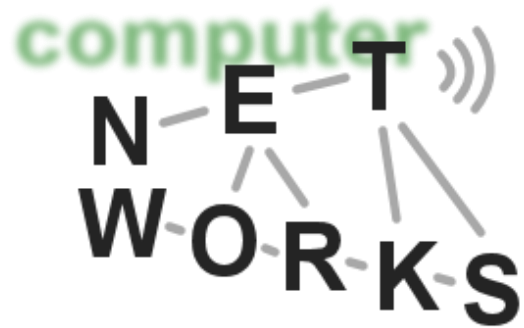


Transport Layer – Part I

Computer Networks, Winter 2013/2014



Chapter 4: The Transport Layer

5: Application Layer

4: Transport Layer

3: Network Layer

2: Link Layer

1: Physical Layer

Chapter 4: The Transport Layer

Our goals:

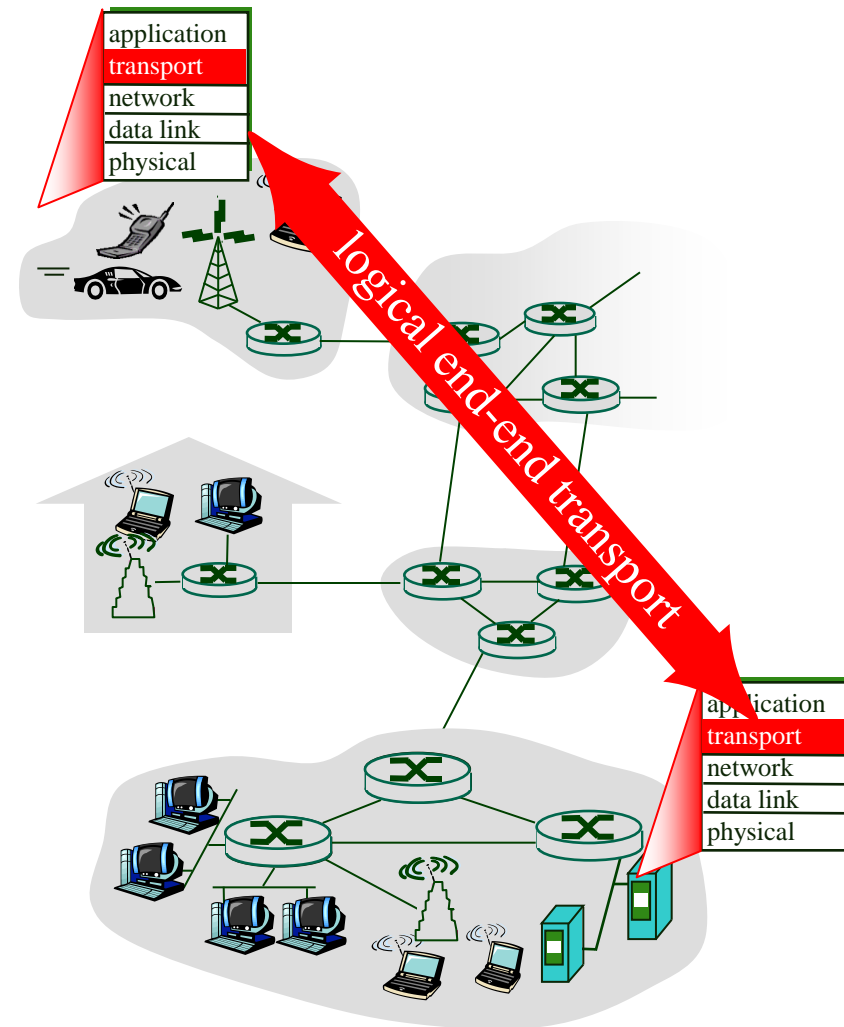
- understand principles behind transport layer services:
 - multiplexing/demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- learn about transport layer protocols in the Internet:
 - UDP: connectionless transport
 - TCP: connection-oriented transport
 - TCP congestion control

Transport Layer

- 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

Transport services and protocols

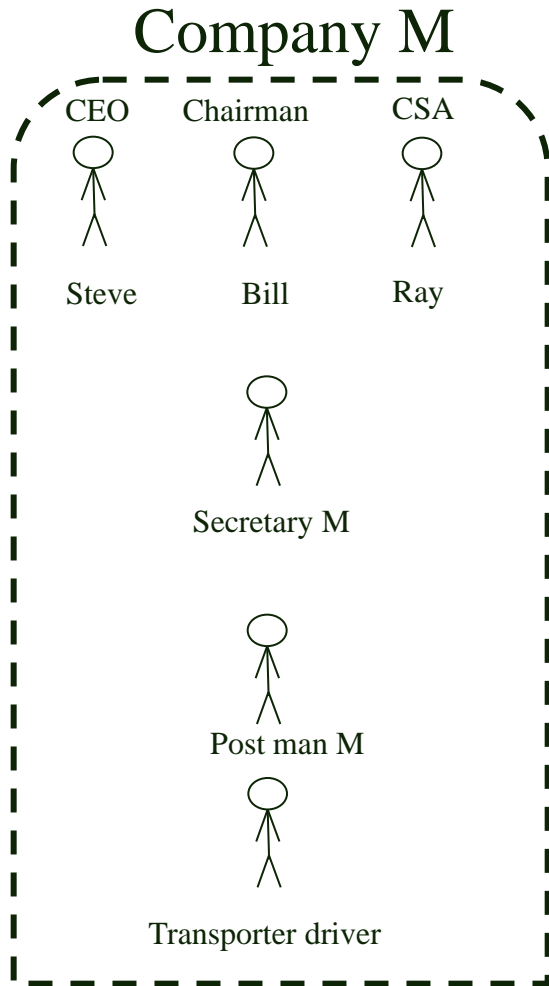
- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
 - send side: breaks app messages into **segments**, passes to network layer
 - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol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



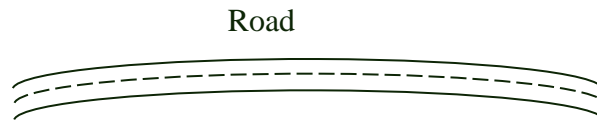
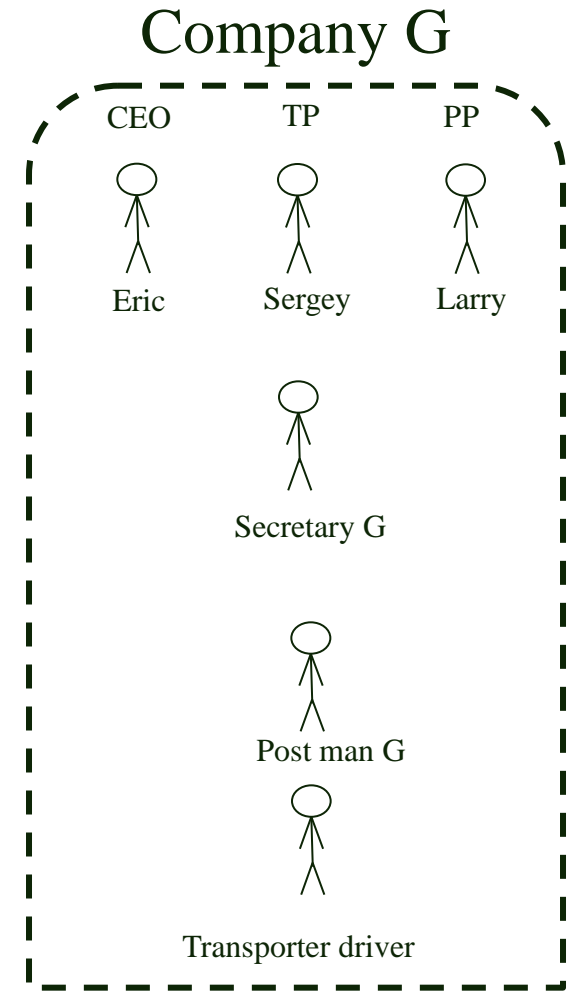
Transport vs. network layer

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

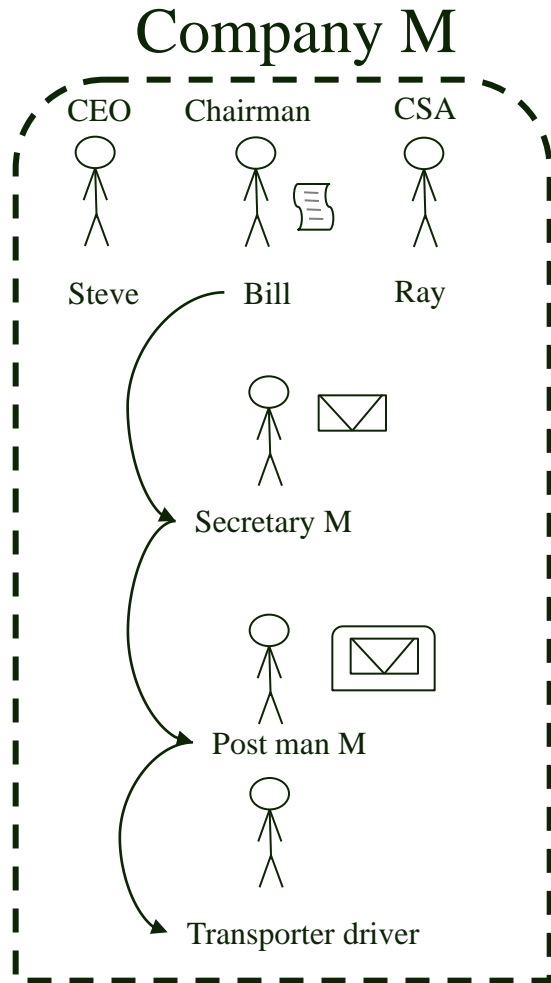
Transport Protocol: Analogy



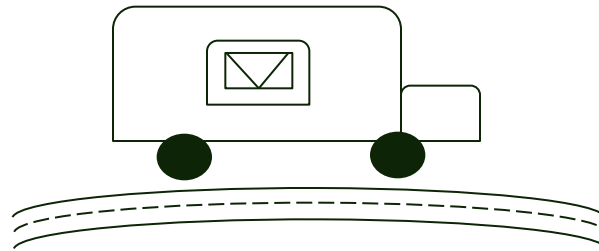
CEO = Chief Executive Officer
CSA = Chief Software Architect
TP = Technology President
PP = Products President



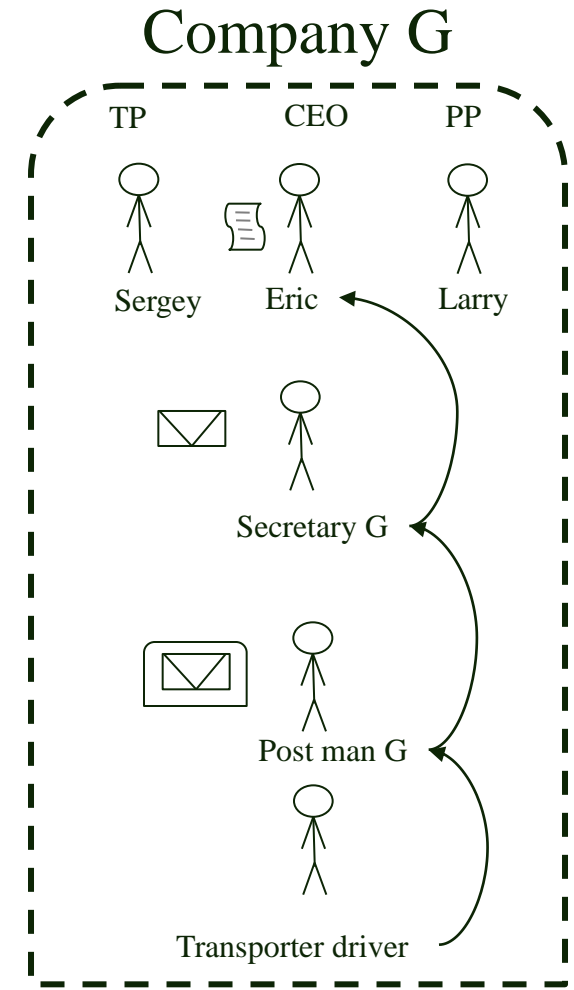
Transport Protocol: Analogy



CEO = Chief Executive Officer
CSA = Chief Software Architect
TP = Technology President
PP = Products President



