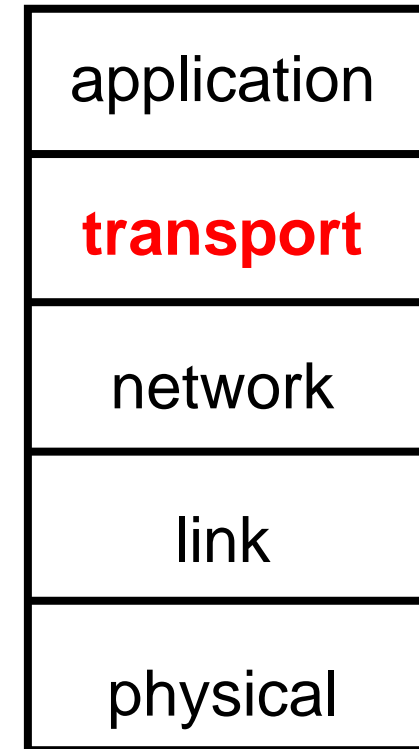


2. Transport layer and network layer

- * transport layer
 - transport-layer services
 - multiplexing and demultiplexing
 - connectionless transport: UDP
 - connection-oriented transport: TCP
- * network layer – data plane
 - overview of network layer
 - what's inside a router
 - Internet Protocol (IP)
 - generalized forwarding and Software-Defined Networking (SDN)

- *application*: supporting network applications
 - FTP, SMTP, HTTP
- *transport*: process-process data transfer
 - TCP, UDP
- *network*: routing of datagrams from source to destination
 - IP, routing protocols
- *link*: data transfer between neighboring network elements
 - Ethernet, 802.111 (WiFi), PPP
- *physical*: bits “on the wire”



**5-layer Internet
protocol stack**

- transport layer provides communication services directly to the application processes running on different end systems
- extends the network layer's delivery service between two end systems
- What are the principles? How to implement them in protocols?
- connectionless transport protocol - User Datagram Protocol (UDP)
- reliable communication - Transmission Control Protocol (TCP)

2.1 Transport-layer services

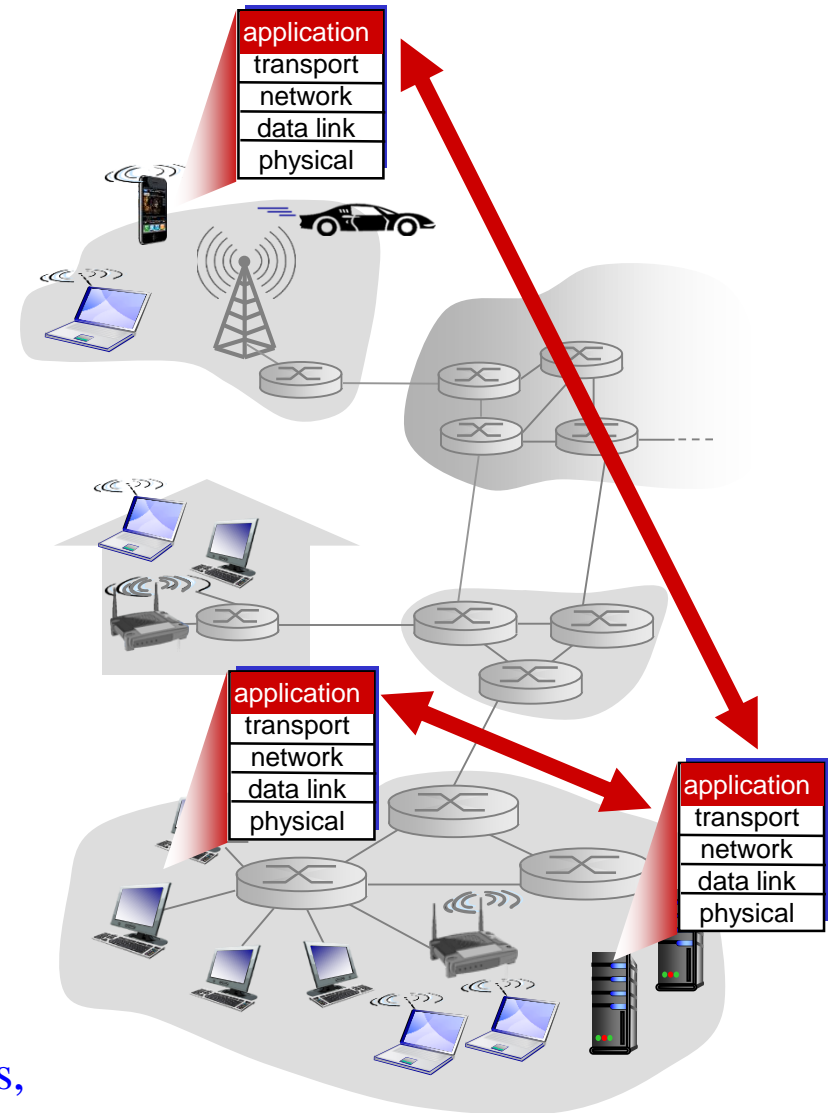
write programs that:

- ❖ run on (different) *end systems*
- ❖ communicate over network
- ❖ e.g., web server software communicates with browser software

no need to write software for
network-core devices

- ❖ network-core devices do not run user applications
- ❖ applications on end systems allows for rapid app development, propagation

Transport-layer protocols are implemented in the end systems,
not in the network routers



process: program running within a host

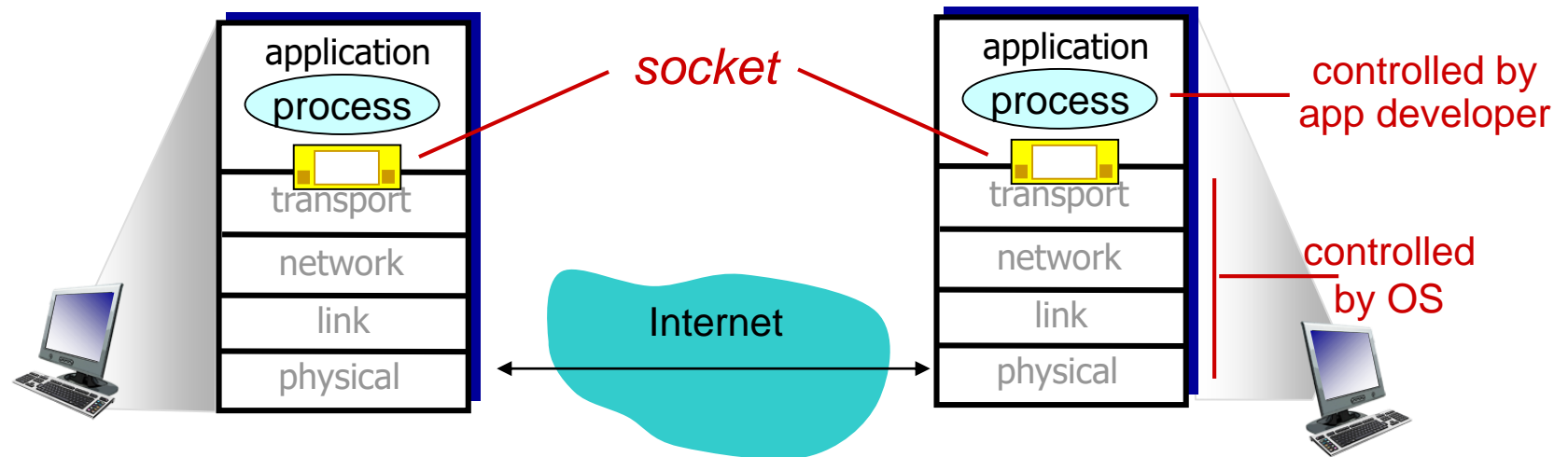
- ❖ within same host, two processes communicate using **inter-process communication** (defined by OS)
- ❖ processes in different hosts communicate by **exchanging messages**

