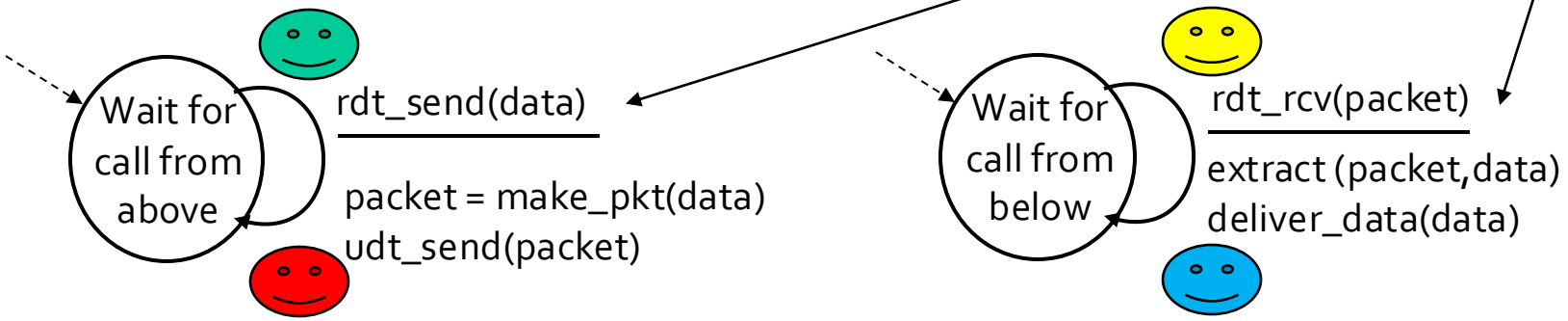


rdt1.0: reliable transfer over a reliable channel

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

Simple
provides neither
flow nor error control

event
actions



sender

receiver

rdt2.0: channel with bit errors (unreliable channel)

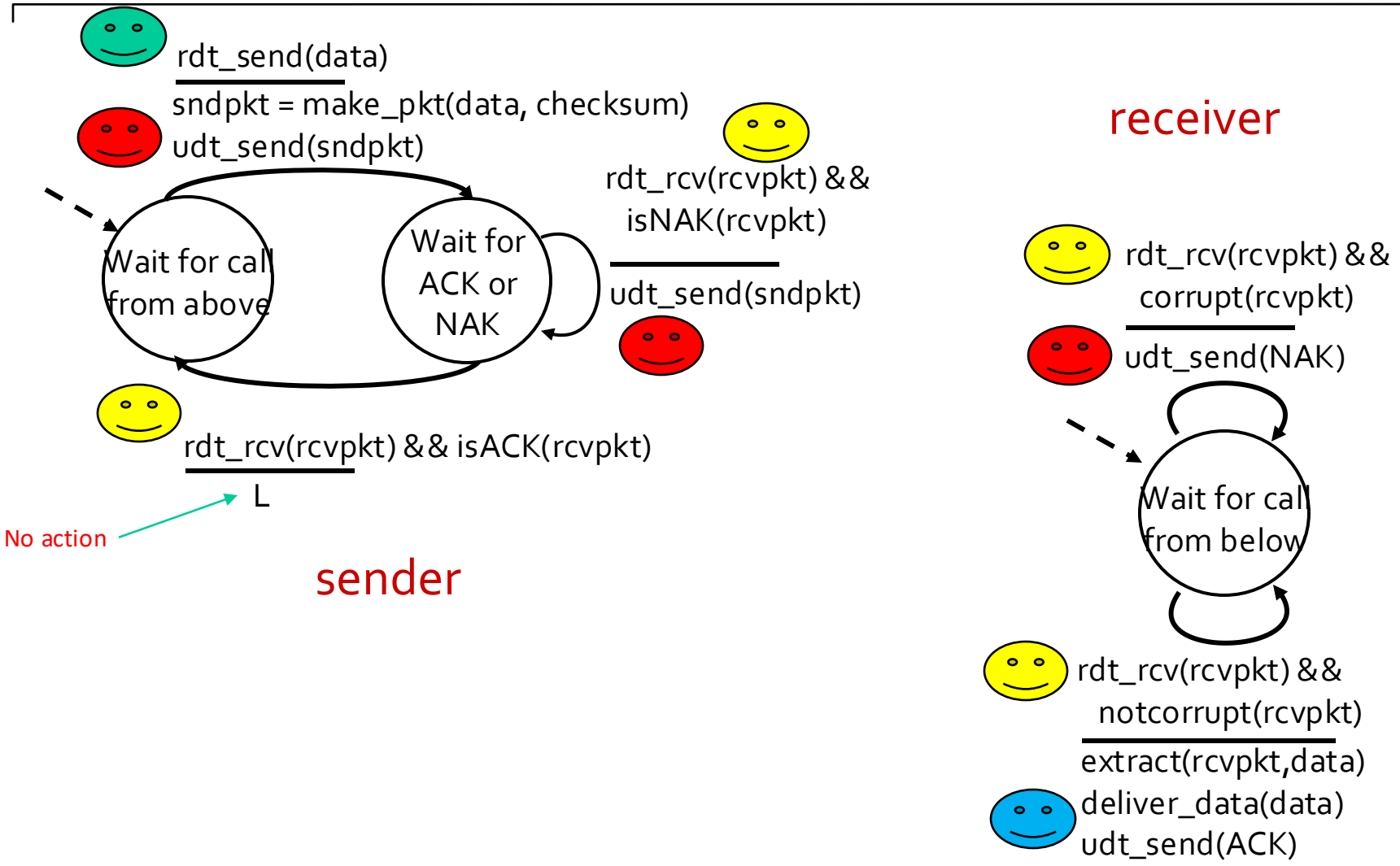
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
  - ARQ (Automatic Repeat reQuest) protocols:
    - Error detection
    - Receiver feedback
    - Retransmission
- How to recover from errors?