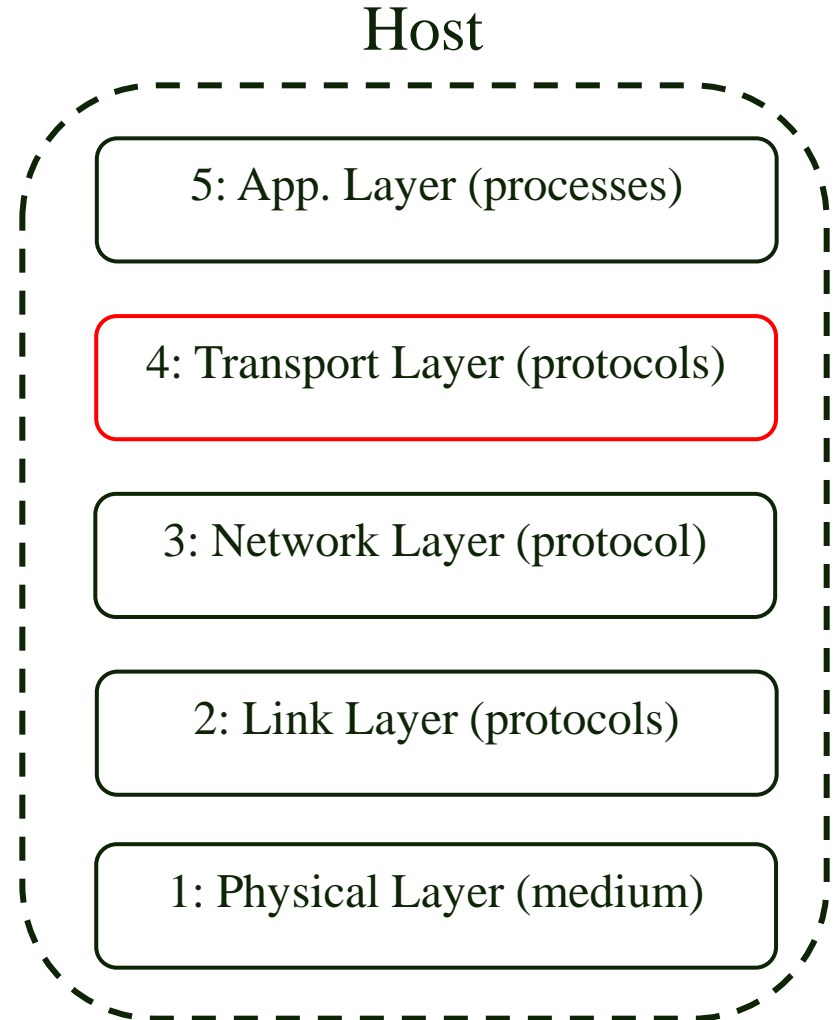
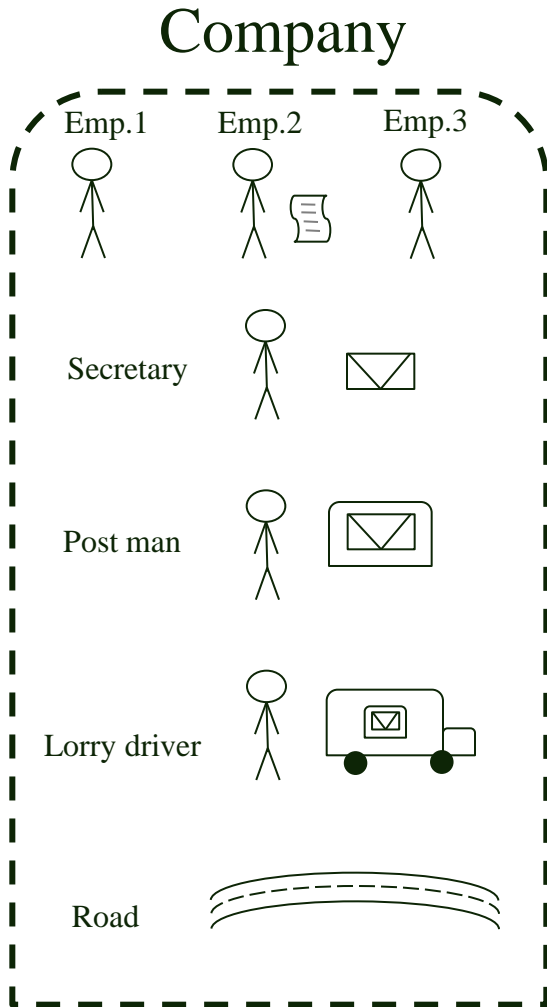
Road



Transport Protocol: Analogy

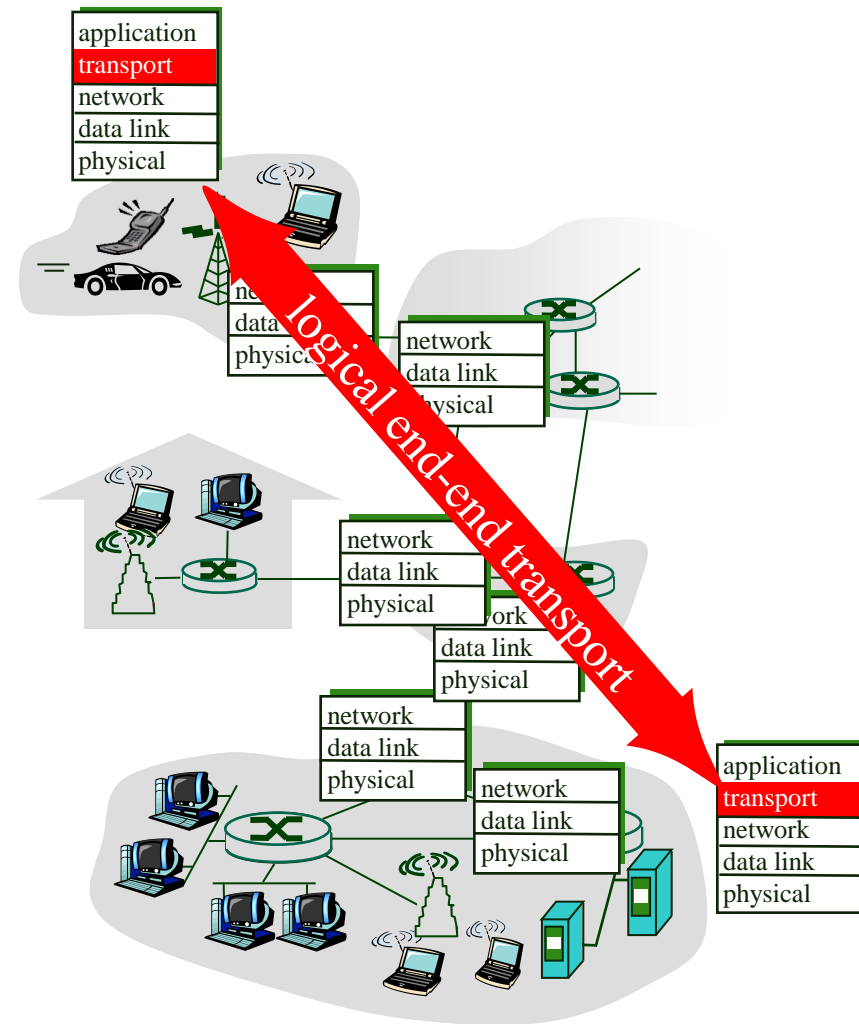
- *Postal service (Network Layer):* logical communication between company buildings.
- *Secretary service (Transport Layer):* logical communication between employees of G und M.
 - relies on & enhances, postal services

Transport Protocol: Analogy



Internet transport-layer protocols

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



Excursus: Sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
 - unreliable datagram
 - reliable, byte stream-oriented

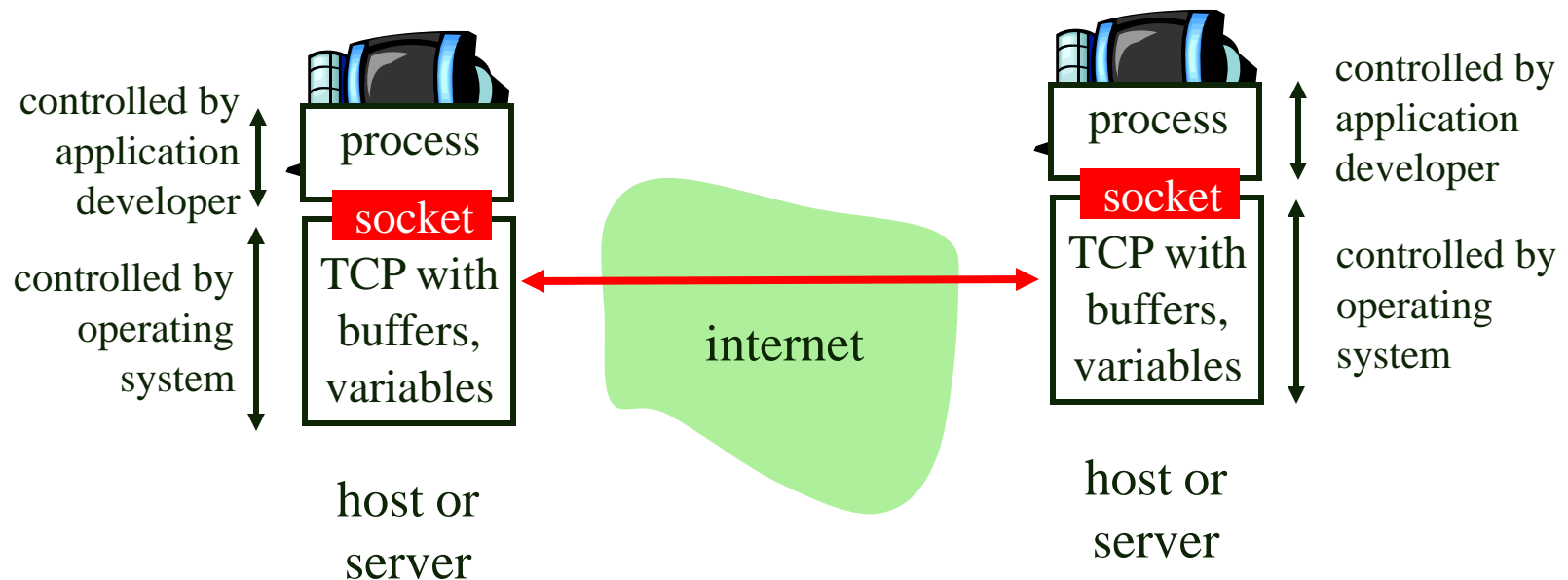
socket

a *host-local*,
application-created,
OS-controlled interface (a
“door”) into which
application process can **both**
send and
receive messages to/from
another application process

Excursus: Socket programming *with TCP*

Socket: a door between application process and end-end-transport protocol (UDP or TCP)

TCP service: reliable transfer of **bytes** from one process to another



Excursus: Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process
- When **client creates socket**: client TCP establishes connection to server TCP

- When contacted by client, **server TCP creates new socket** for server process to communicate with client
 - allows server to talk with multiple clients
 - source port numbers used to **distinguish** clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Transport Layer

- 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

Multiplexing/demultiplexing

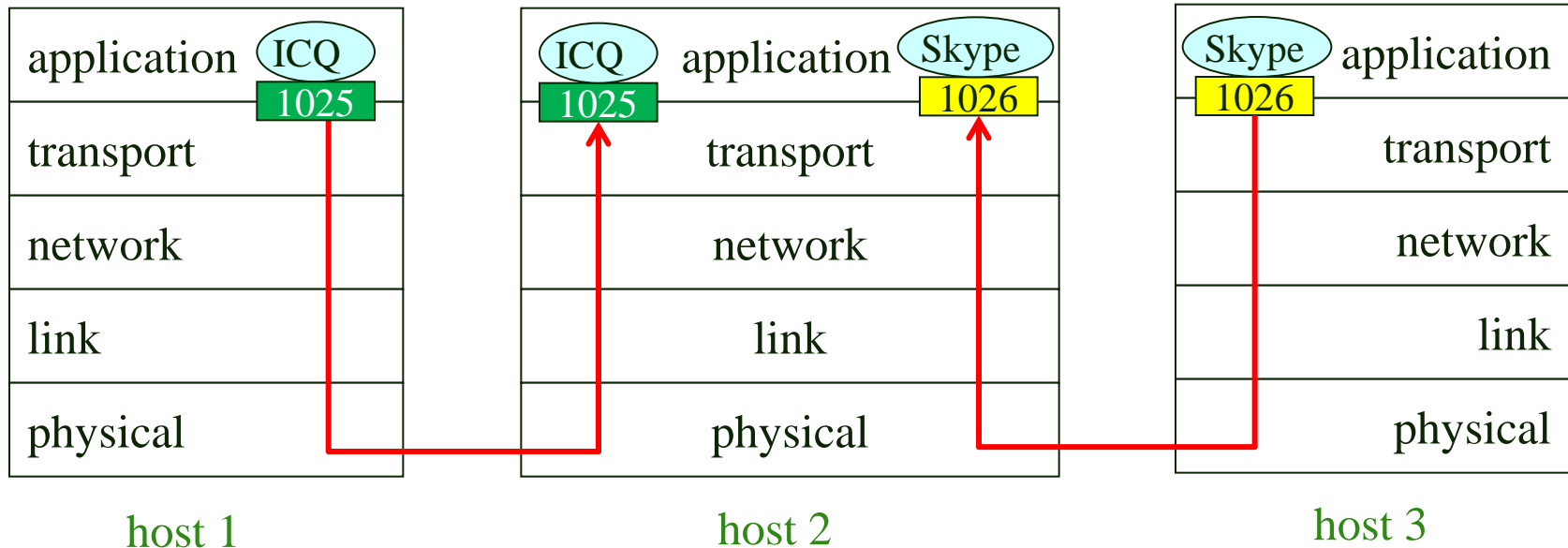
Multiplexing at send host:

