Chapter 3
Transport Layer

A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in powerpoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach Featuring the Internet,
Jim Kurose, Keith Ross
Addison-Wesley

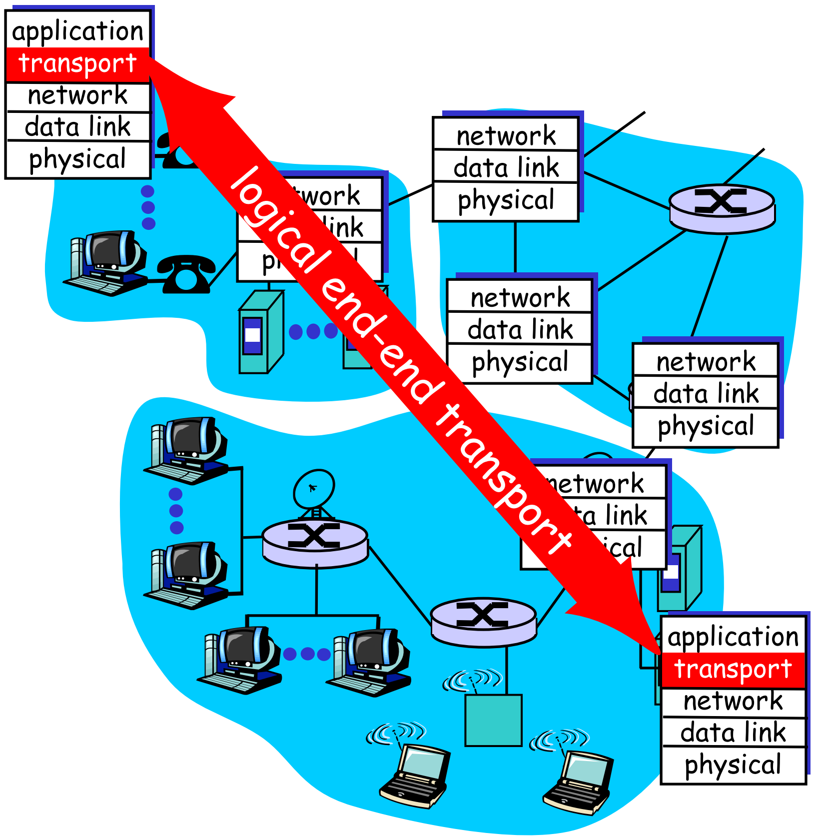
Chapter 3: Transport Layer

Our goals:

- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control - basic notes

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

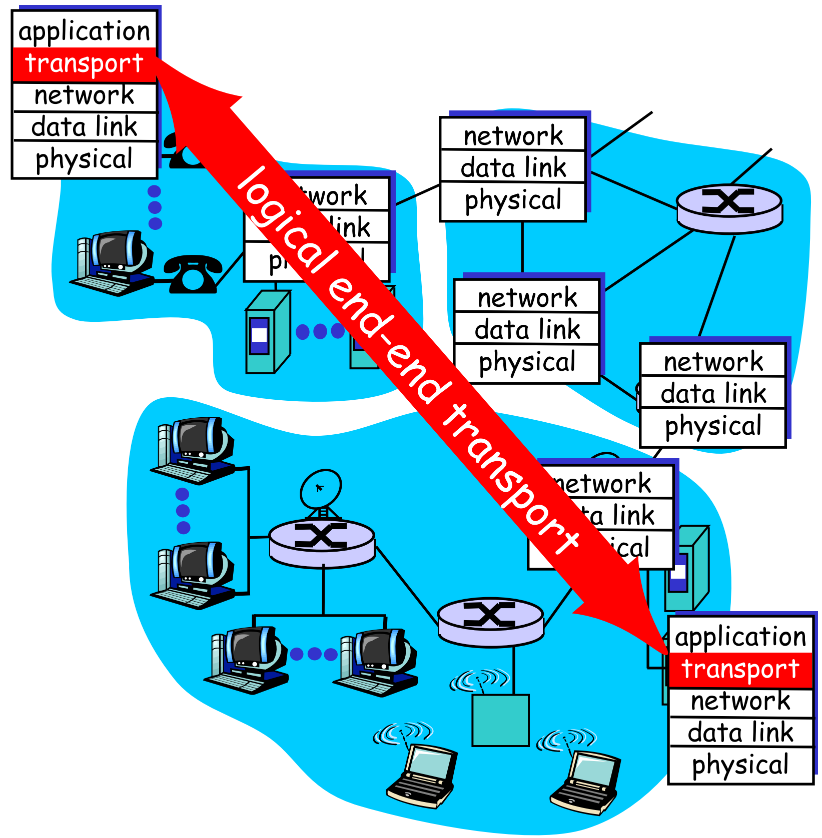
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



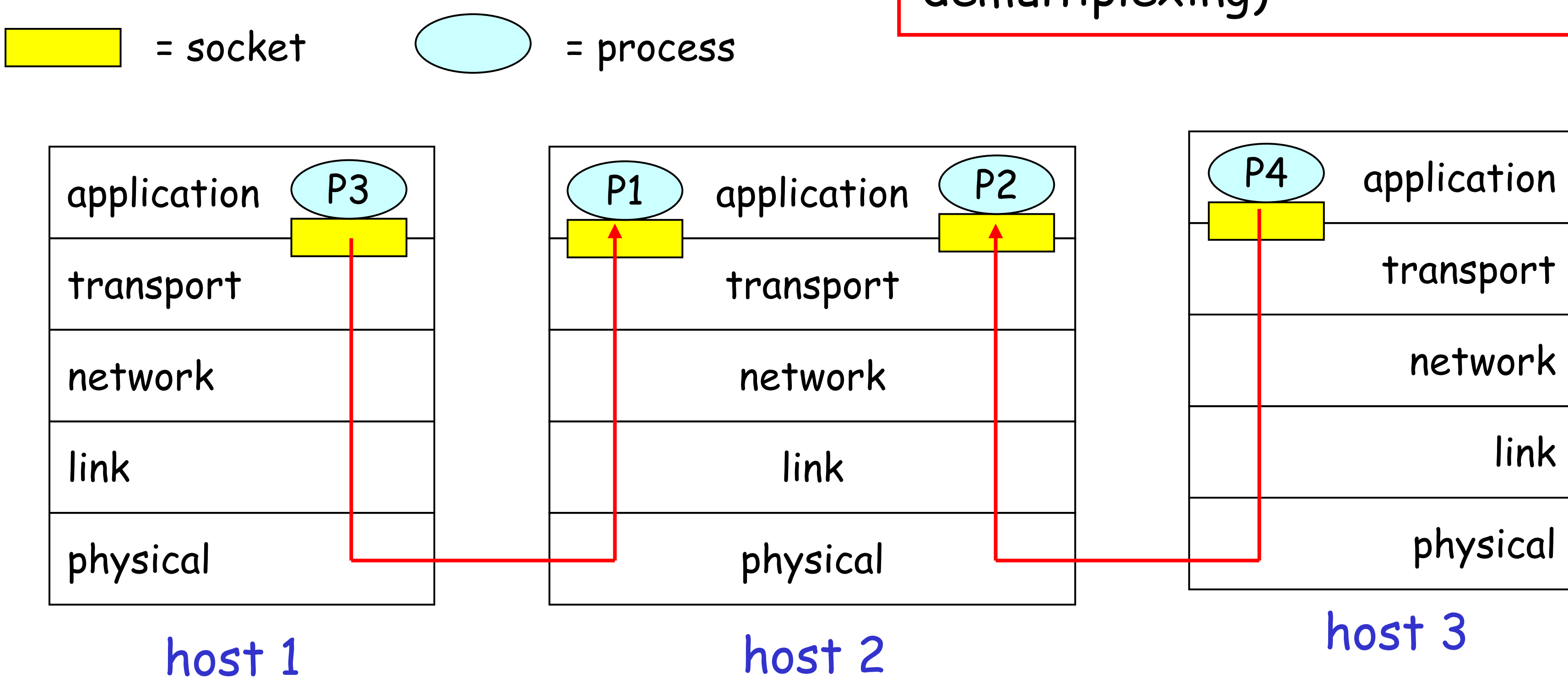
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