rdt2.0: channel with bit errors (unreliable channel)

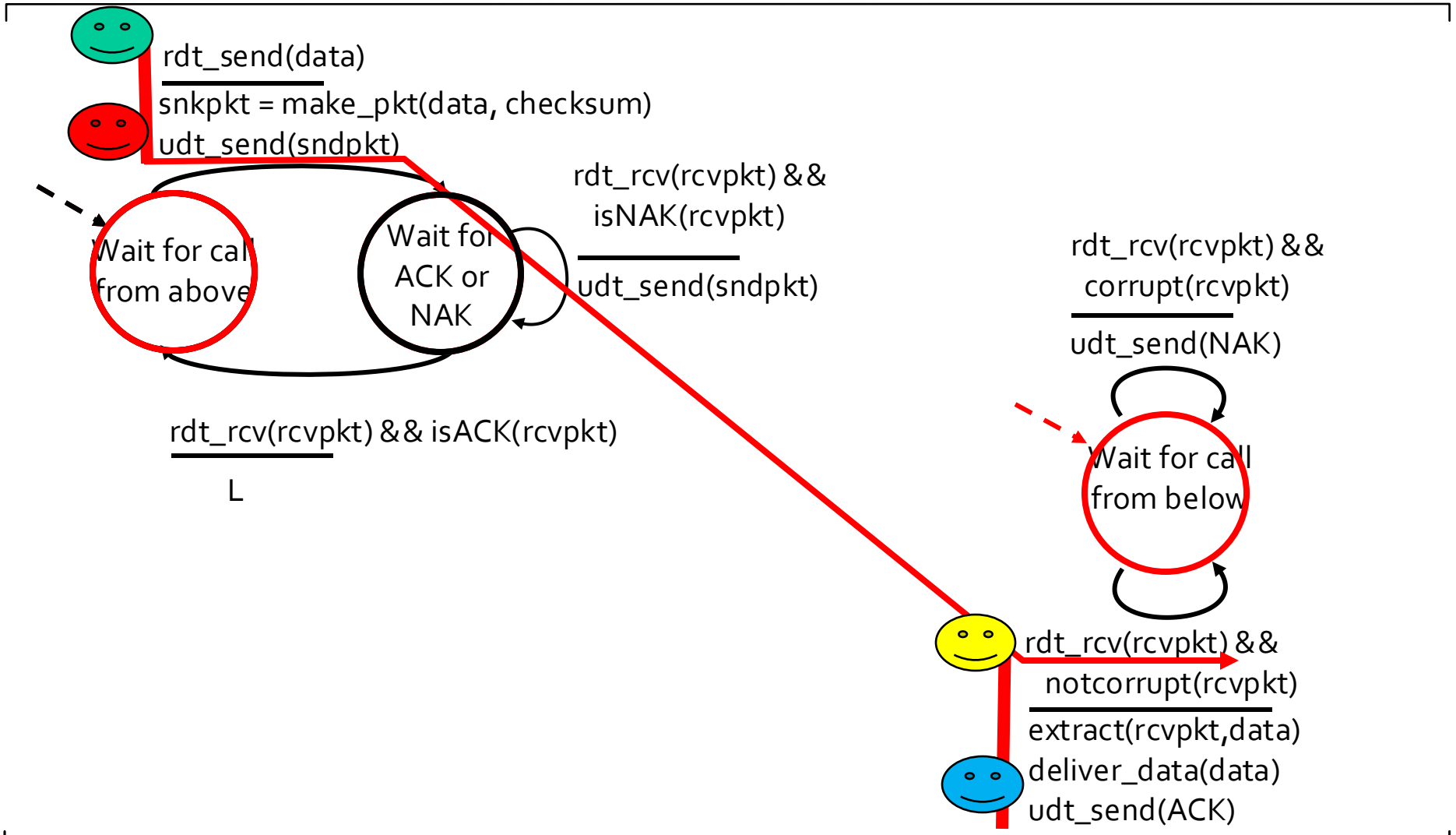
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
  - ARQ (Automatic Repeat reQuest) protocols:
    - Error detection
    - Receiver feedback
    - Retransmission
- How to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received 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