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

Demultiplexing at rcv host:

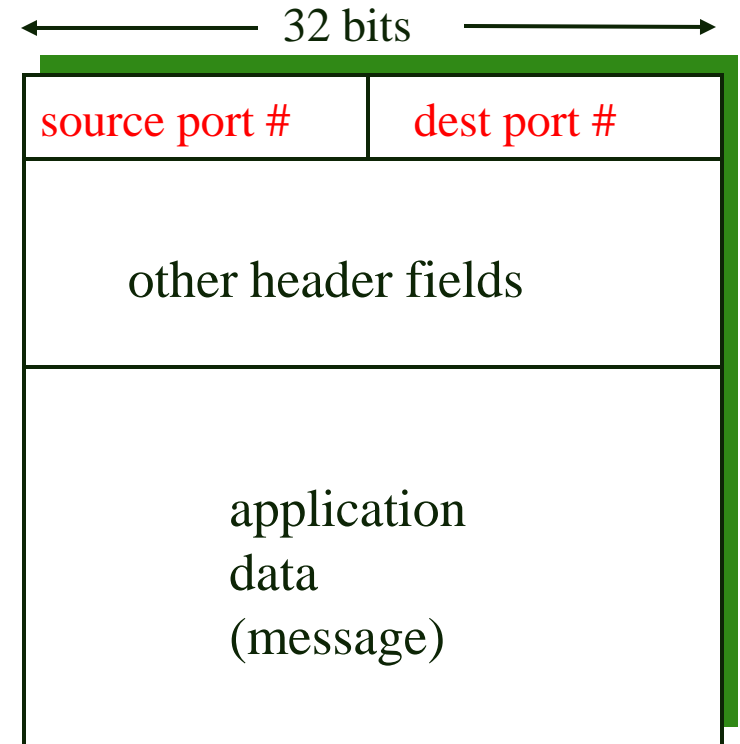
delivering received segments
to correct socket

 = socket  = process



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
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket clientSocket =  
    new DatagramSocket();
```

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

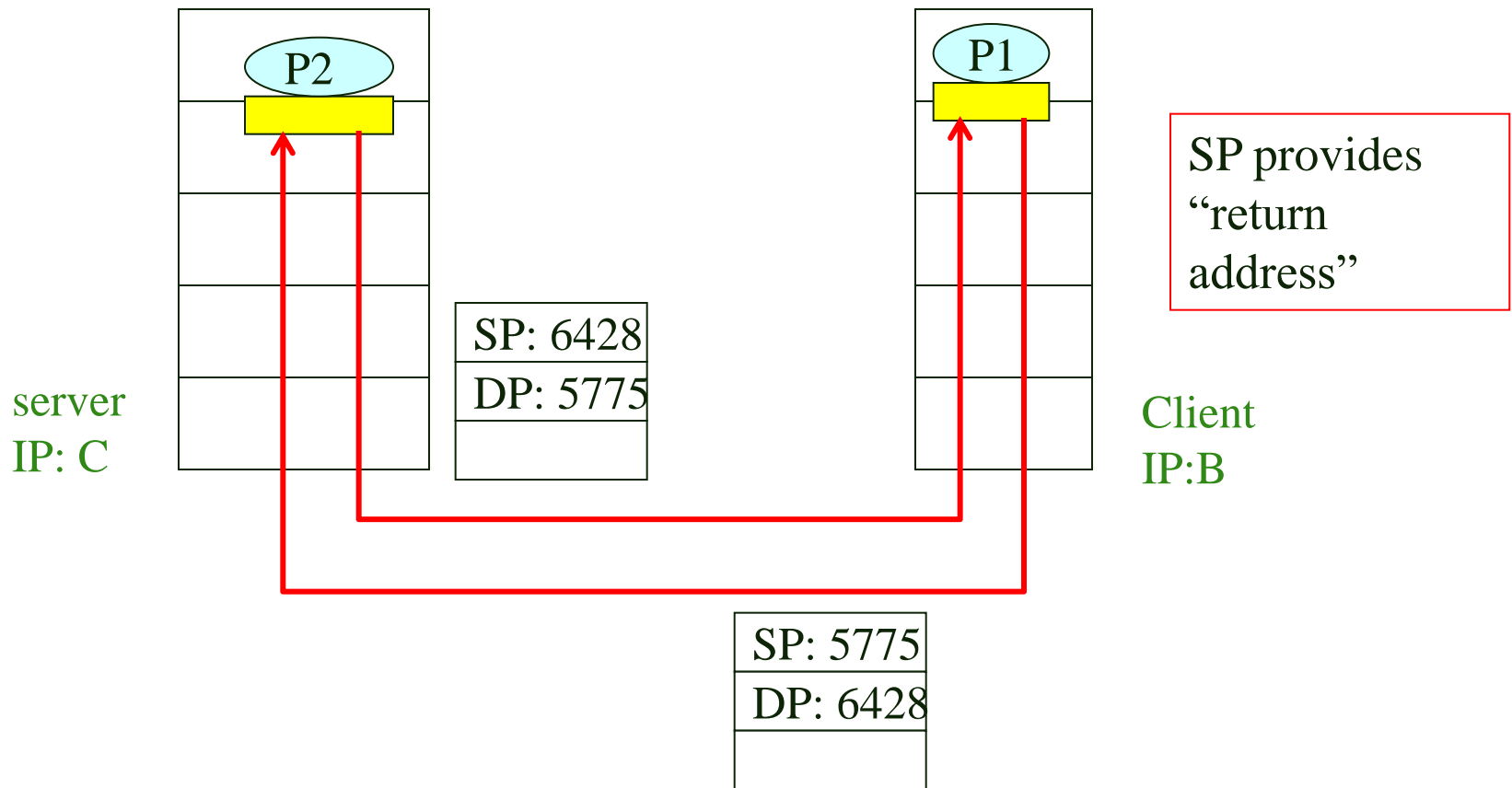
- UDP socket identified by two-tuple:

(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)

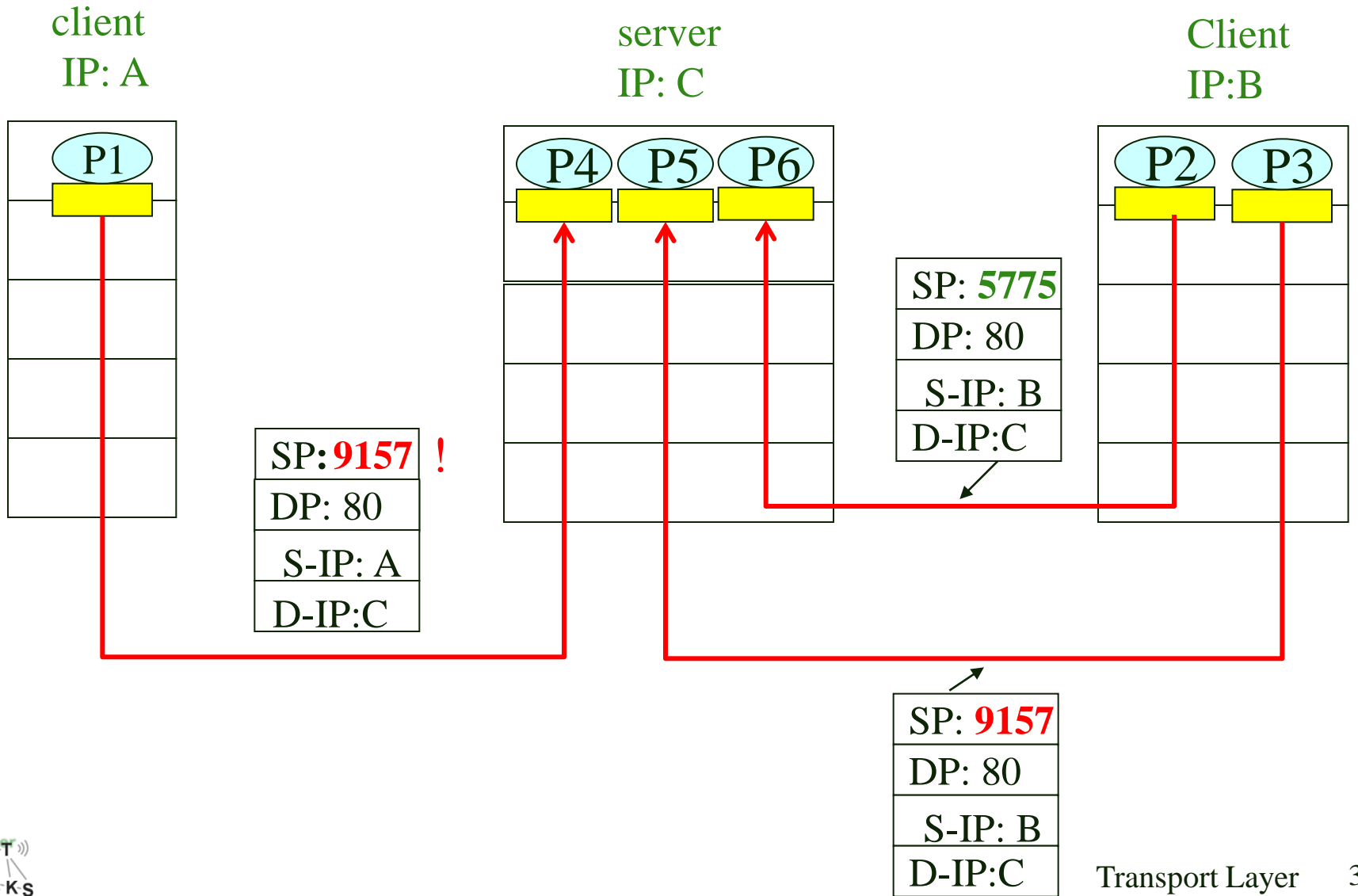
```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