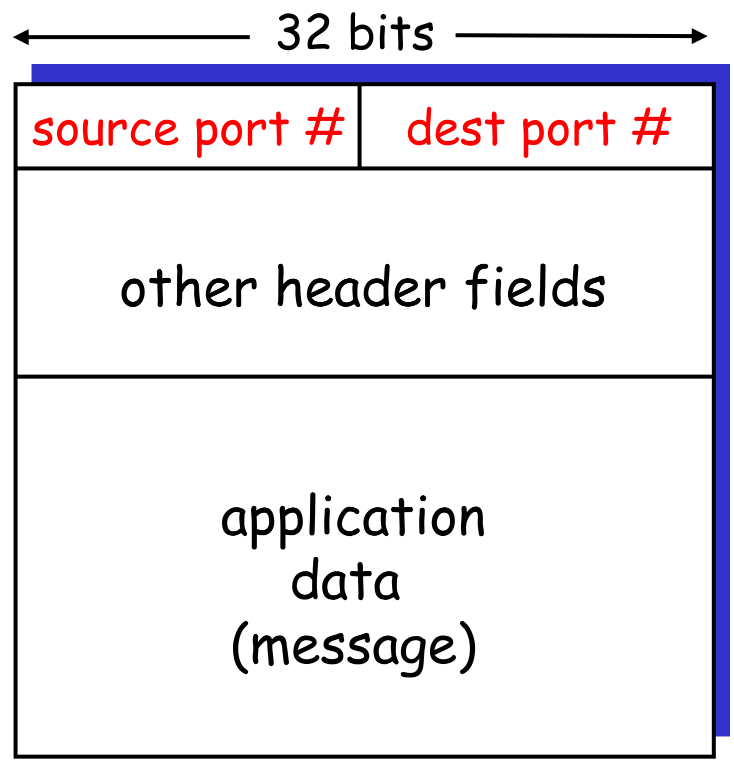
Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
clientSocket1 = socket(AF_INET, SOCK_DGRAM)

clientSocket1.bind(("", 1220))

clientSocket2 = socket(AF_INET, SOCK_DGRAM)

clientSocket2.bind(("", 1221))
```

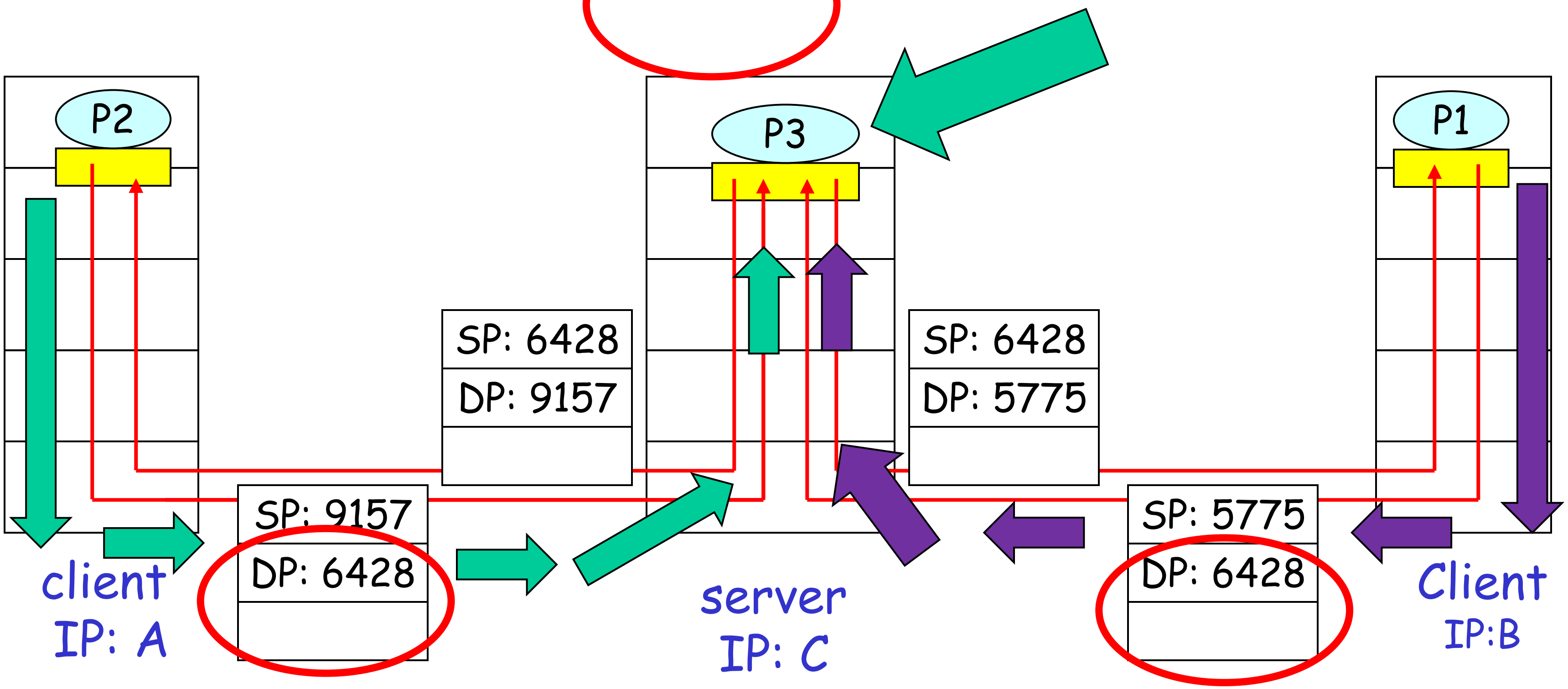
UDP socket identified by two-tuple:

(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

Connectionless demux (cont)

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", 6428))
```