clients, servers

client process: process that initiates communication

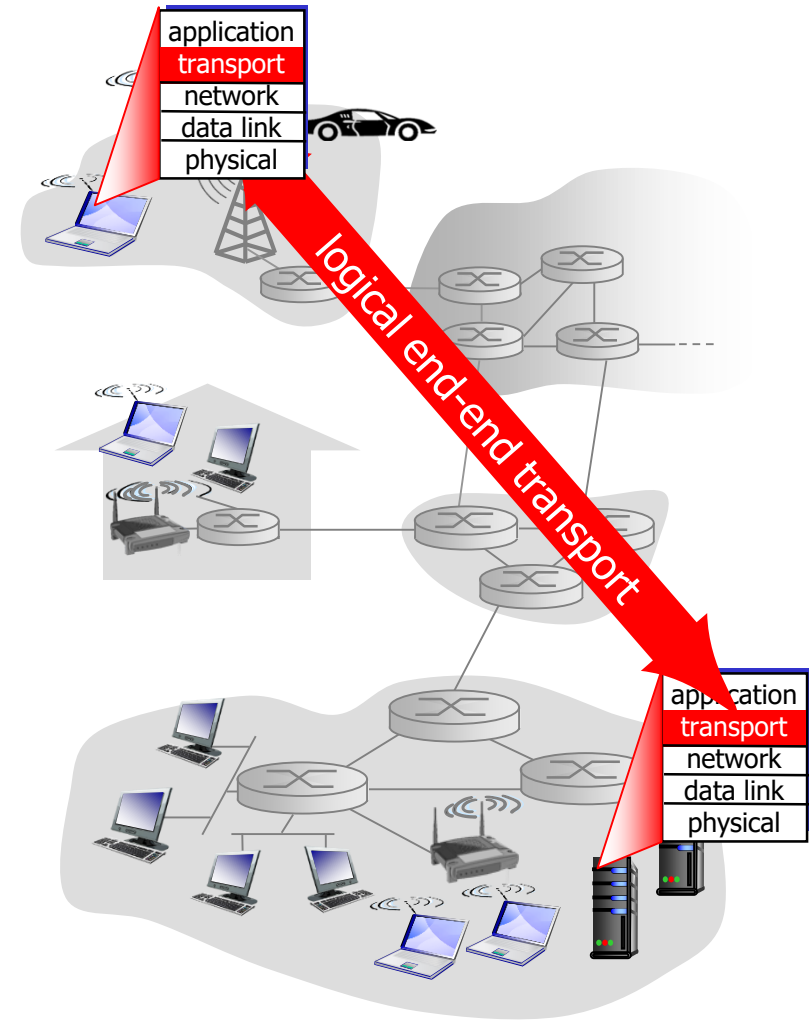
server process: process that waits to be contacted

- ❖ process sends/receives messages to/from its **socket**
- ❖ socket analogous to door
 - sending process shoves message out door
 - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



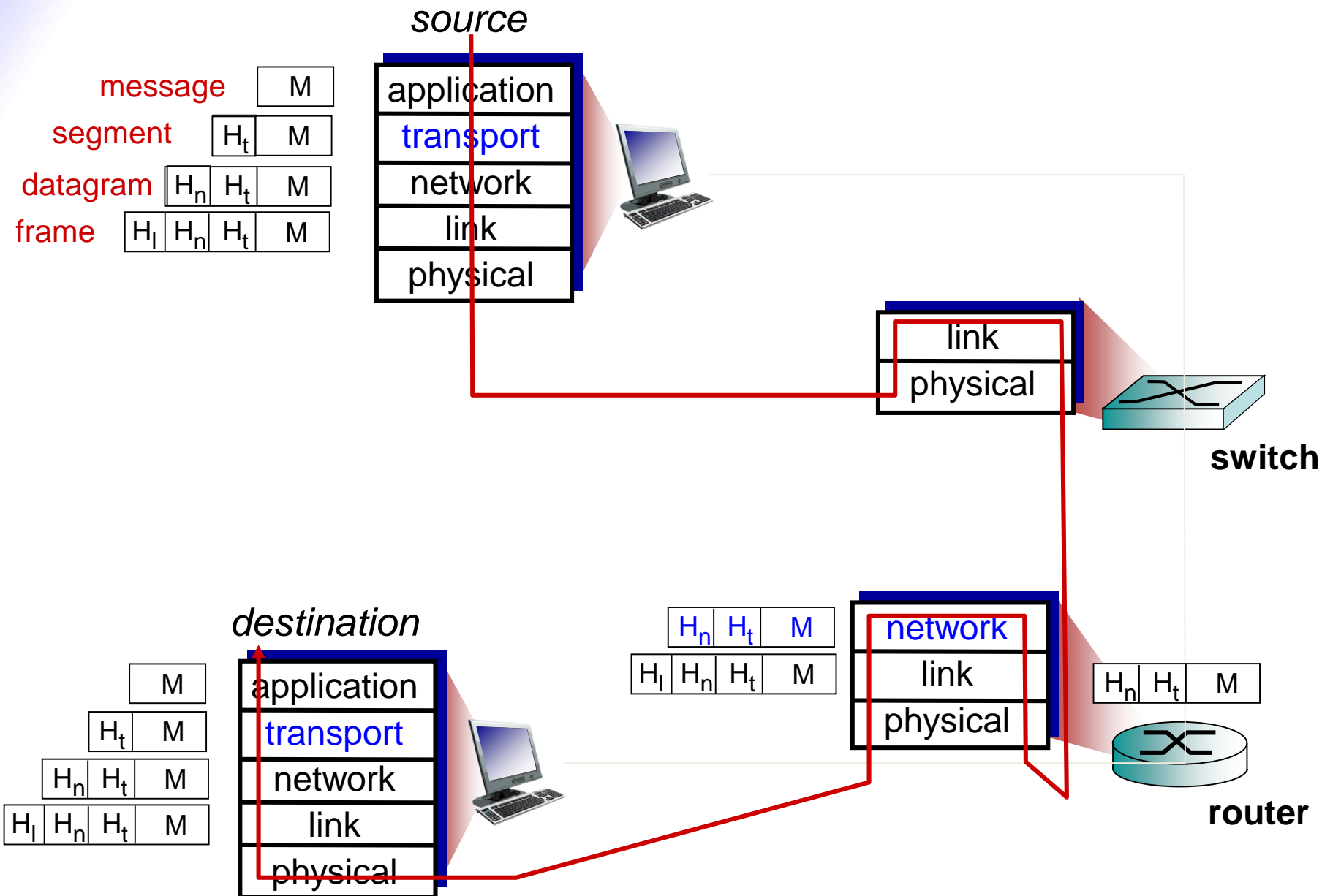
- ❖ to receive messages, process must have *identifier*
- ❖ host device has unique 32-bit IP address
- ❖ Q: does IP address of host on which process runs suffice for identifying the process?
 - A: no, *many* processes can be running on the same host
- ❖ *identifier* includes both **IP address** and **port numbers** associated with process on host.
- ❖ example port numbers:
 - HTTP server: 80
 - mail server: 25
- ❖ to send HTTP message to gaia.cs.umass.edu web server:
 - **IP address**: 128.119.245.12
 - **port number**: 80