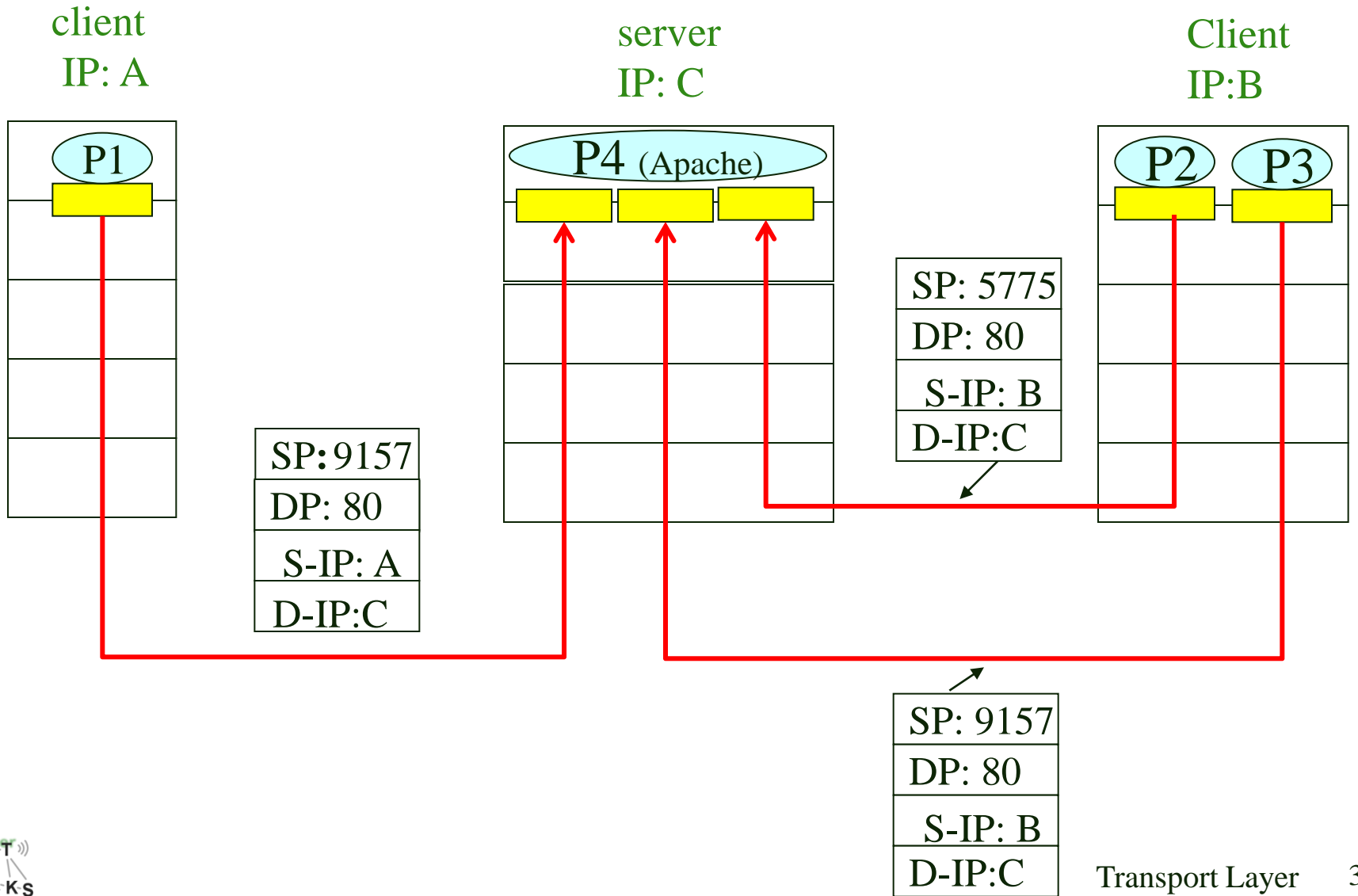
Connection-oriented demux

- 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)



Connection-oriented demux (cont)



Transport Layer

- 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

The Problem with TCP

- TCP offers a **reliable** and **easy** to use transport protocol to programmers.
 - Congestion control
 - Retransmissions etc.
- However congestion control imposes transmission-rate **constraints**.
- If a traffic jam is detected on a path, sender **decreases** sending rate “dramatically”.
- **Problem**: One cannot “switch” off functions of TCP ex. Congestion control.

UDP: User Datagram Protocol [RFC 768]

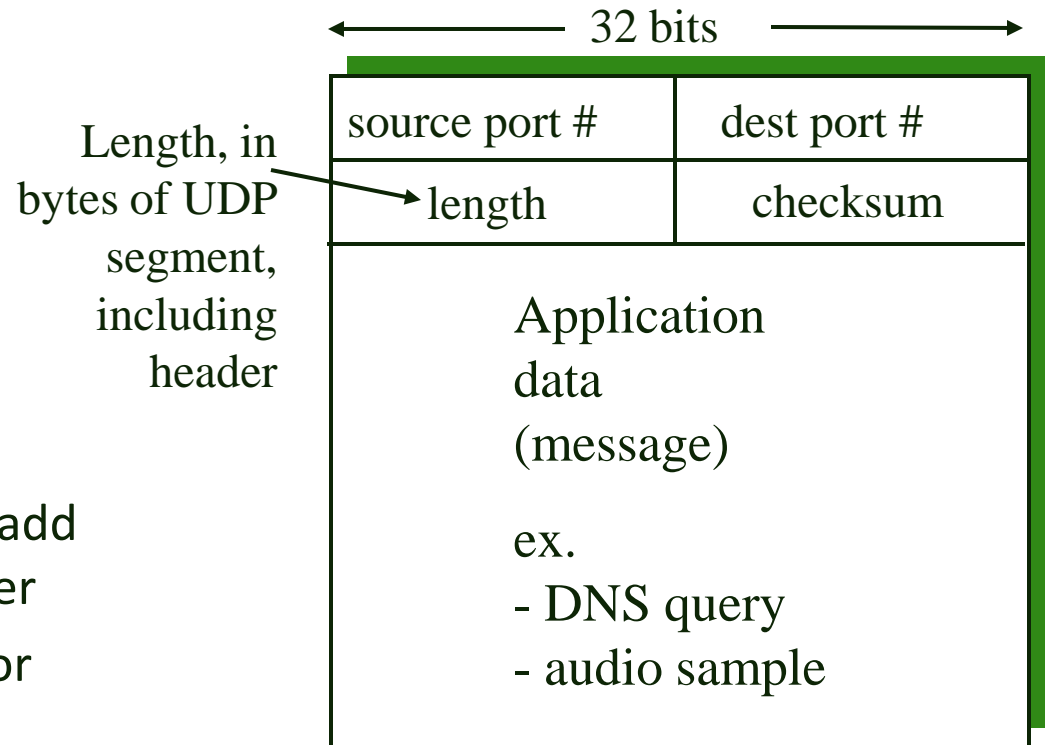
- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
 - lost
 - delivered out of order to app
- *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled **independently** of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state (buffers & parameters) at sender, receiver
- small segment header (8 bytes v.s. 20 bytes)
- no congestion control & retransmission: UDP can blast away as fast as desired (e.g. used by VOIP)

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!
 - ex. ACK/NAK, retransmissions (**non-trivial**).



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 (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?*
More later

UDP checksum example

- Lets take the word “hi” (8bit ASCII)
- Convert it to binary
 - h = 01101000
 - i = 01101001

- Add both words

$$\begin{array}{r} 01101000 \text{ (h)} \\ + 01101001 \text{ (i)} \\ \hline 11010001 \text{ (h+i)} \end{array}$$

- UDP checksum works with 16 Bit words, but we use 8 Bits for simplicity

- The 1s complement is obtained by inverting ones to zeros and vice versa.
- 11010001 -> 00101110 (checksum)

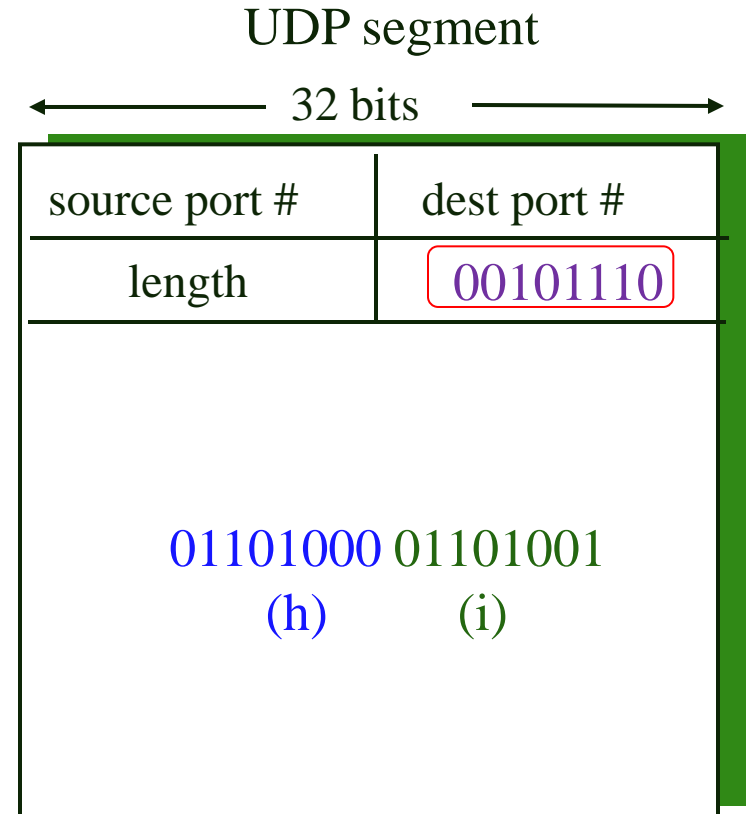
UDP checksum example

- Check (unaltered bits):

$$\begin{array}{r}
 01101000 \text{ (h)} \\
 + 01101001 \text{ (i)} \\
 \hline
 11010001 \text{ (h+i)} \\
 + 00101110 \text{ (checksum)} \\
 \hline
 11111111 \text{ (OK)}
 \end{array}$$

- Check (altered bits):

$$\begin{array}{r}
 01101000 \text{ (h)} \\
 + 011010\mathbf{11} \text{ (i)} \\
 \hline
 110100\mathbf{11} \text{ (h+i)} \\
 + 00101110 \text{ (checksum)} \\
 \hline
 \mathbf{10000001} \text{ (NOK!)}
 \end{array}$$



UDP checksum

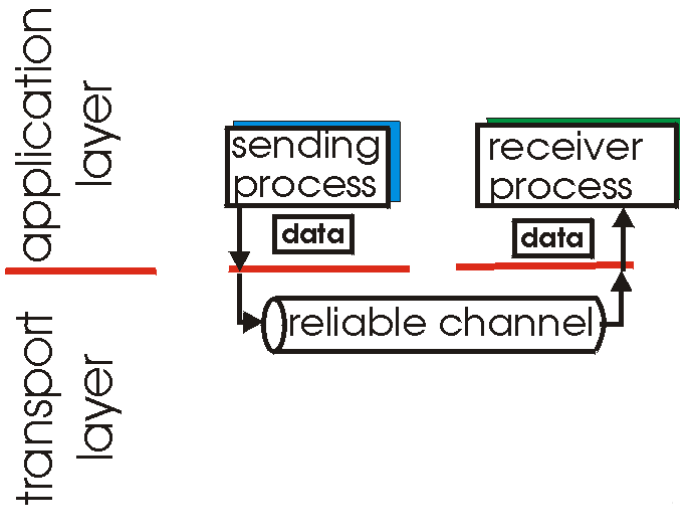
- Why error detection in the first place?
- Link Layer provides CRC! (Ethernet)
- No guarantee for:
 - link-to-link reliability (e.g. non ethernet)
 - memory error detection on routers
- IP is designed to run on any layer 2 protocol (ethernet, PPP, 802.11, 802.16).
- End-to-end error detection is safety measure
- UDP does not recover from errors (discard/warning)