stop and wait

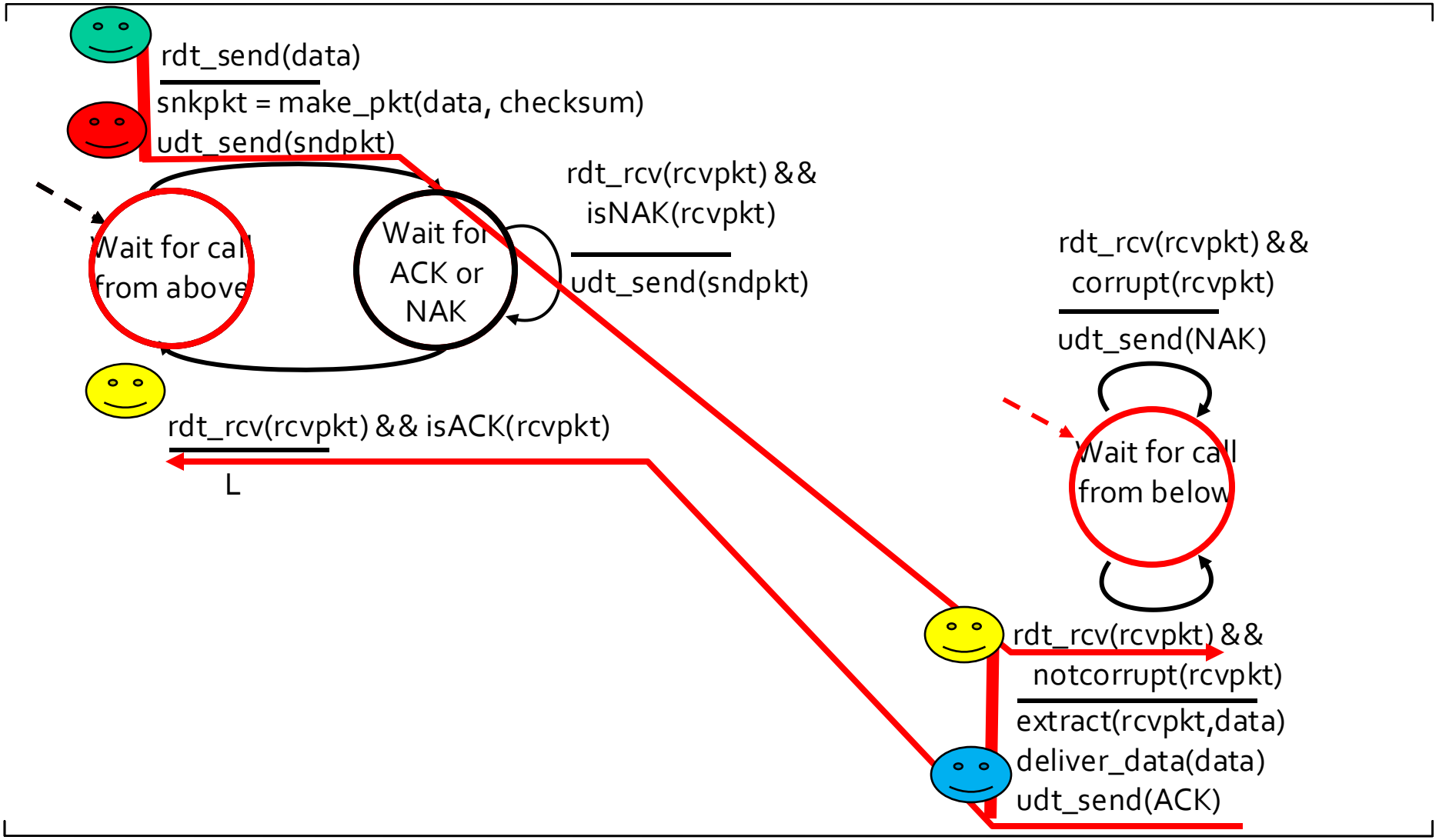
rdt2.0: FSM specification



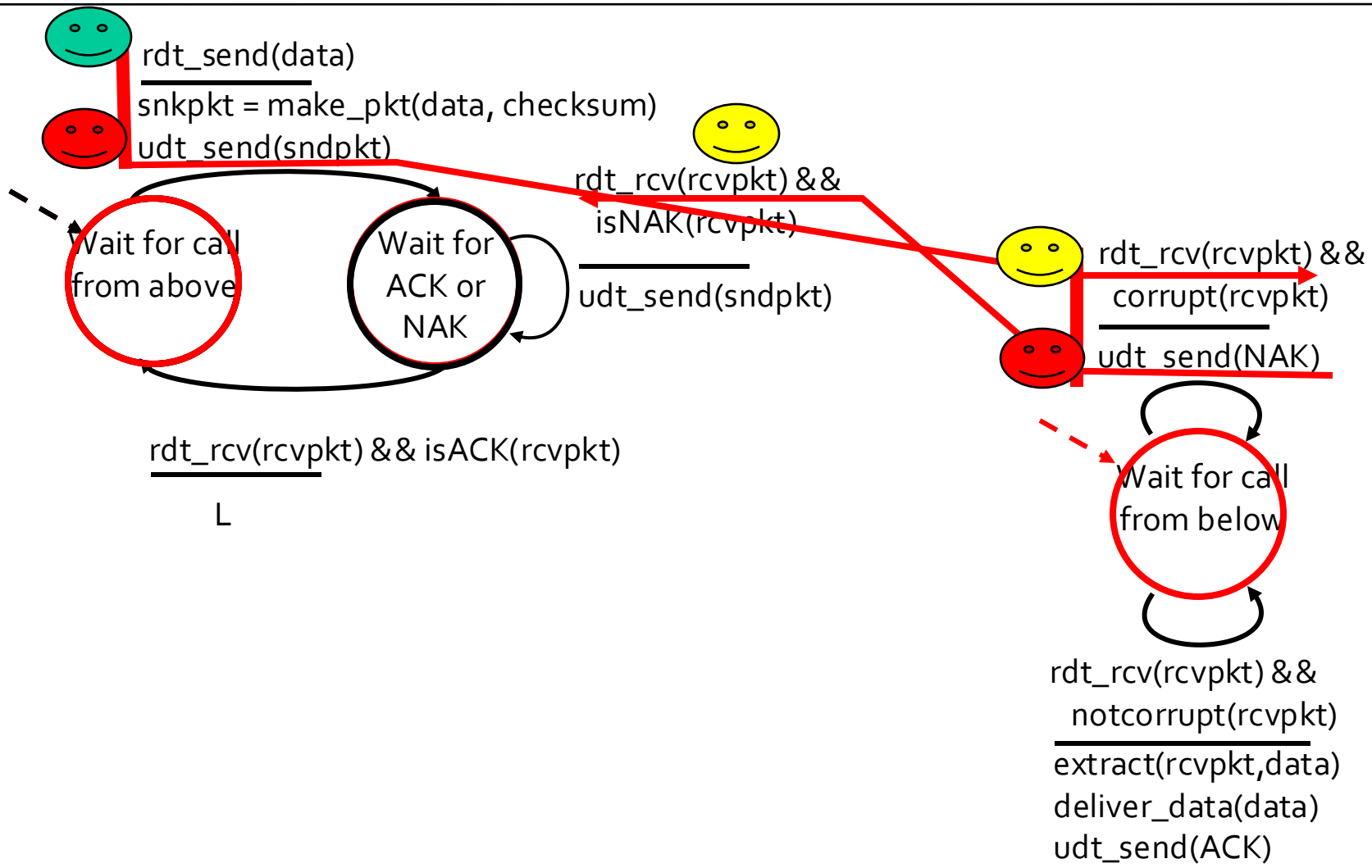