SP provides "return address"

UDP socket identified by two-tuple:

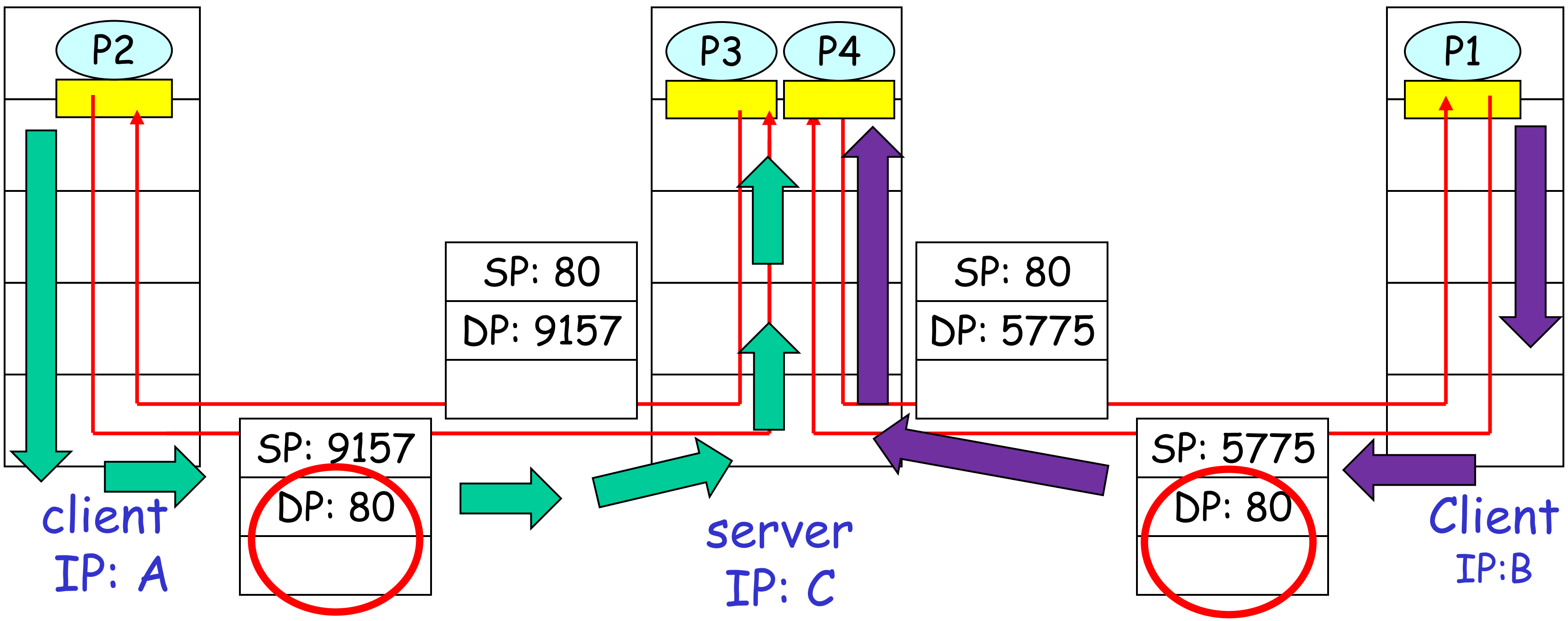
(dest IP address, dest port number)

All segments with same DP routed to same socket

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client

Connection-oriented demux (cont)



TCP socket identified by 4-tuple: source IP address, source port number, dest IP address, dest port number

Segments with same DP can be routed to different sockets!

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

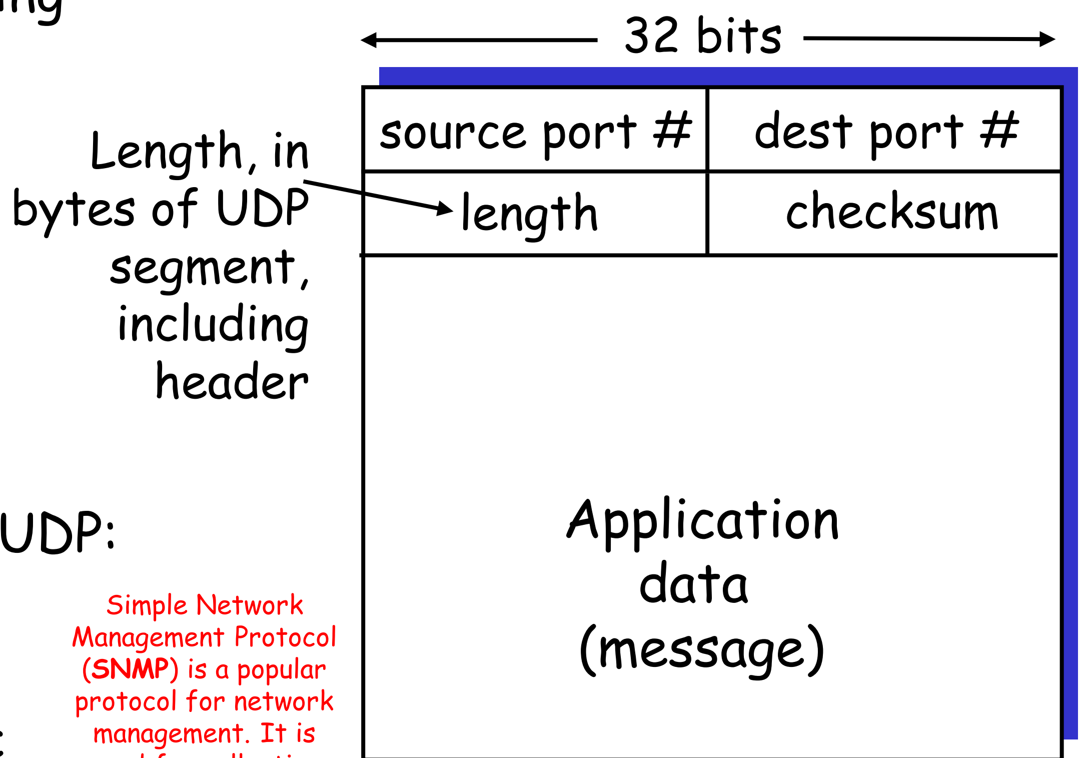
Why is there a UDP?

- 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: more

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

Simple Network Management Protocol (**SNMP**) is a popular protocol for network management. It is used for collecting information from, and configuring, network devices, such as servers, printers, hubs, switches, and routers on an Internet Protocol (IP) network. technet



UDP segment format

UDP checksum

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

Sender:

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

Receiver:

- All message data and checksum added together
- Result should be all 1s
- There may still be errors if two bits may alternately flip

Checksum

Sender

```
           10101001
           00111001
           --------
Sum        11100010
Checksum:  00011101
```

Data parts

Add data words

Checksum = 1's complement
1->0, 0->1

And then when they calculate if the msg arrived OK. And once again how is the sum calculated?