Transport Layer

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

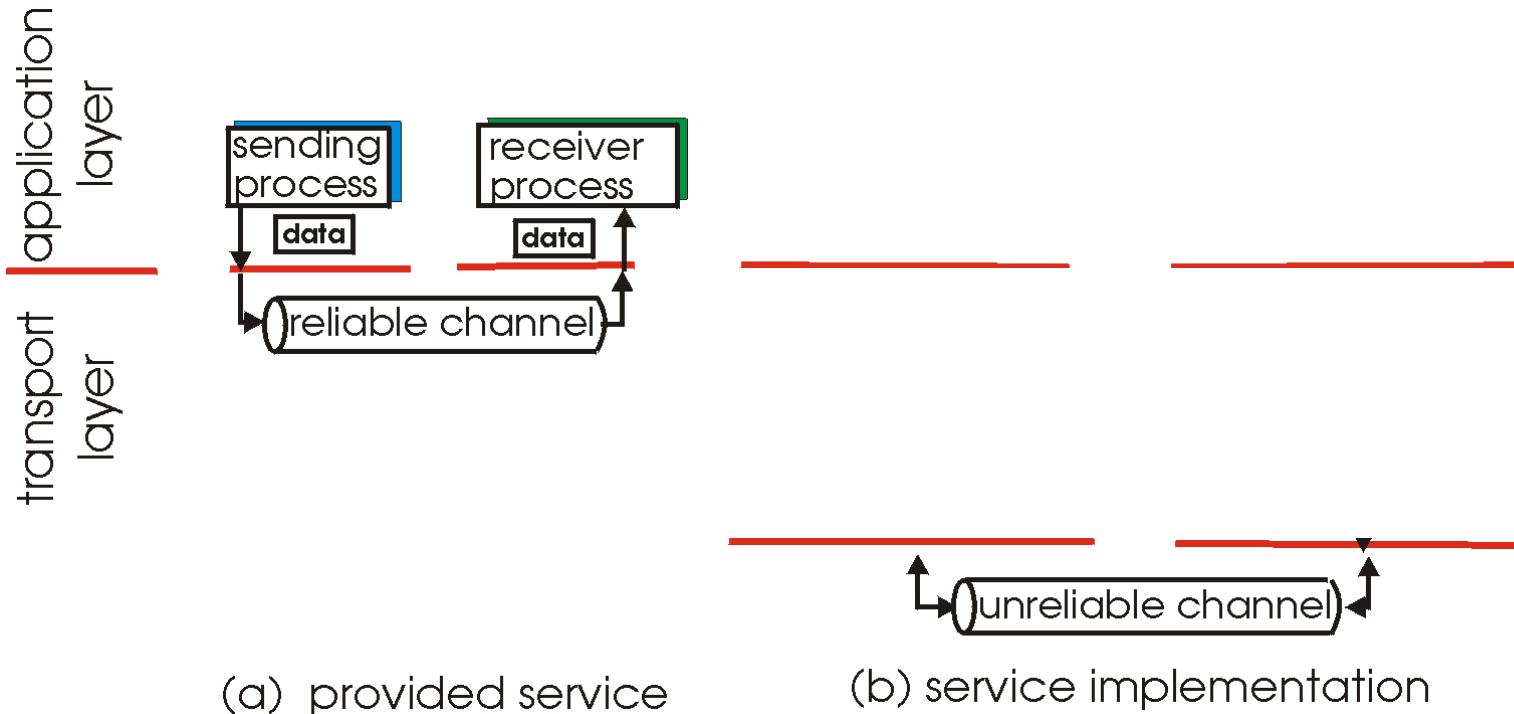


(a) provided service

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

Principles of Reliable data transfer

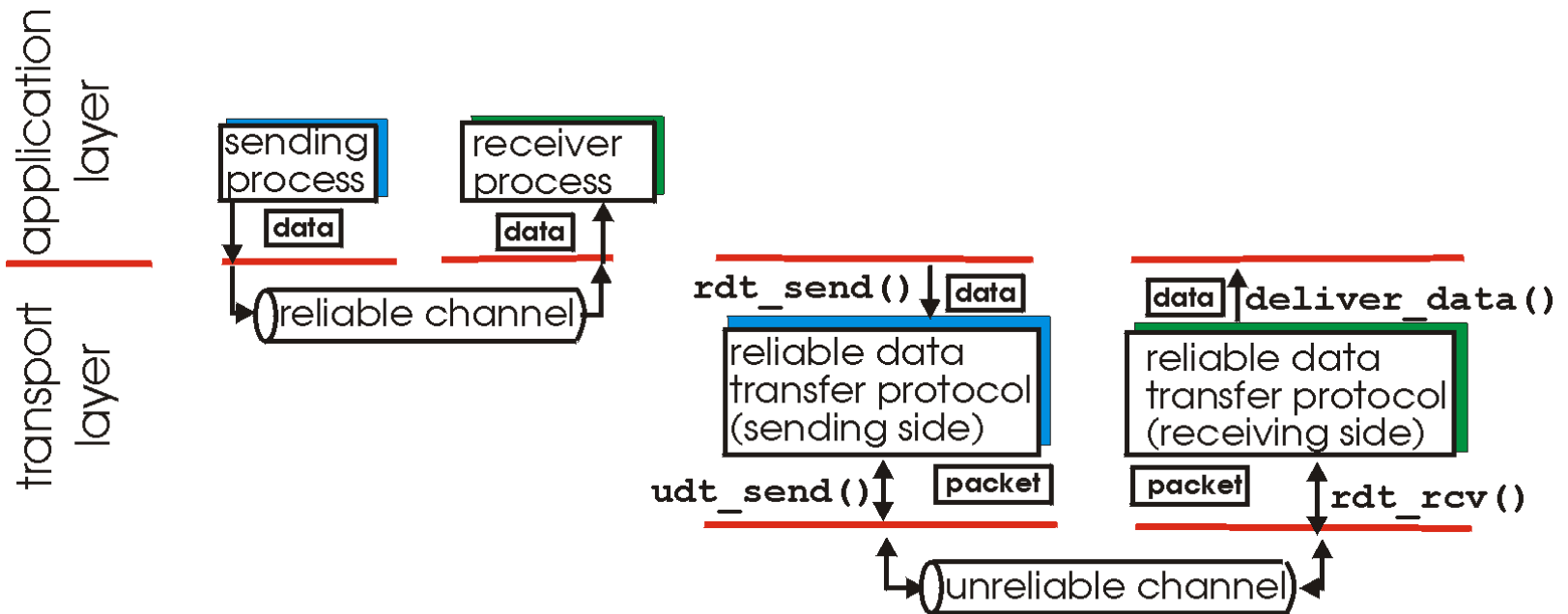
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

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

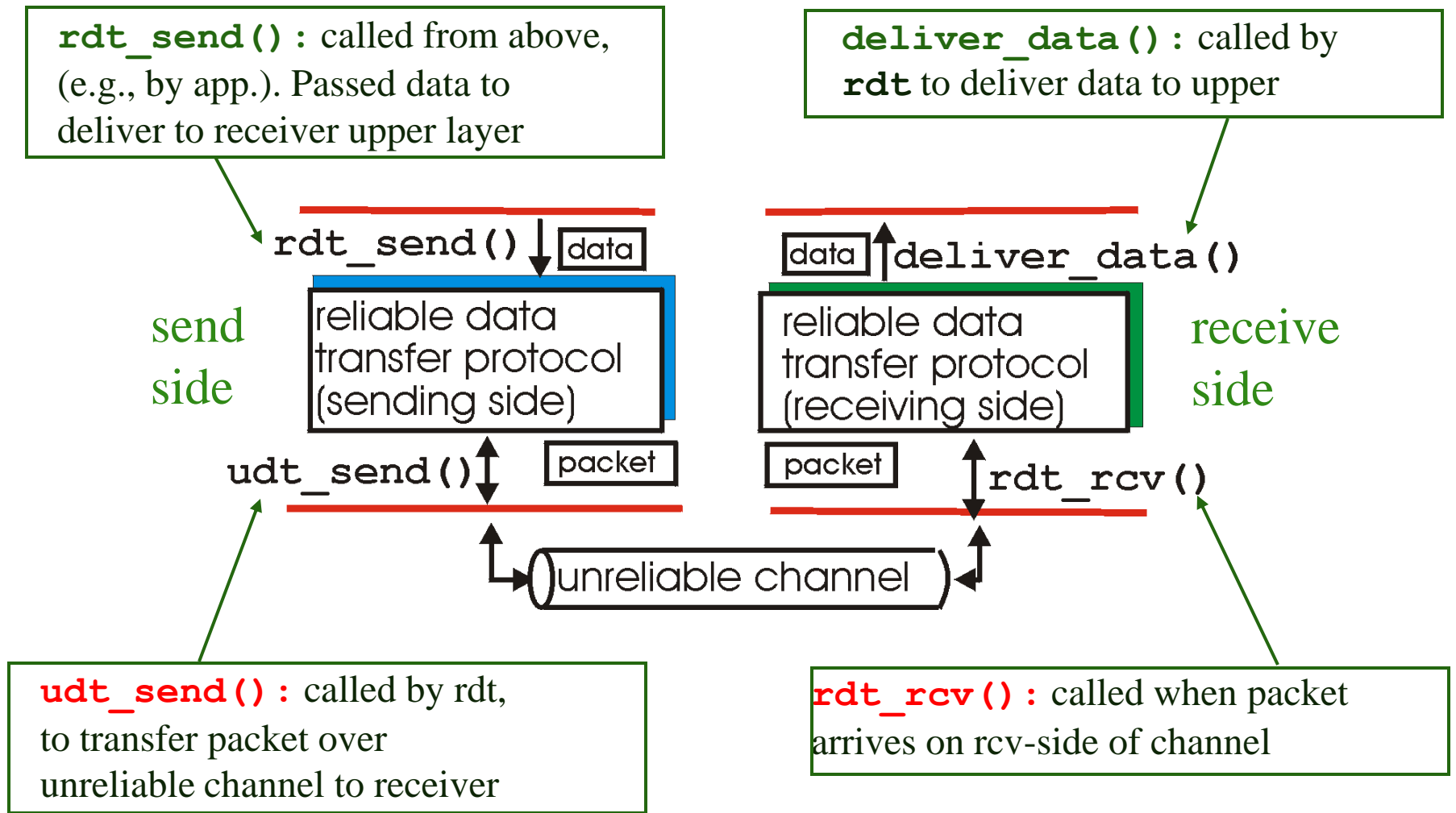


(a) provided service

(b) service implementation

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

Reliable data transfer: getting started



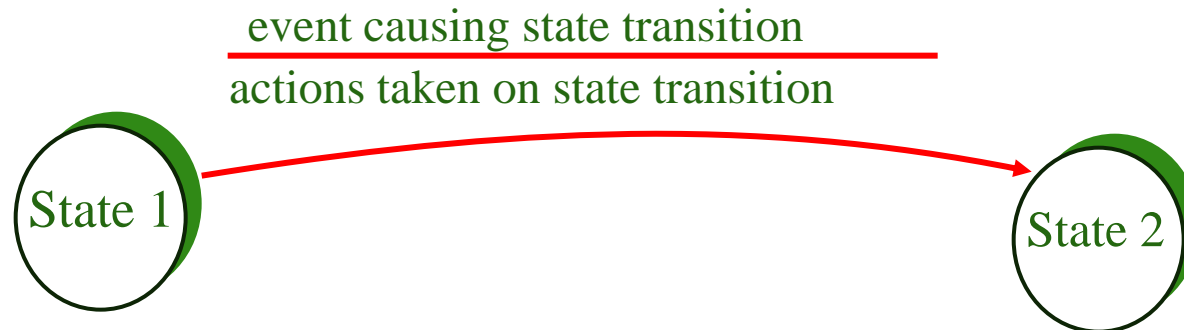
Reliable data transfer: getting started

We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
 - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver
- Use generic term “packet” rather than “segment”

Finite State Machine

- FSM is a model of behavior composed of a finite number of
 - states
 - transitions between states on events
 - actions taken upon events
- Necessary to define the behavior of our protocol, prior to implementation

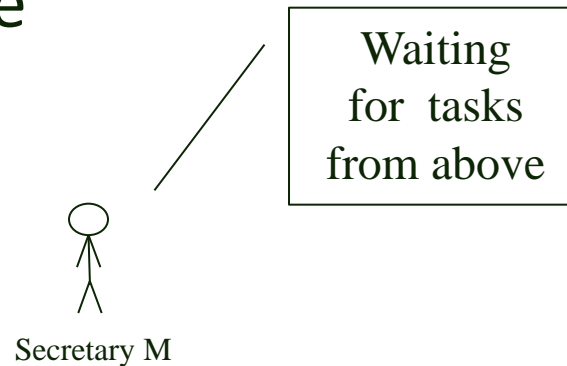


Rdt1.0: reliable transfer over a reliable channel

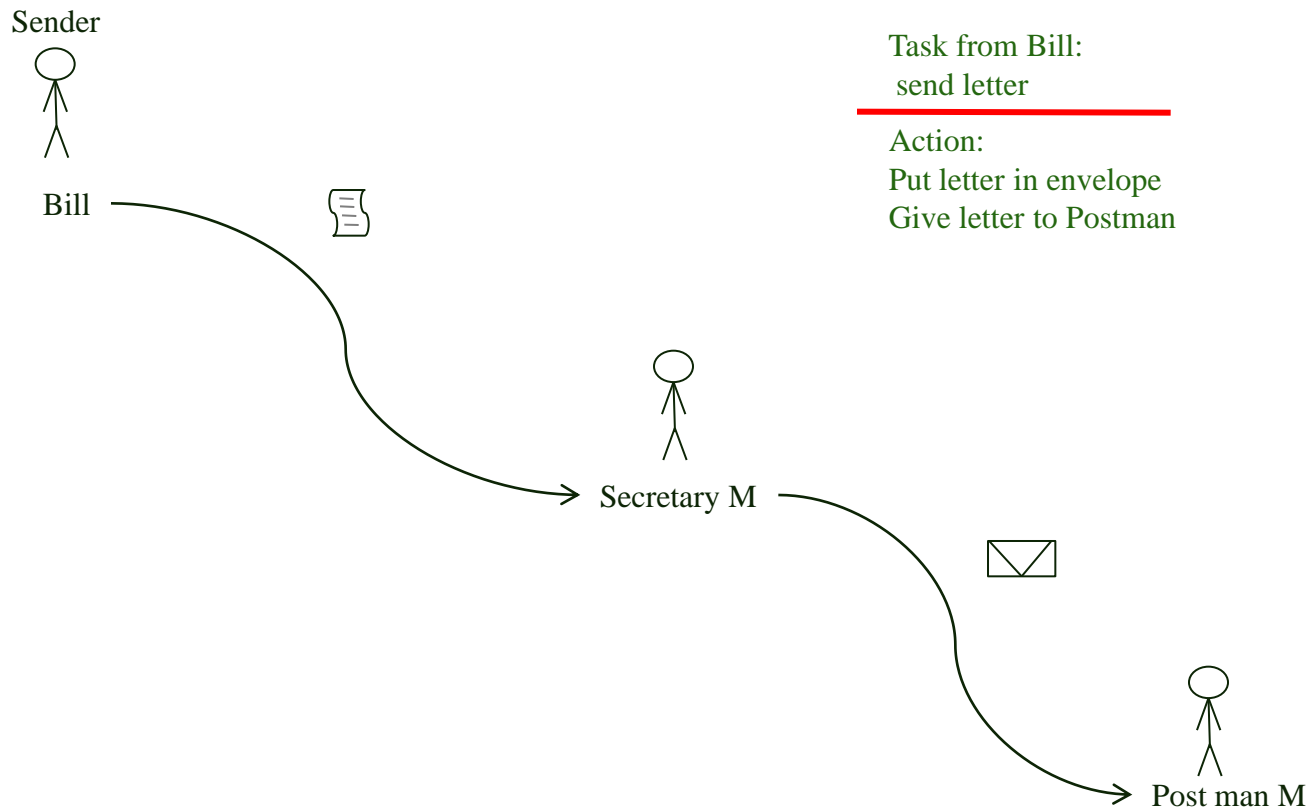
- **Assumption:** underlying channel perfectly reliable
 - no bit errors
 - no loss of packets
- separate FSMs for sender, receiver:
 - sender sends data into underlying channel
 - receiver read data from underlying channel
- We will first look at an analogy with the secretary then the state machines.