rdt2.0: operation with no errors



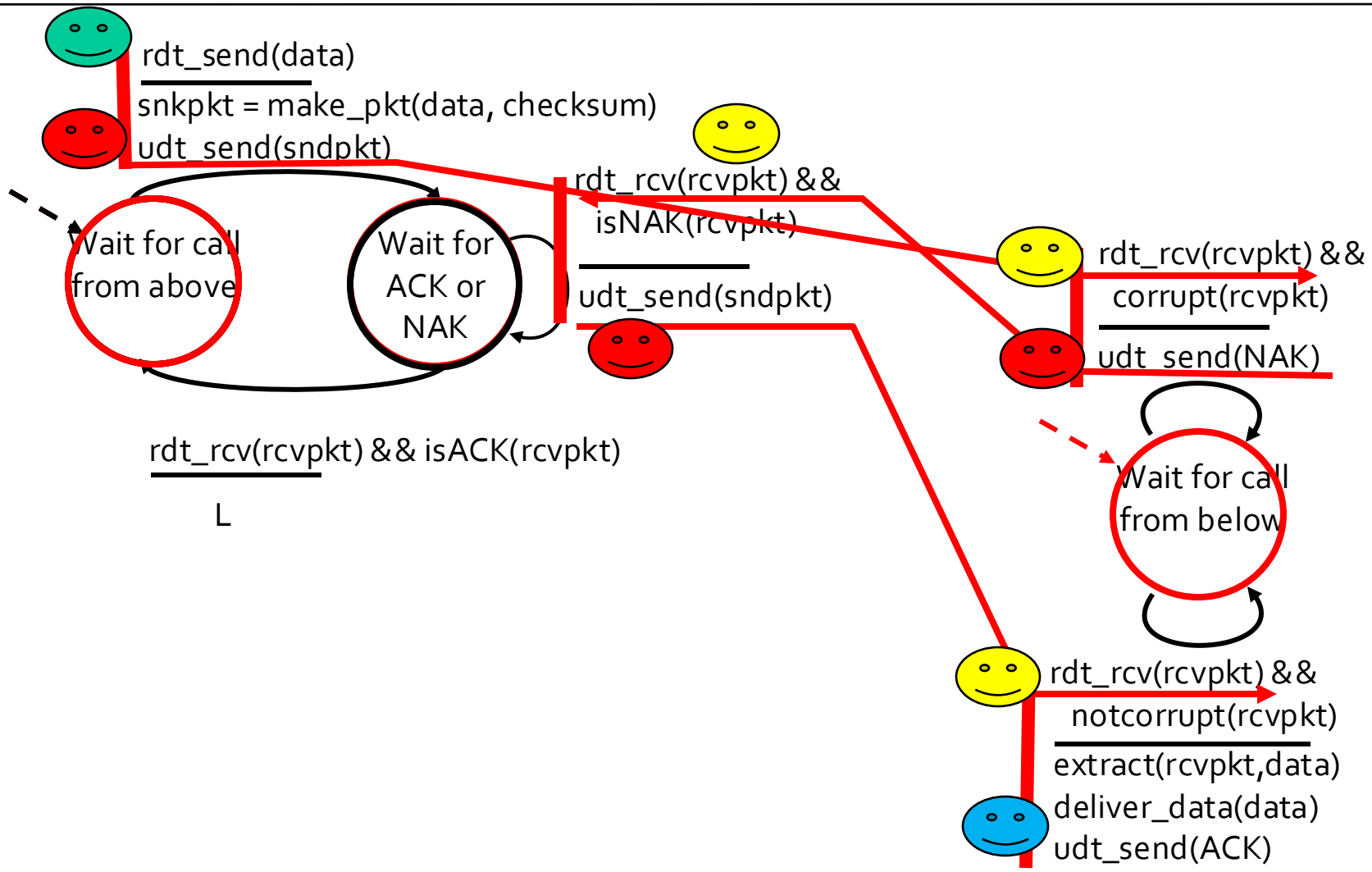
rdt2.0: operation with no errors



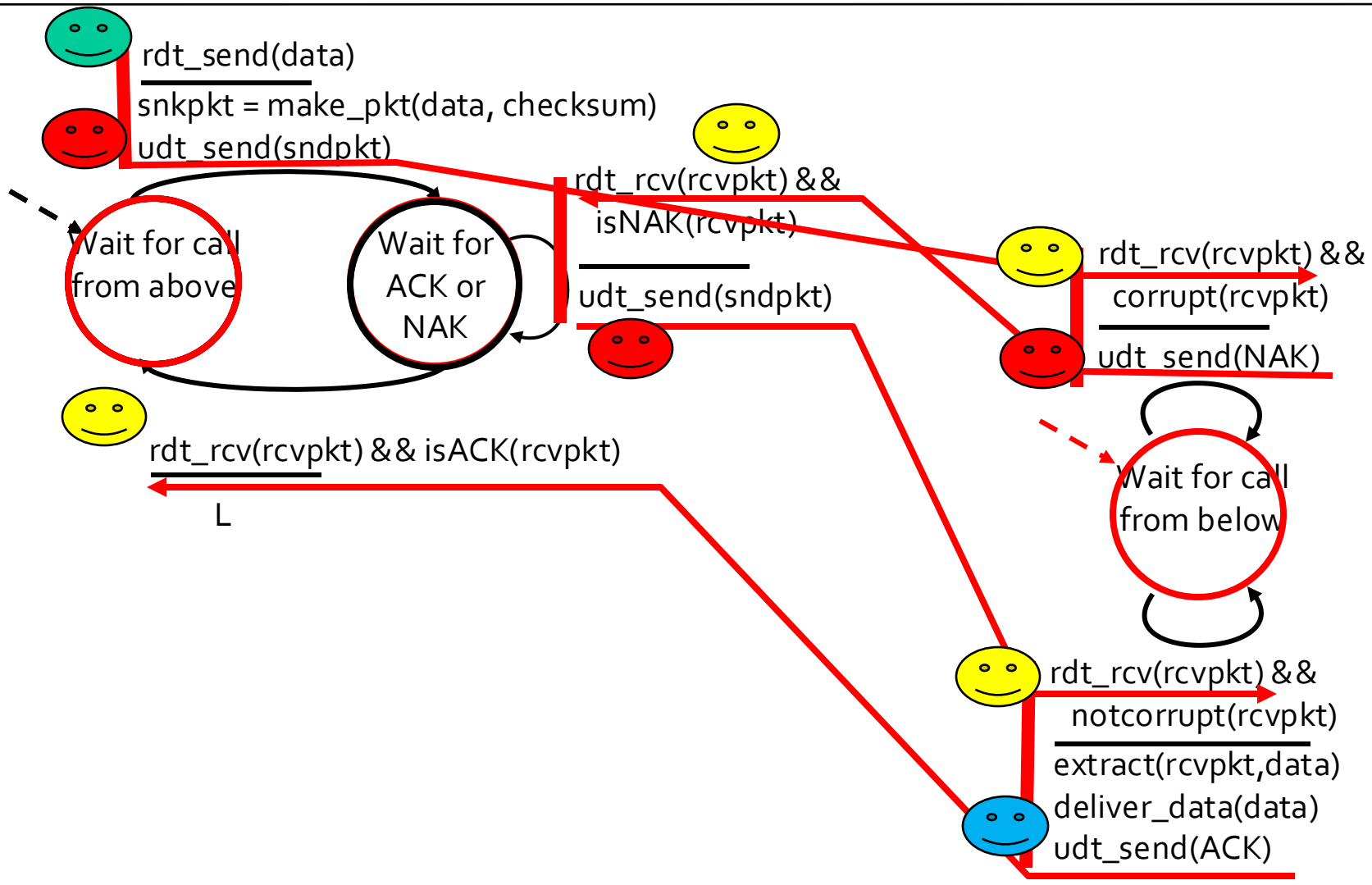
rdt2.0: error scenario



rdt2.0: error scenario



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK is ~~lost~~ or corrupted?