Receiver

```
           10101001
           00111001
           00011101
           --------
Sum        11111111
Complement 00000000  means that the pattern is O.K.
```

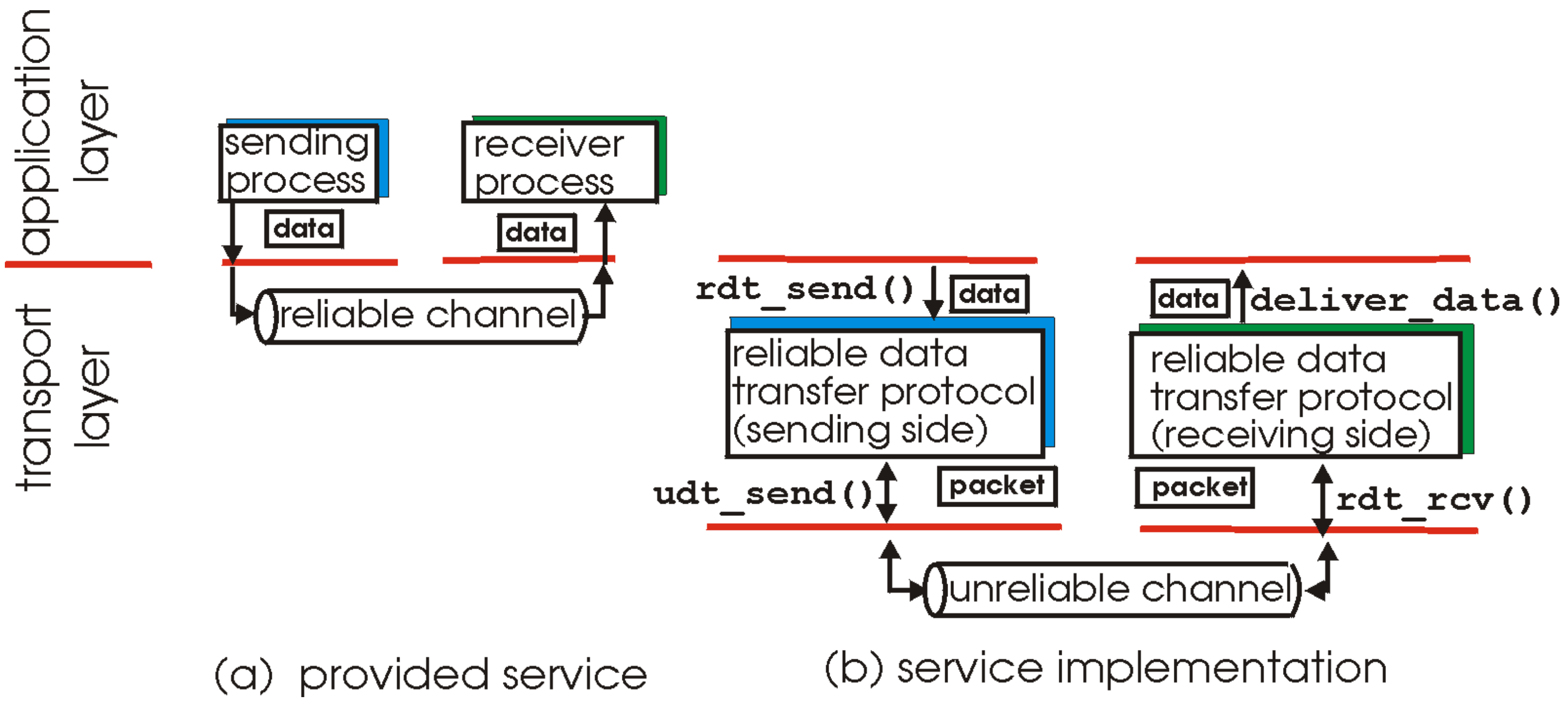
Data parts

Checksum

All 1's expected

Principles of Reliable data transfer

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

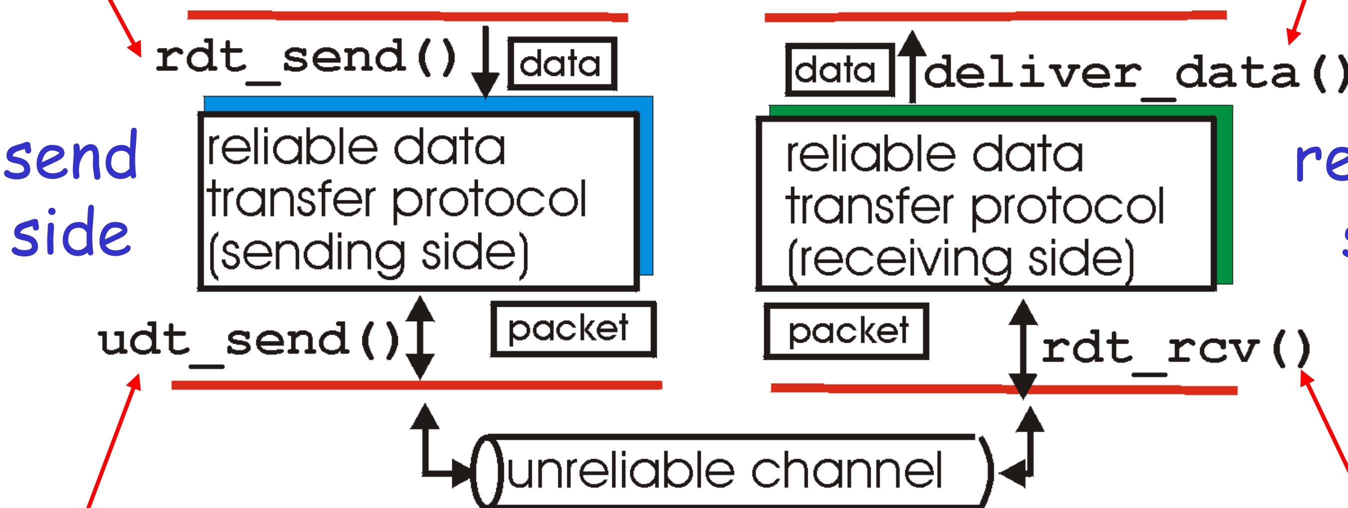


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



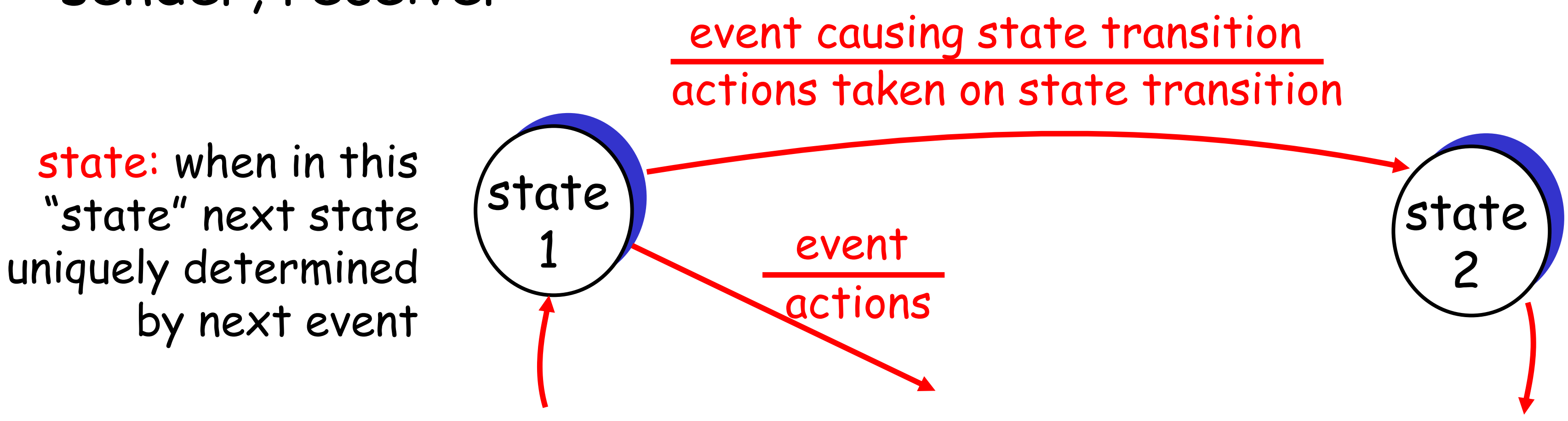
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

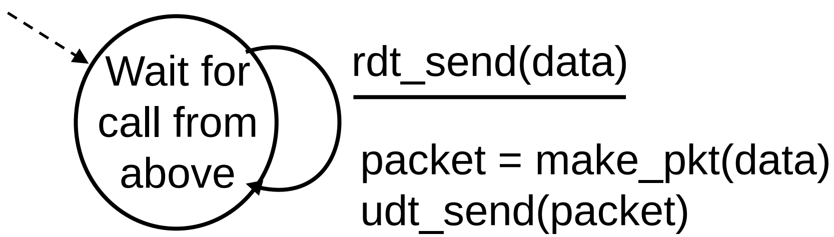
We'll:

- 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

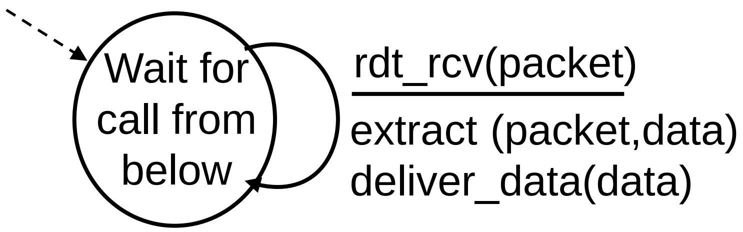


Rdt1.0: reliable transfer over a reliable channel

- 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

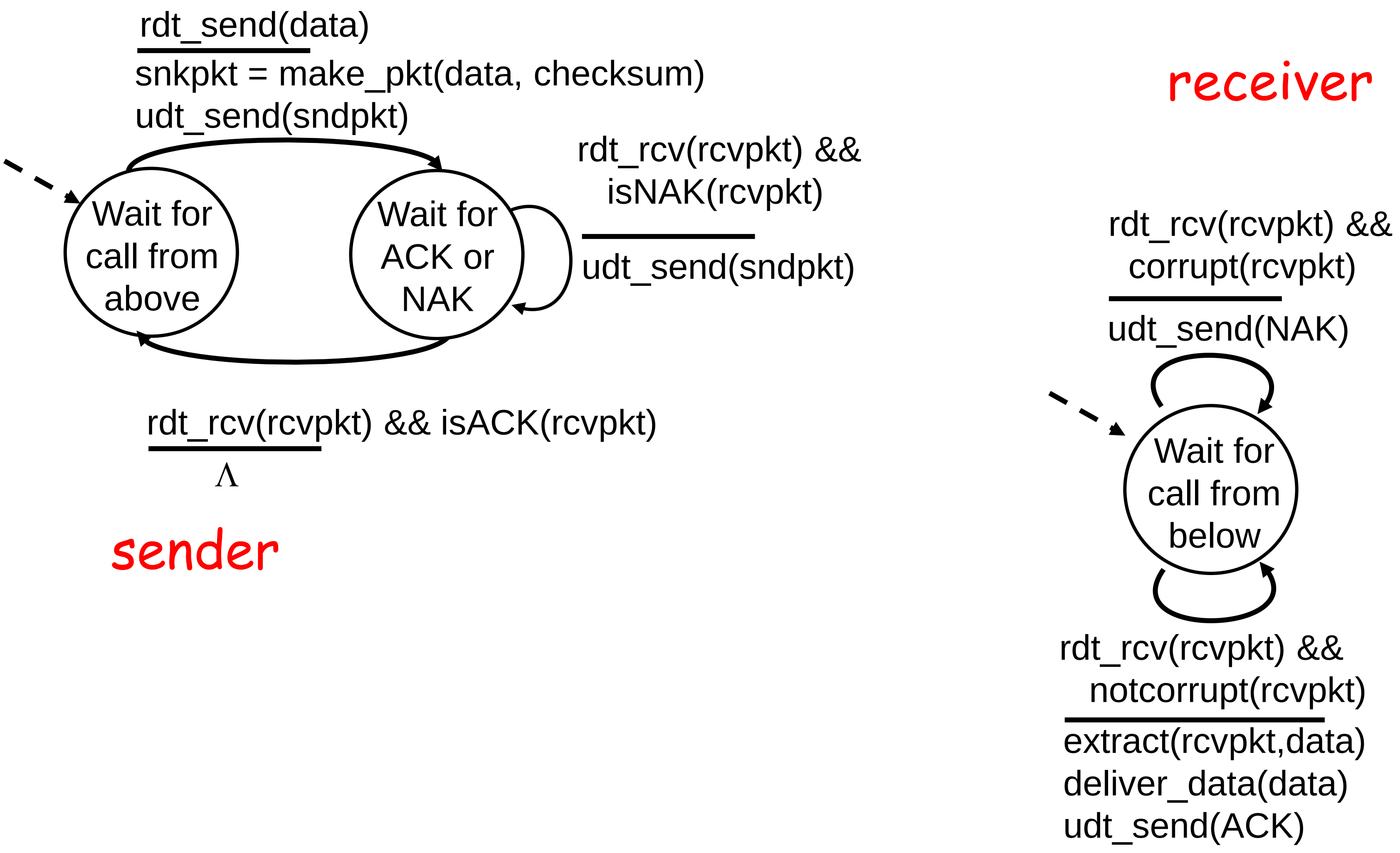


receiver

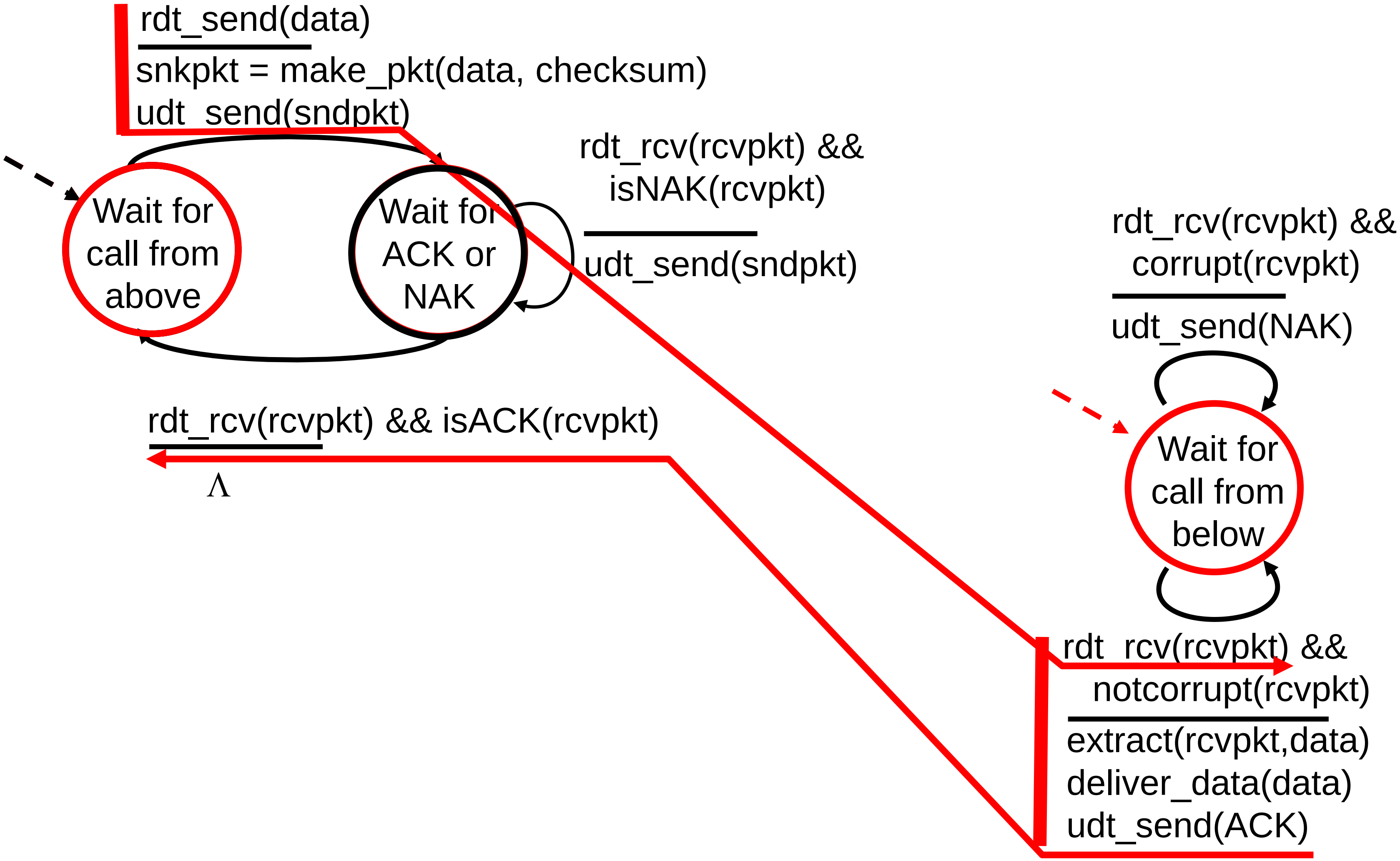
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

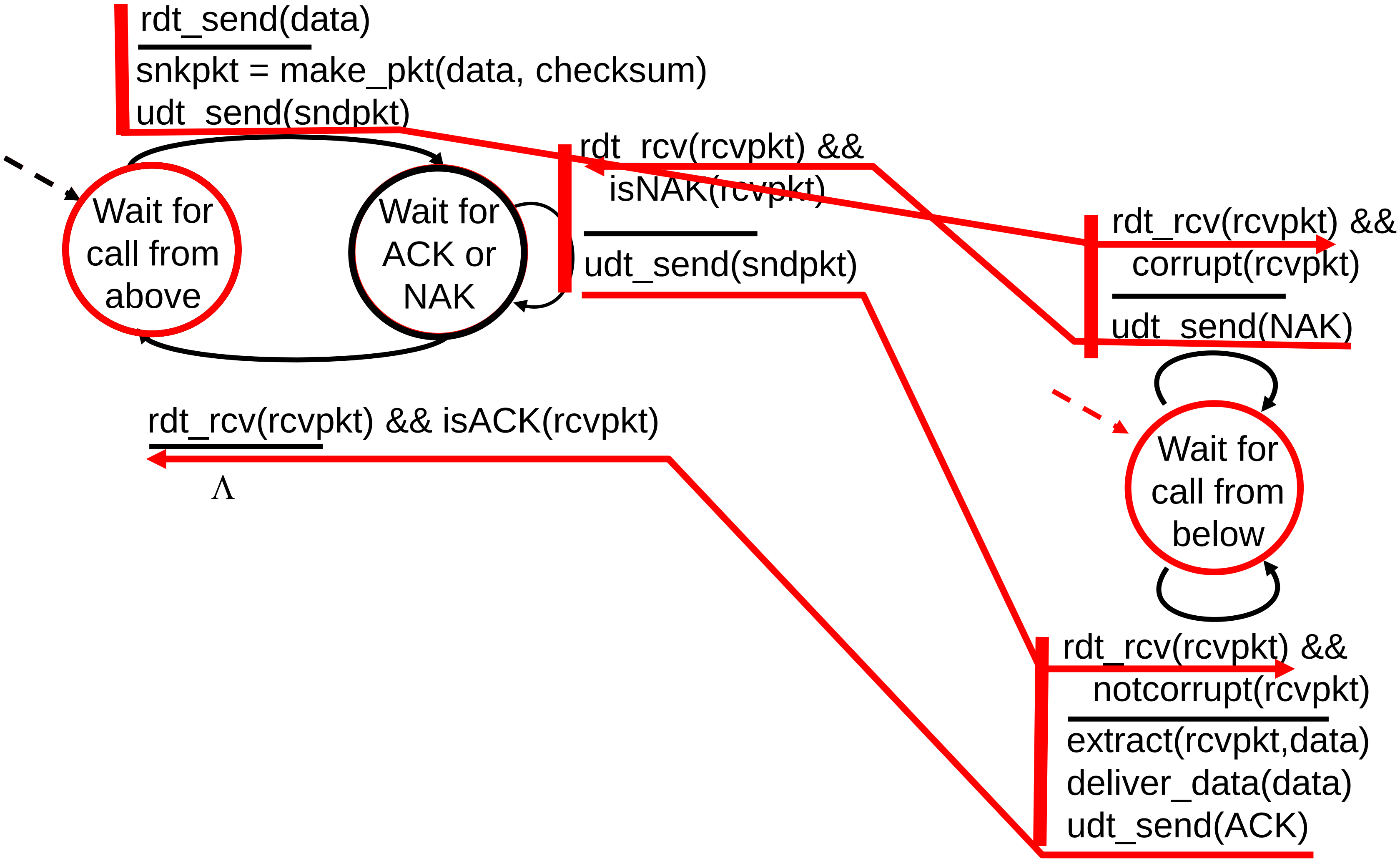
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