Rdt1.0: reliable transfer over a reliable channel (Analogy)

- The secretary from our previous example has one state
- He waits for tasks from his boss
- Task is sending letters



Rdt1.0: reliable transfer over a reliable channel (Analogy)

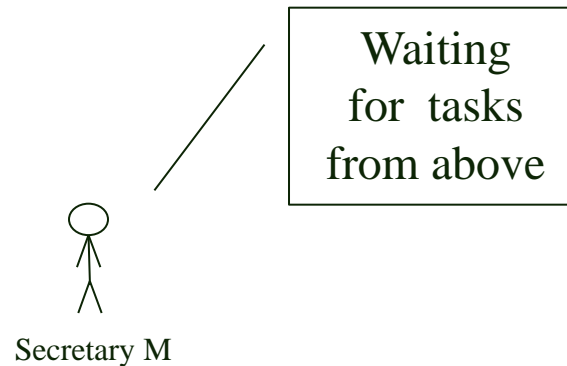


Task from Bill:
send letter
Action:
Put letter in envelope
Give letter to Postman

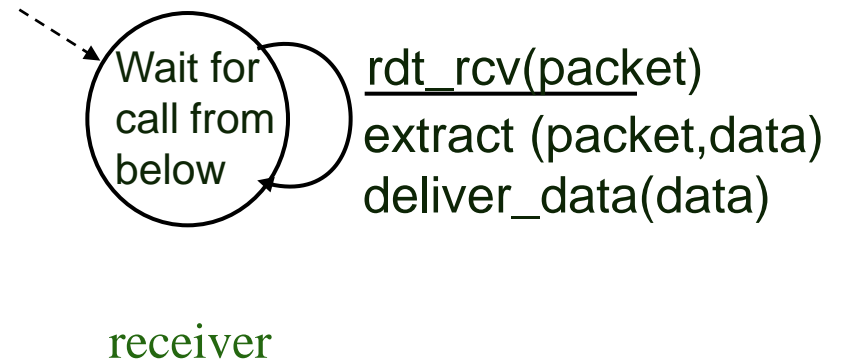
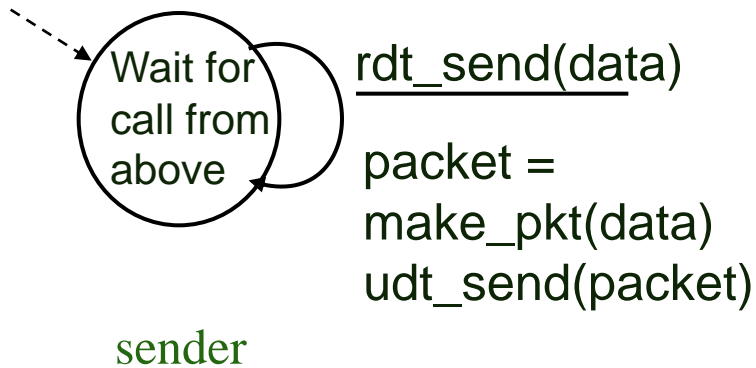
event → transition
transition → action

Rdt1.0: reliable transfer over a reliable channel (Analogy)

- The secretary goes back to his state, waiting for more tasks.



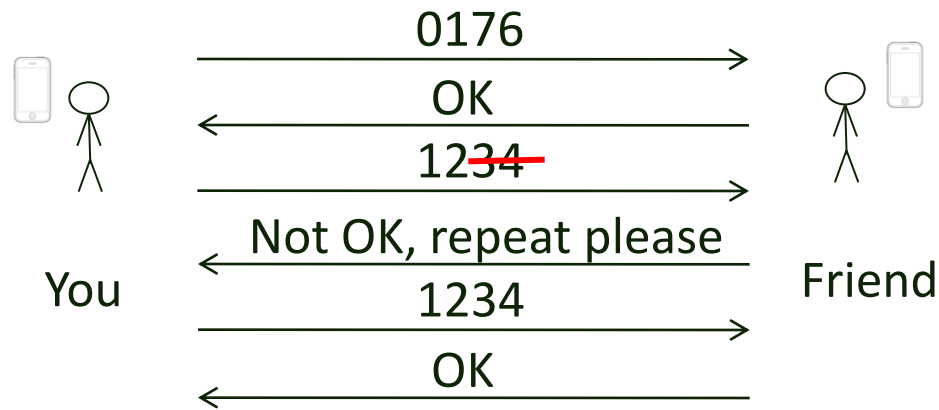
Rdt1.0: reliable transfer over a reliable channel



event → transition
transition → action
 Λ = no event/action

Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
 - checksum to detect bit errors 00101110
- **the question:** how to recover from errors?
- **Analogy:**
 - Imagine you dictate phone number over cell phone to friend.
 - Bad reception may scramble your voice.



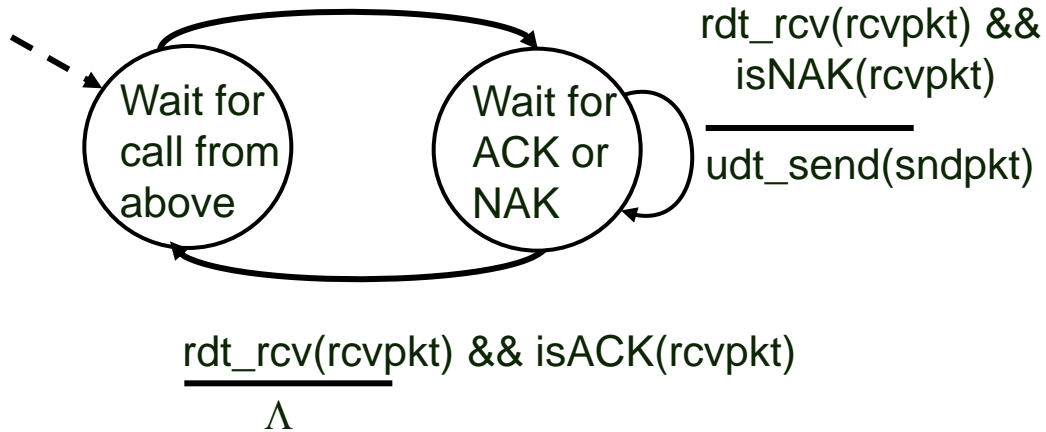
Rdt2.0: channel with bit errors

- *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
- **A**utomatic **R**epeat re**Q**uest type of protocol (**ARQ**)

rdt2.0: FSM specification

```

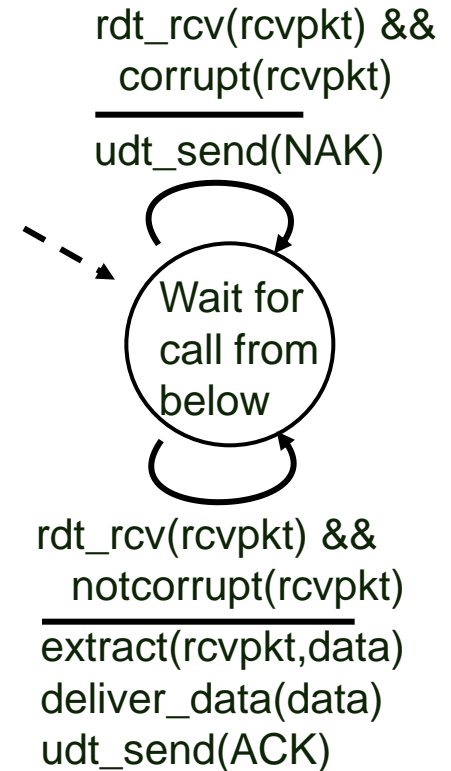
rdt_send(data)
snpkt = make_pkt(data, checksum)
udt_send(sndpkt)
    
```



sender

event → transition
 transition → action
 Λ = no event/action

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

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

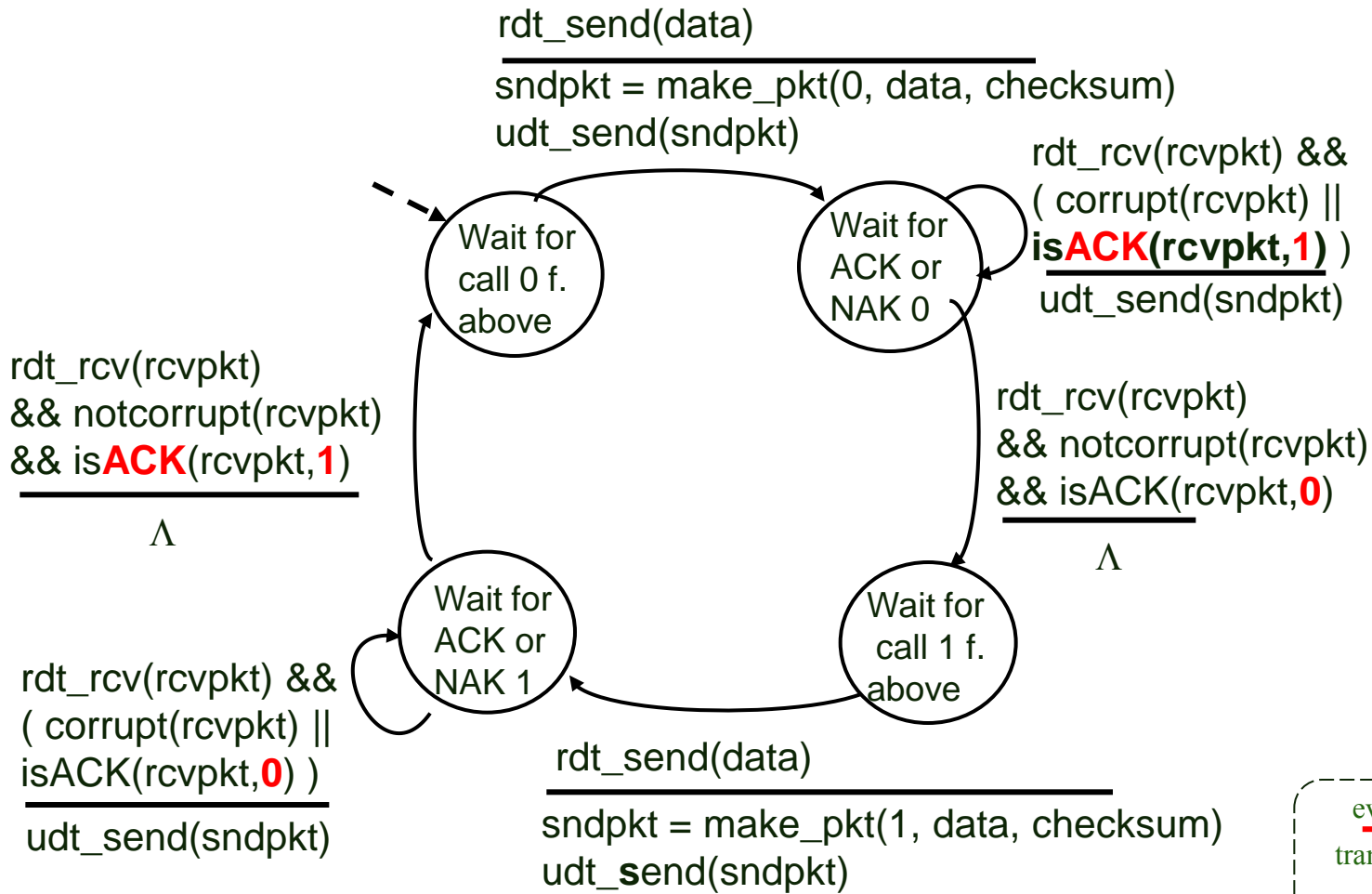
Using only ACK + Sequence:

- We can **discard** NAK packets, by using only ACK + Seq.#
- duplicate ACK at sender results in same action as NAK:
retransmit current pkt

stop and wait

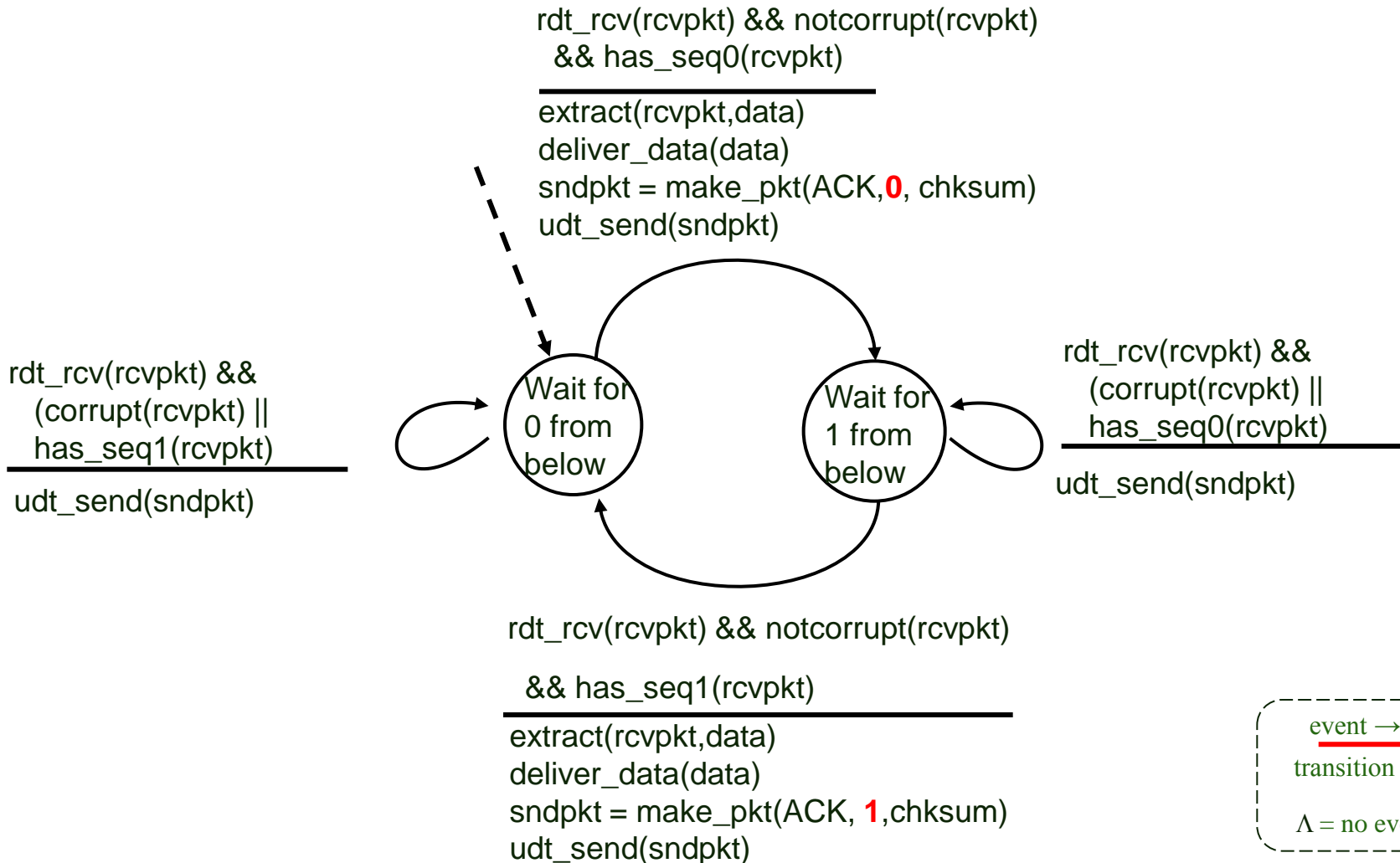
Sender sends one packet,
then waits for receiver
response

rdt2.2: sender, handles garbled ACKs



event \rightarrow transition
 transition \rightarrow action
 Λ = no event/action

rdt2.2: receiver, handles garbled ACKs



event → transition
transition → action
 Λ = no event/action

rdt2.2: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK 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 received OK at sender

rdt: What do we have so far?

- rdt 1.0

- simple transfer over reliable channel (unrealistic)



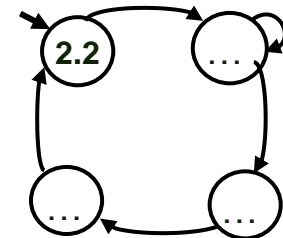
- rdt 2.0

- bit error prone channel (more realistic)
- checksum (data), ACK/NAK, retransmit
- **but what if ACK corrupt?**



- rdt 2.2

- checksum (data & ACK)
- retransmit if ACK corrupt
- **but what if data OK, but ACK corrupt? -> duplicate**
- introduce sequence numbers (more states)
- slimed down: discard NAK by introducing seq. in ACK
- **but what if channel loses packets?**



rdt3.0: channels with errors *and* loss

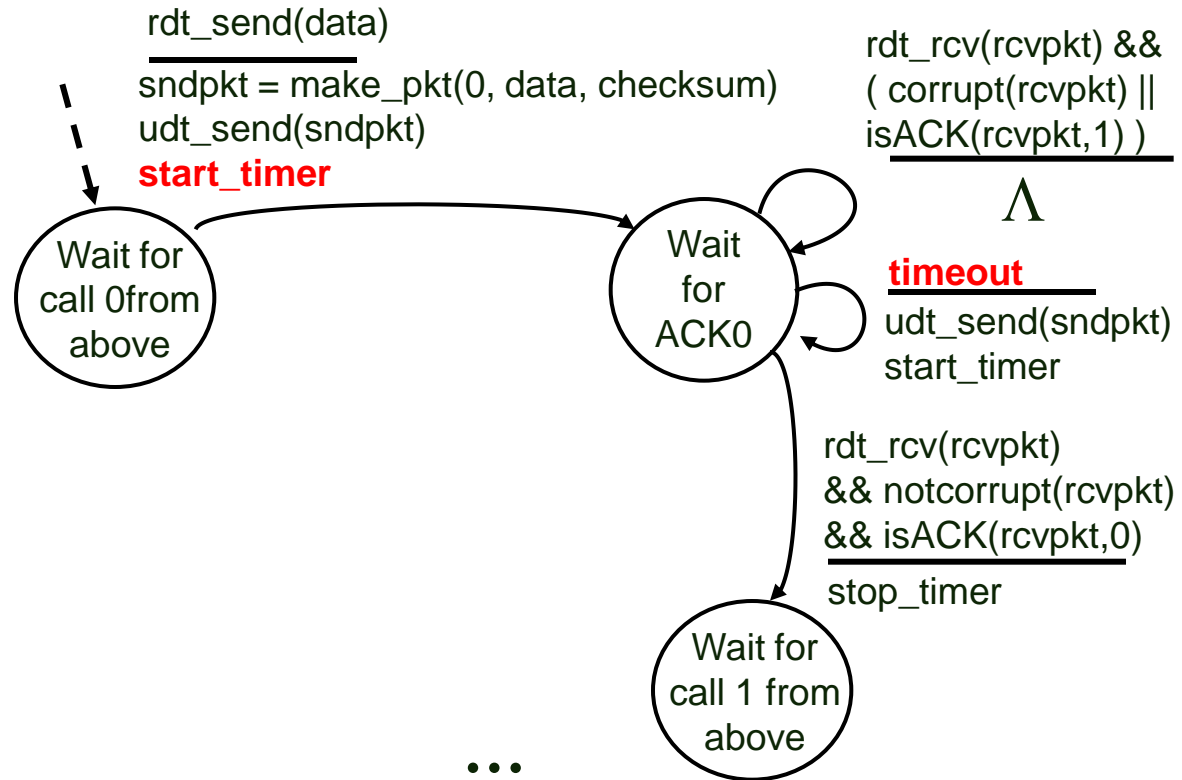
New assumption: underlying channel can also lose packets (data or ACKs)

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

Approach: sender waits “reasonable” amount of time for ACK

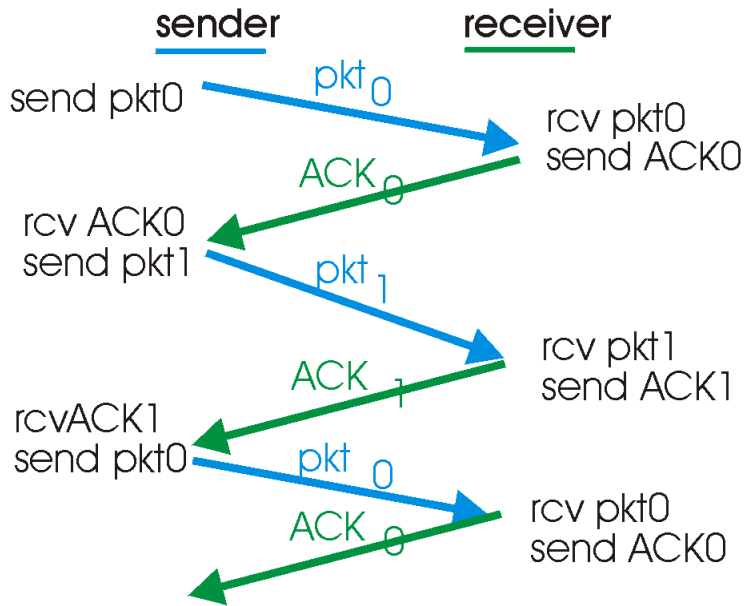
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but use of seq. #'s already handles this
 - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 sender