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

handling duplicates:

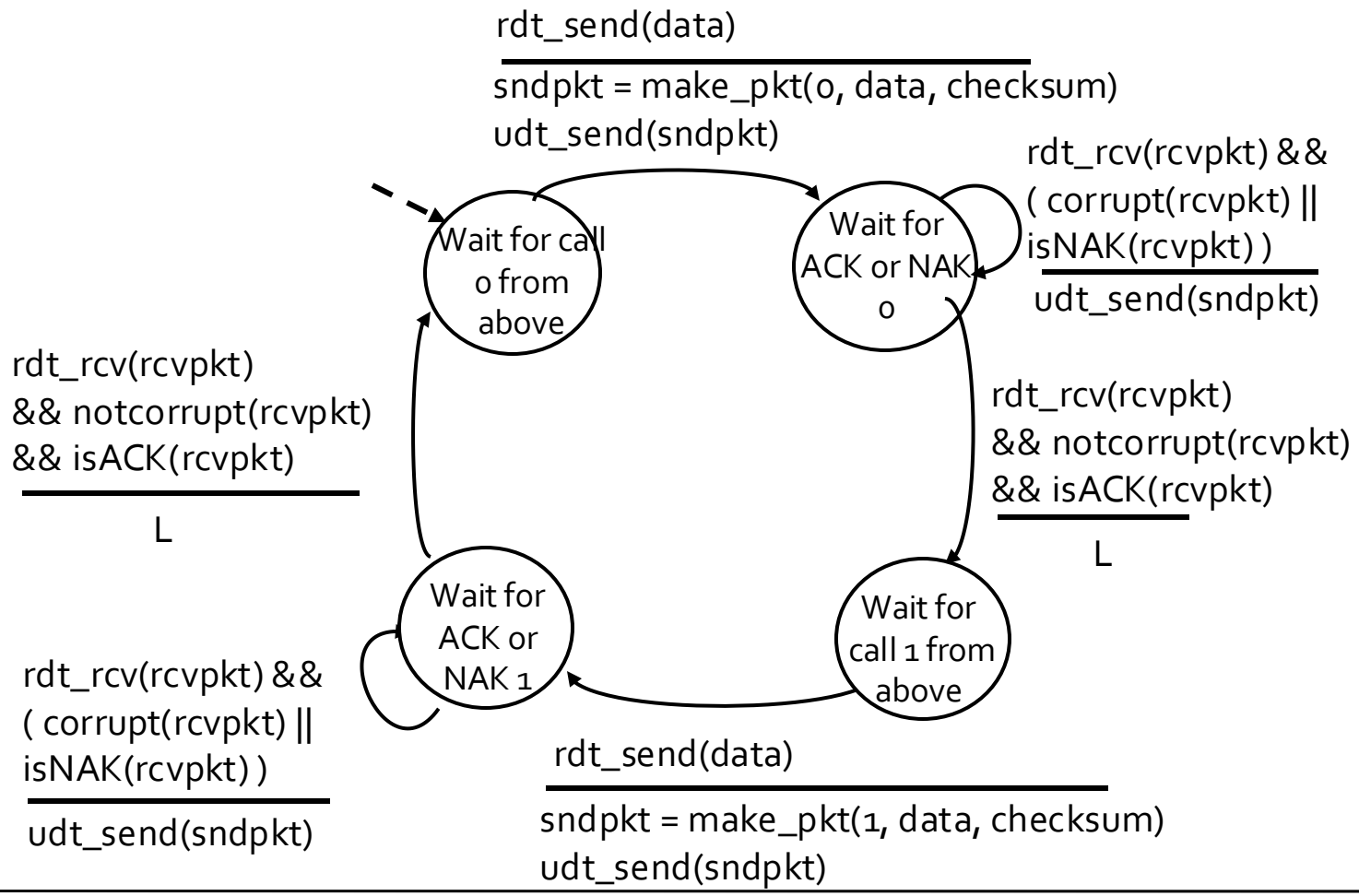
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