Was data received and delivered to app layer(ACK)/ or rejected (NAK)

What happens if ACK/NAK corrupted?

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

What to do?

- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

Can credit a bank account twice for instance!

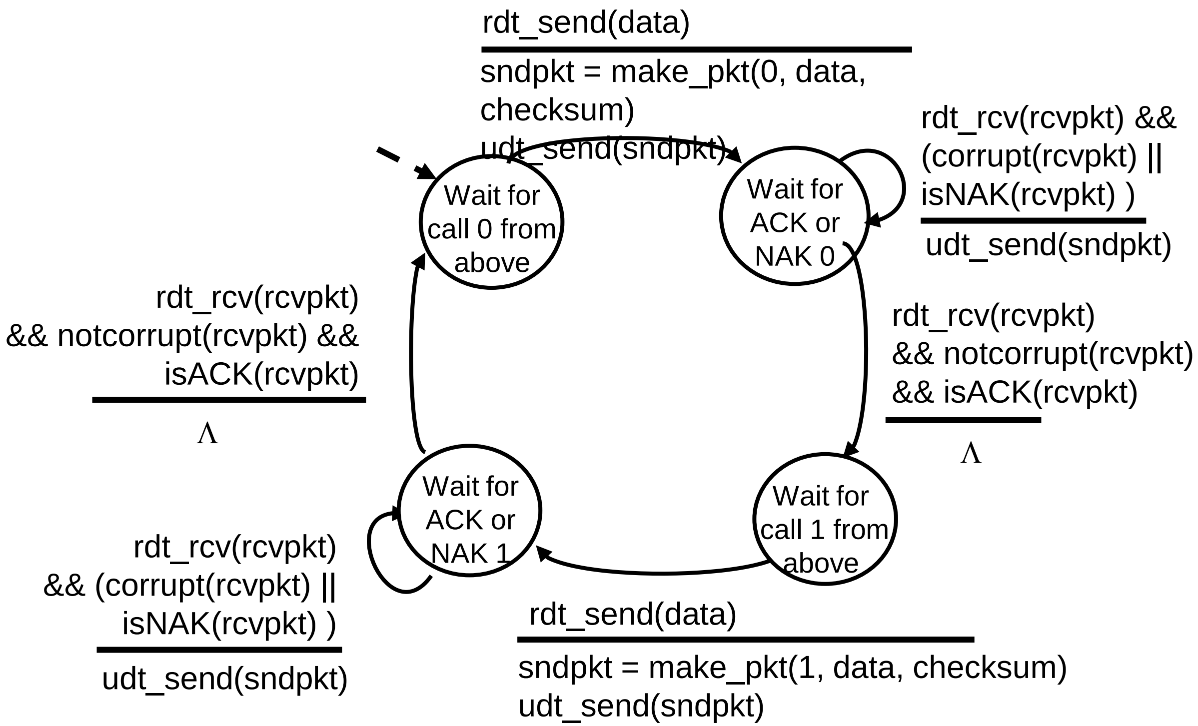
Handling duplicates:

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

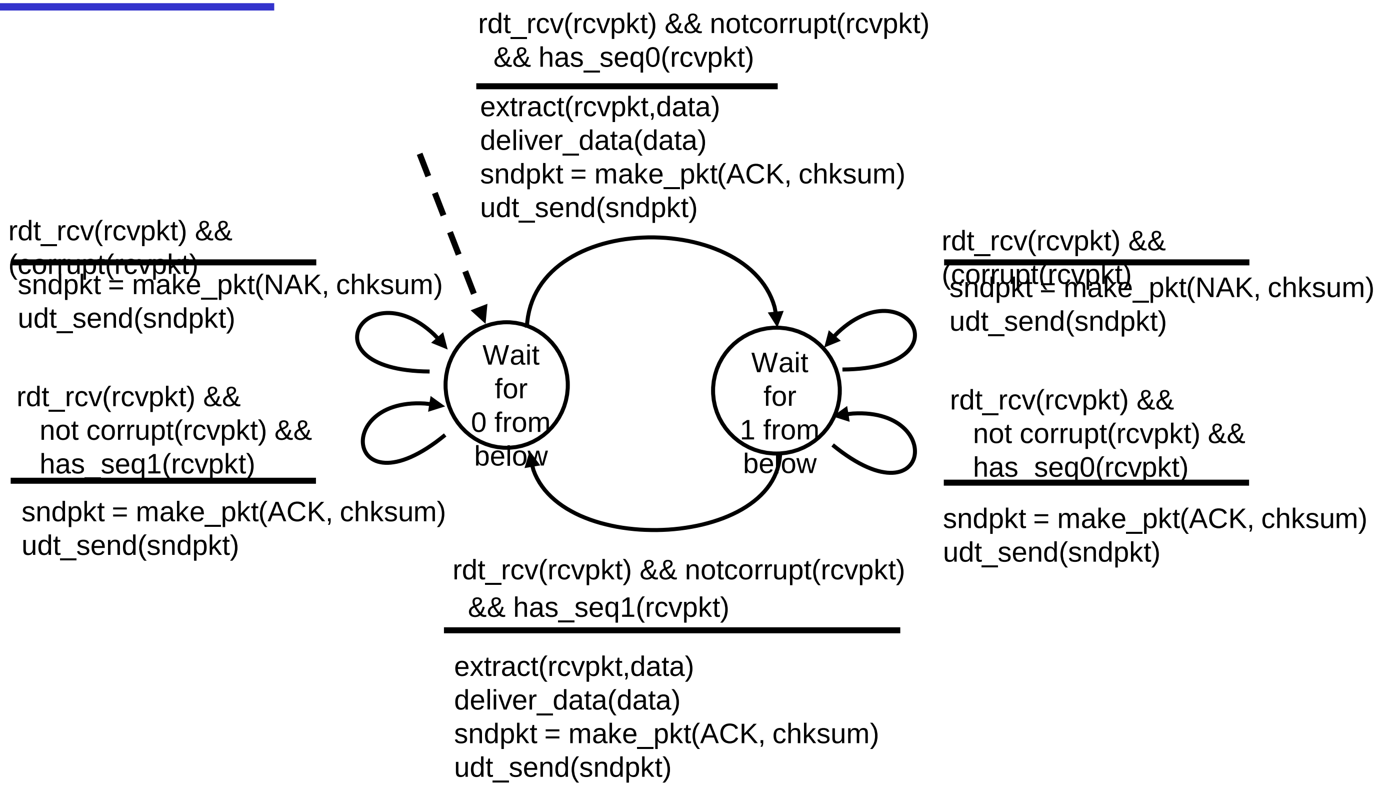
stop and wait

Sender sends one packet, then waits for receiver response

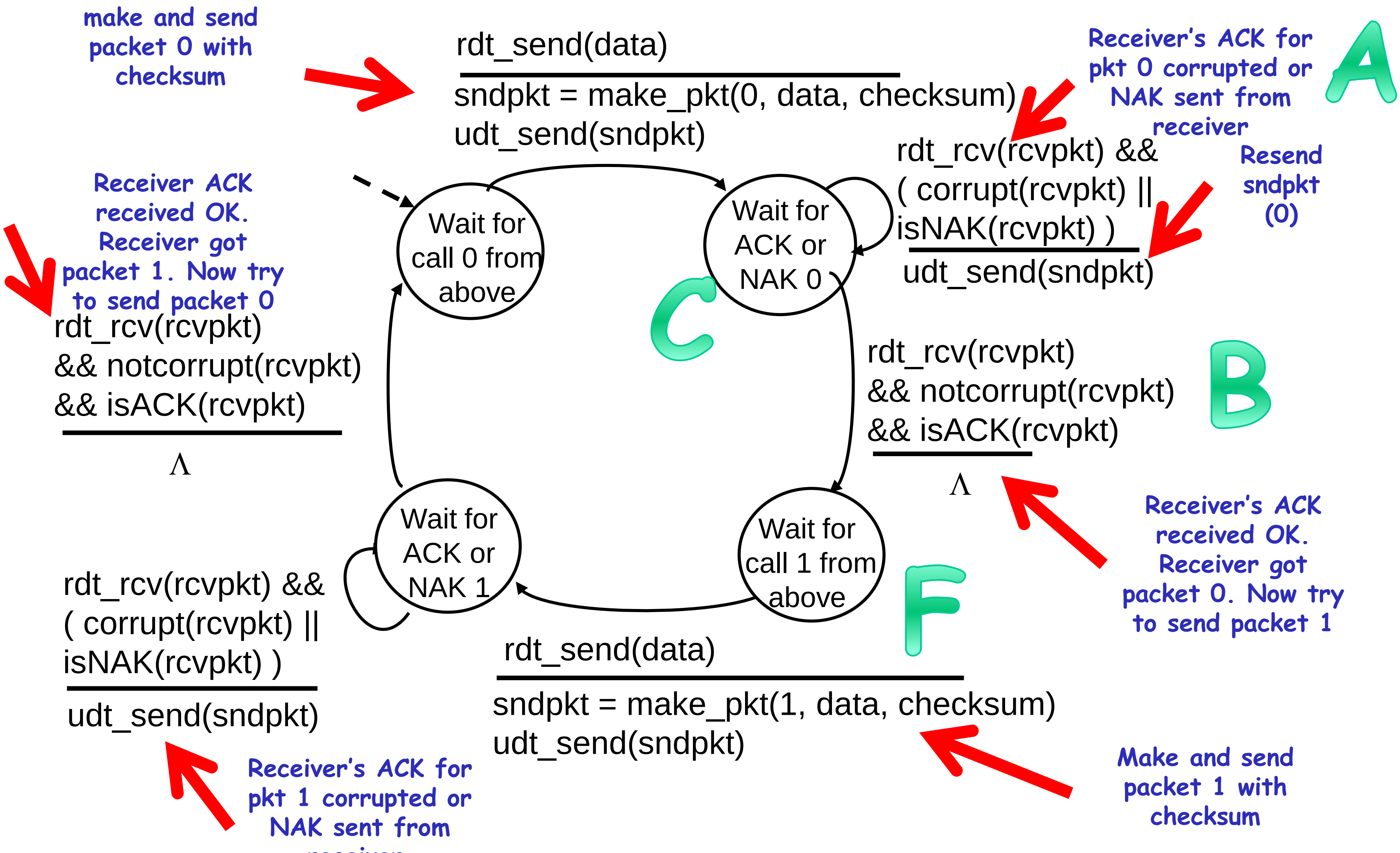
rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs

Answer: D means we are waiting for pkt 1 but sender resends packet 0 because our ACK got corrupted. When sender gets our resent pkt 0 ACK, sender sends pkt 1, which we receive in E and deliver.

Question-Explain D and E in terms of sender and receiver

Receiver processed its packet **0** and sends ACK. Sender can receive ACK **0** correctly or ACK can get corrupted.
What does sender do if ACK 0 corrupt?
Ans-sender resends the **0** packet (**A** in sender).
What does sender do if ACK 0 correct? – it moves on to send packet **1** (**B then F in sender**), pkt 1 is expected by the receiver.

Waiting for pkt 0 and received it so extract and deliver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Waiting for pkt 0 and corrupt, send NAK

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

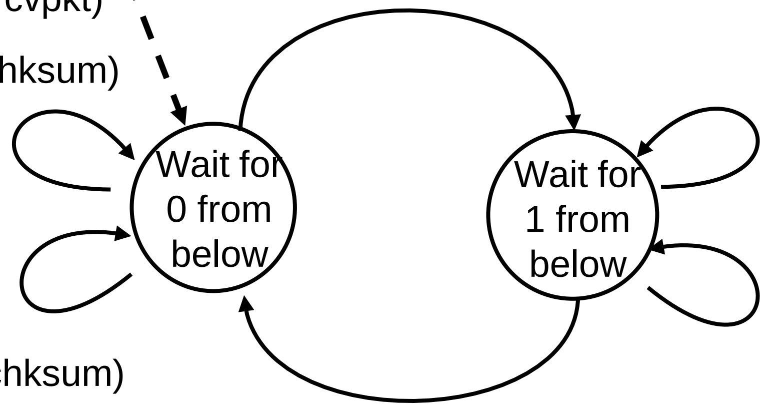
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

Waiting for pkt 1, received and corrupt, send NAK

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

D

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

E

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Waiting for pkt 0 but received pkt 1. This is a retransmission of pkt1 so send ACK 1

Waiting for pkt 1 and received it so extract and deliver

Waiting for pkt 1 but received pkt 0. This is a retransmission of pkt 0 so send ACK 0 (Sender is at C and then moves to B because of receiver's D)

rdt2.1: discussion

Need to distinguish between packet expected and duplicates

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