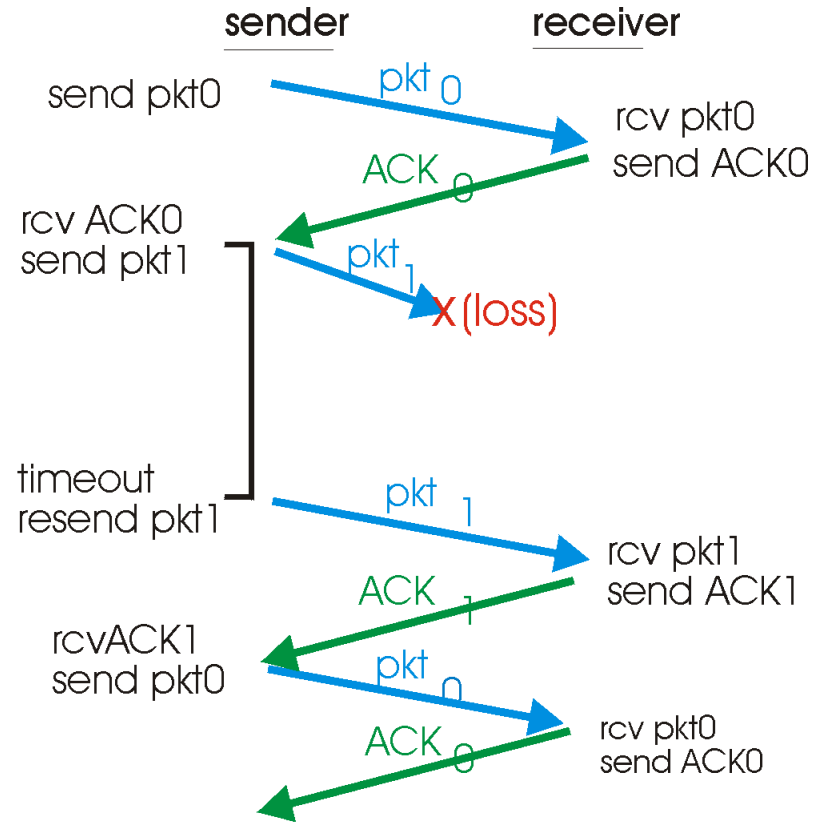


event → transition
 transition → action
 Λ = no event/action

rdt3.0 in action

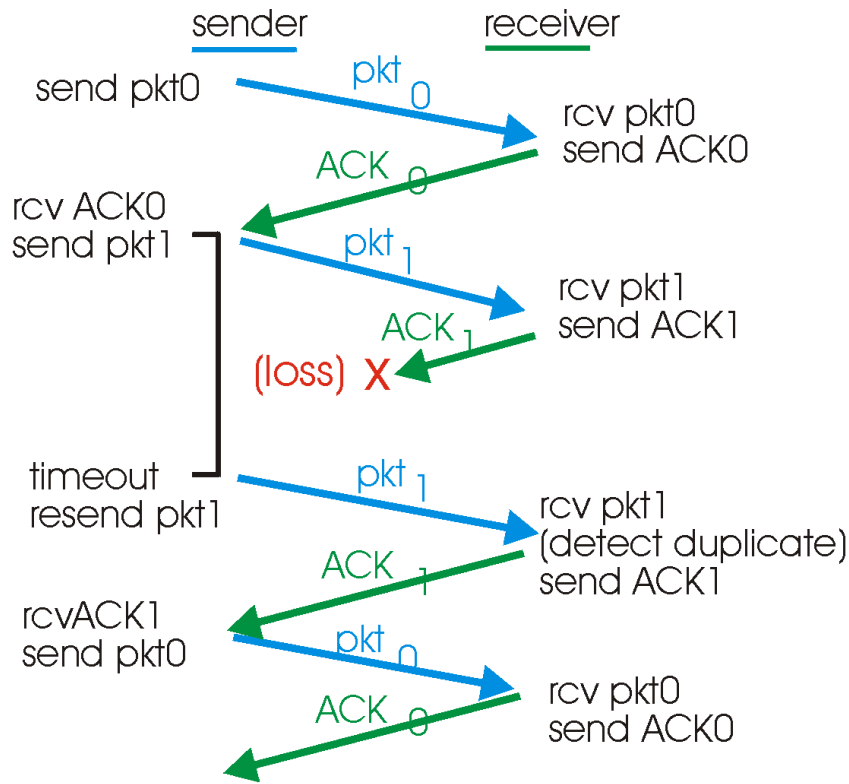


(a) operation with no loss

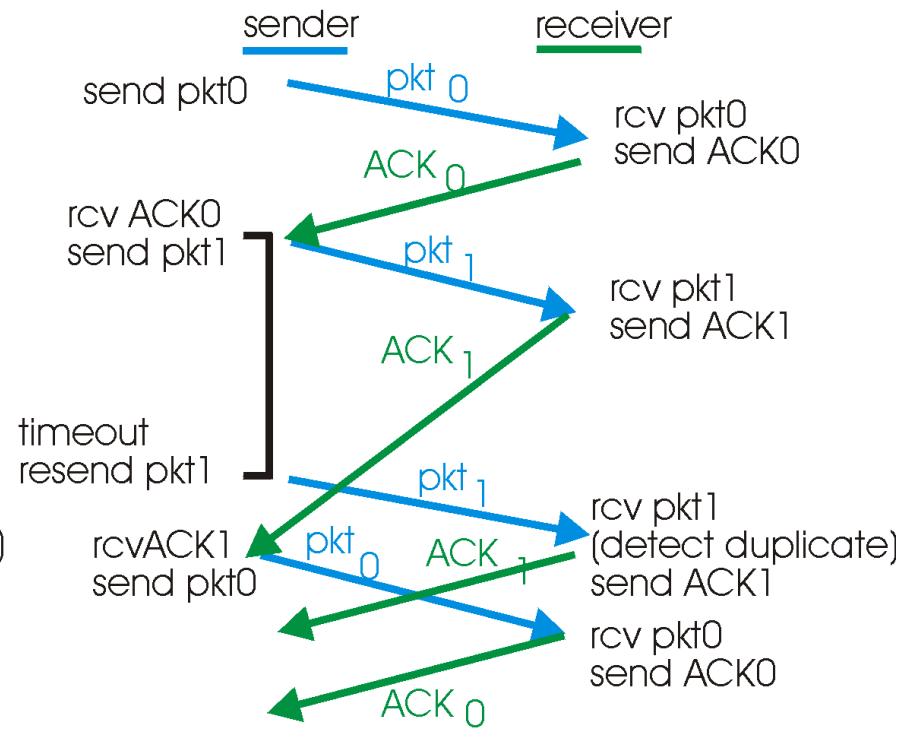


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

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

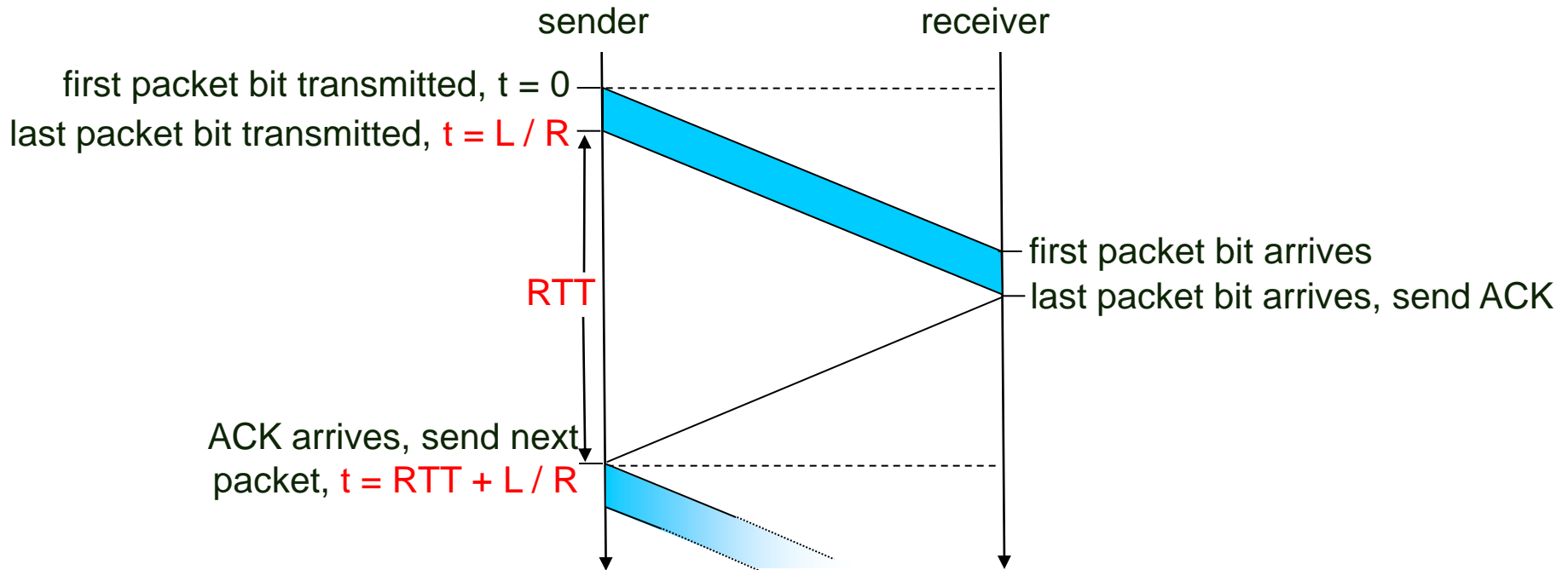
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

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

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

- 1KB pkt every 30 msec -> 33kB/sec thrupt over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

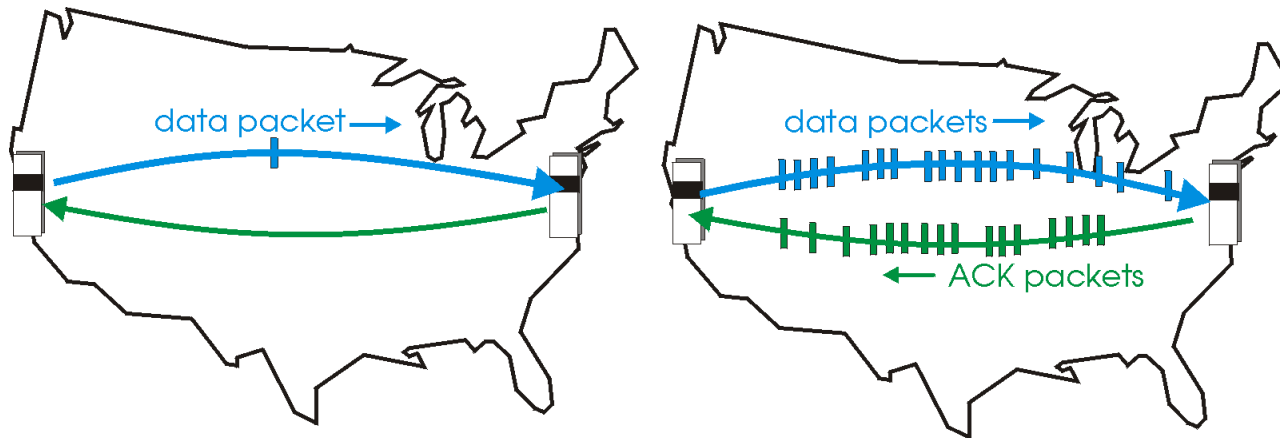


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

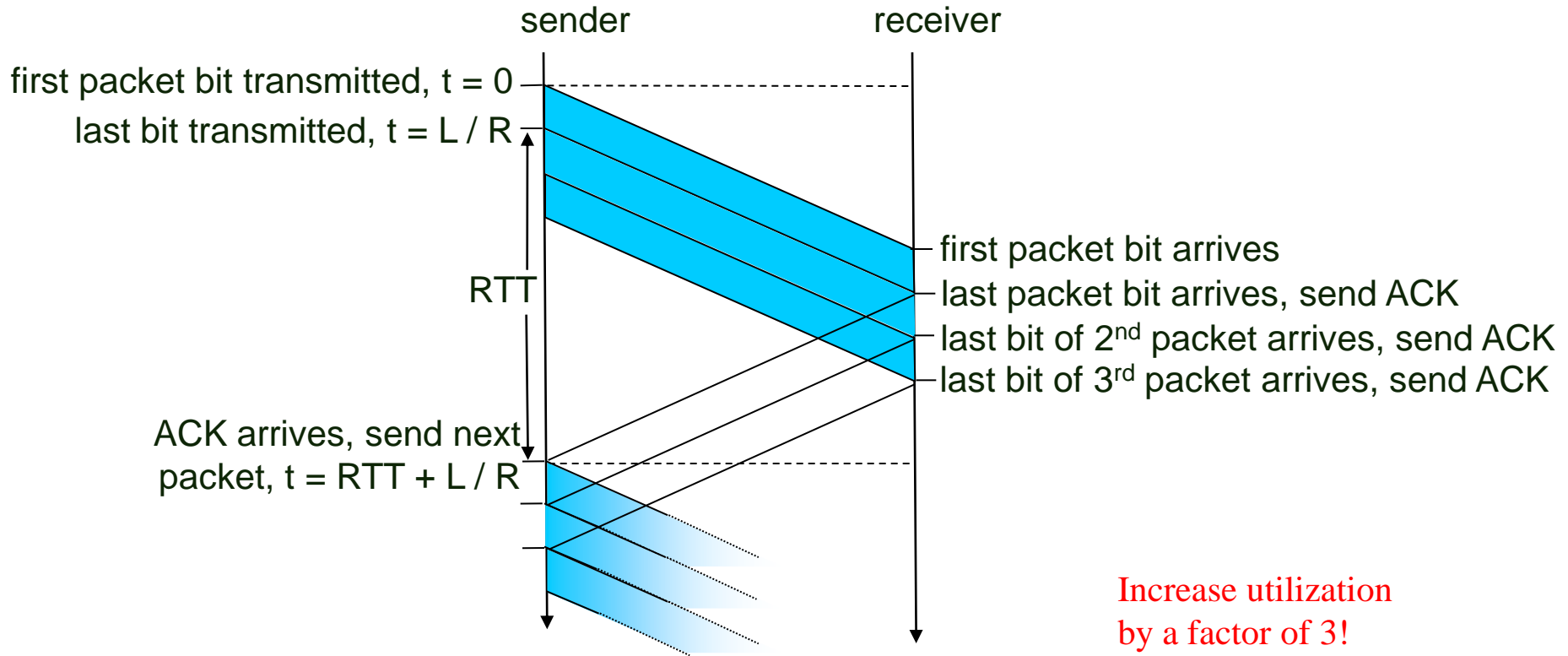


(a) a stop-and-wait protocol in operation

(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!

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

Pipelining 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 unacked packets

Selective Repeat: big pic

- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
 - When timer expires, retransmit only unack packet

Go-Back-N (GBN) Demonstration

- Protocol Demo ([Link](#))
- http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html

Transport Layer I: Summary

- principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer

Next:

- flow control
- congestion control
- instantiation and implementation in the Internet
 - UDP
 - TCP

Thank you

Any questions?