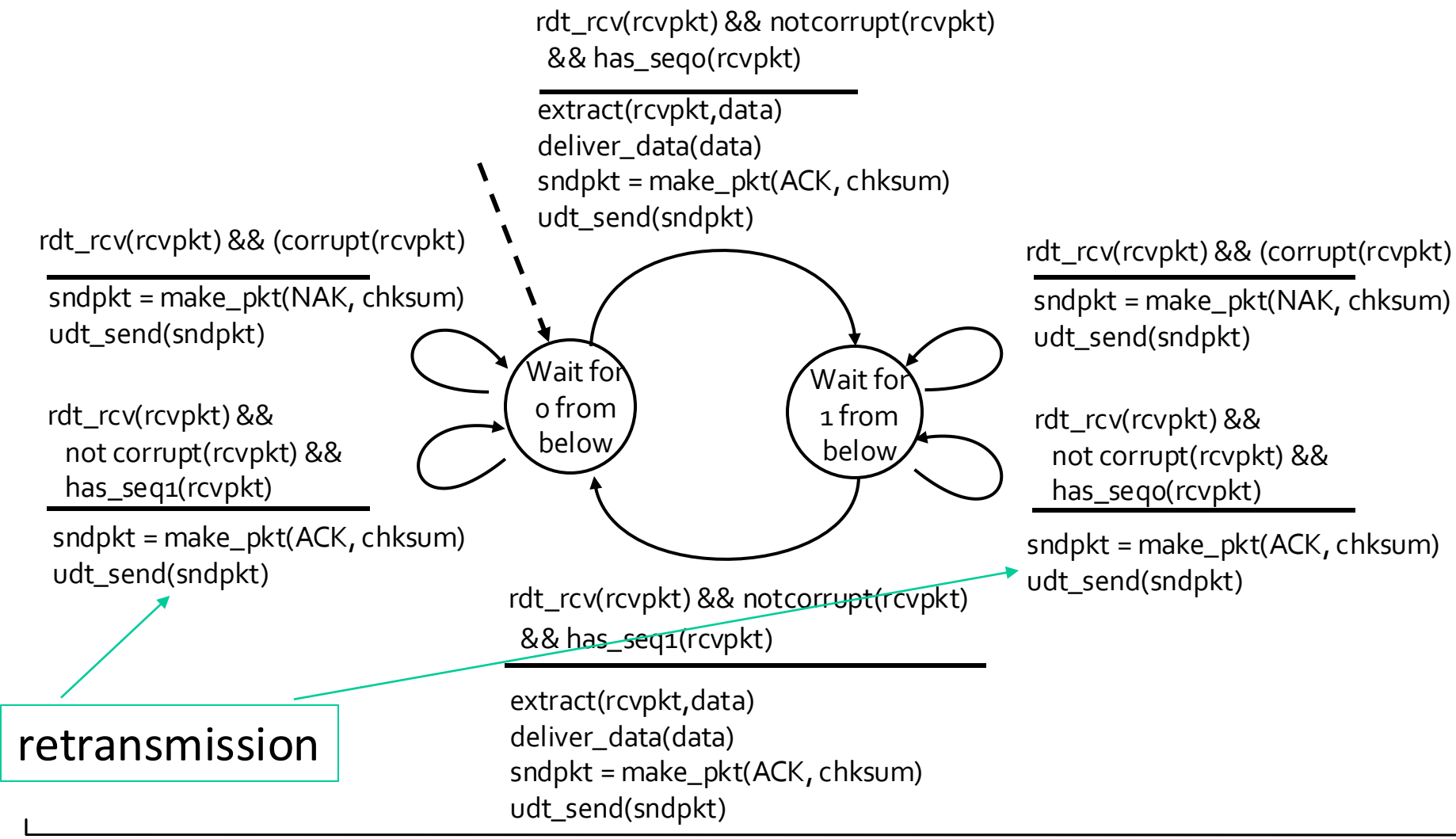
sender sends one packet, then waits for receiver response

rdt2.1: sender handles distorted ACK/NAKs

The protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1



rdt2.1: receiver handles distorted ACK/NAKs



rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice.
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

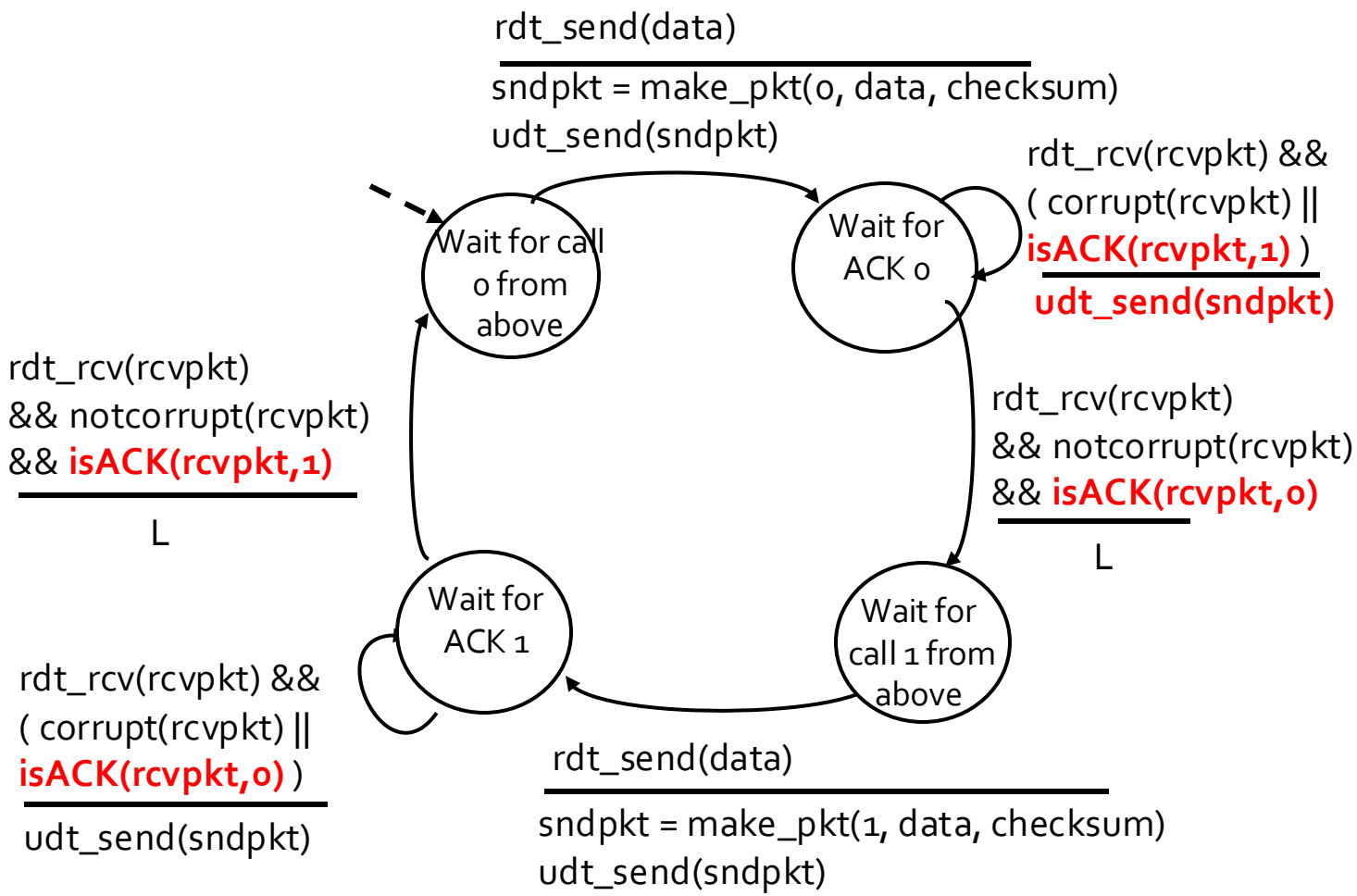
receiver:

- 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

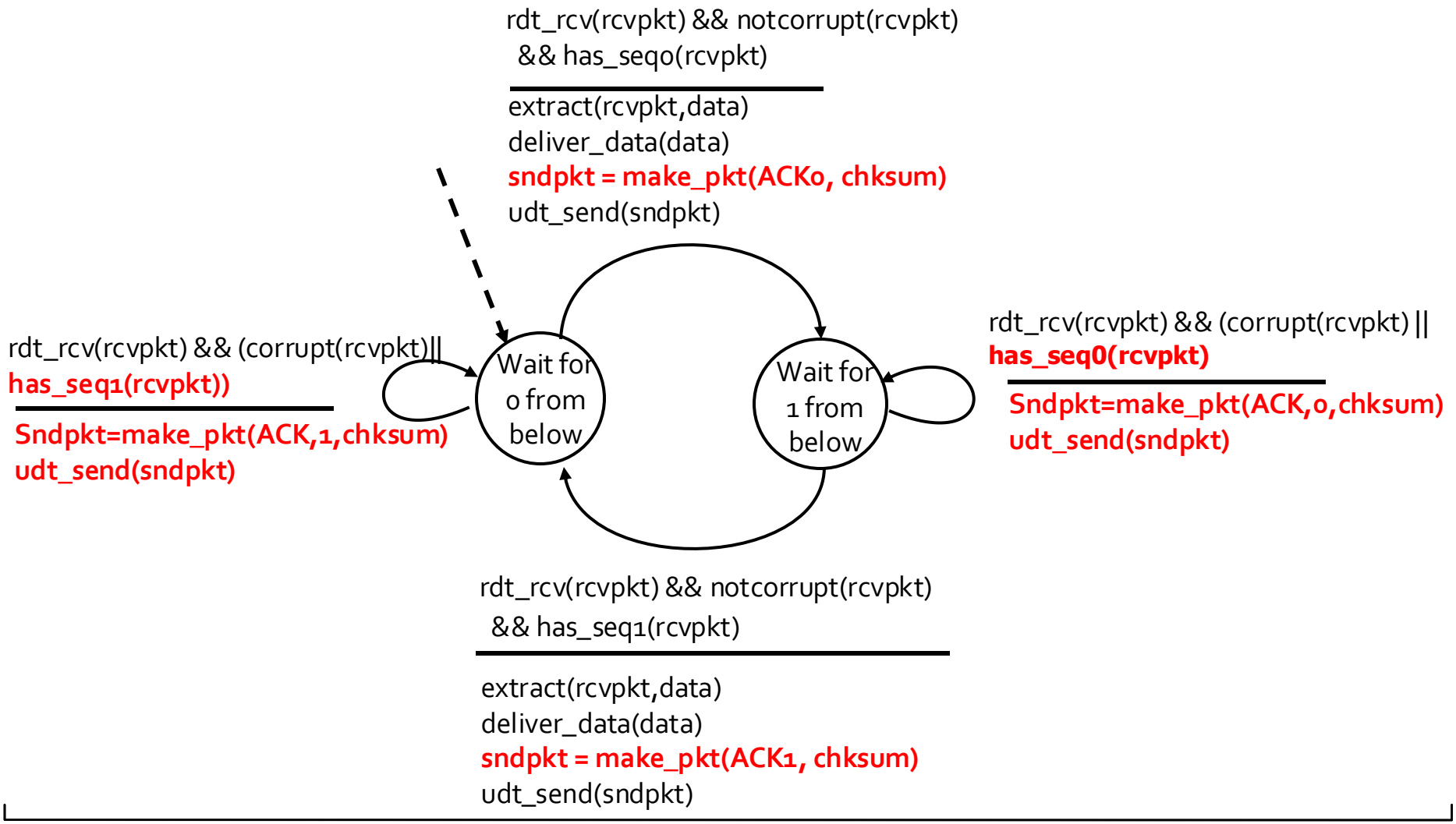
rdt2.2: a NAK-free protocol

- 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: NAK-free sender



rdt2.2: NAK-free receiver



Summary

Today:

- Transport layer services
- Port number, encapsulation/decapsulation, multiplexing/ demultiplexing
- Transport protocols:
  - Reliable channel – Simple, rdt1.0
  - Unreliable channel – Stop-and-Wait

Canvas discussion:

- Reflection
- Exit ticket

Next time:

- read 3.4 and 3.5 of K&R
- follow on Canvas! material and announcements

Any questions?