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



- **sender** – application process **sends messages to transport layer**
- **transport layer converts messages into transport-layer packets** (segments) with addition of transport-layer header
- **transport layer passes segment to network layer**
- **network layer converts segment into network-layer packet (datagram)** with the addition of network-layer field
- **datagram is sent to the destination**

- network routers act only on network-layer field of the datagram
- receiver – network layer extracts the transport-layer segment from datagram, passes the segment up to the transport layer
- transport layer processes the received segment, passes the data/messages to the receiving application process



Transport vs. network layer

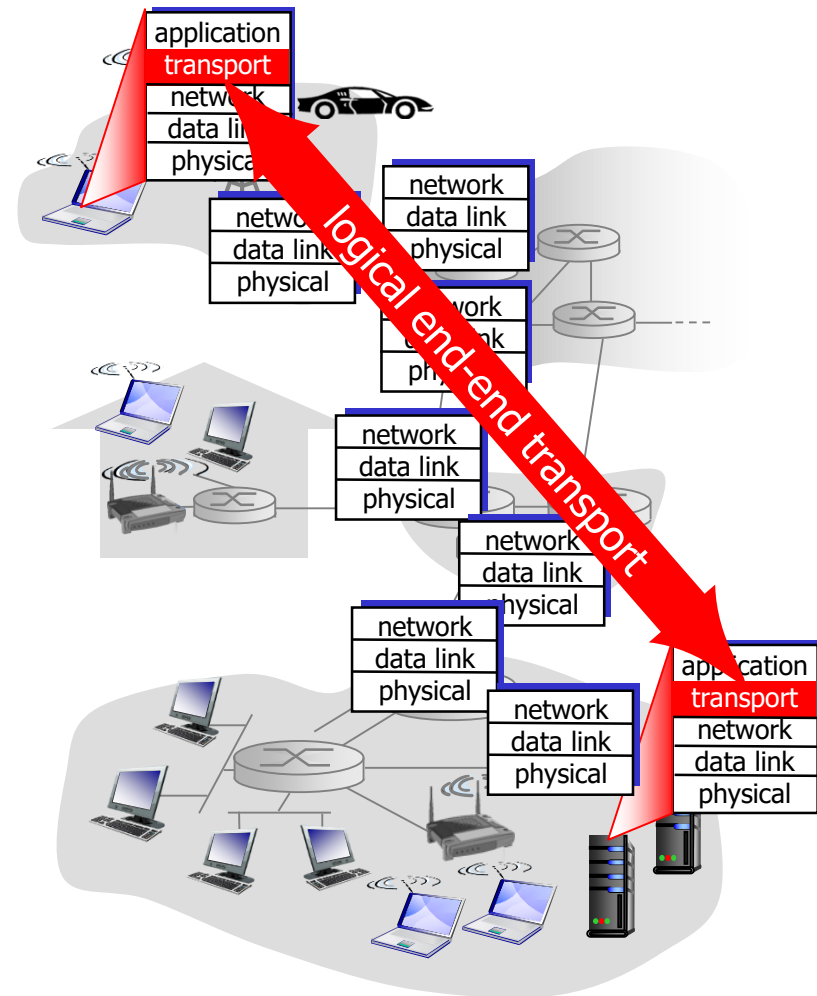
- ❖ *network layer*:
logical communication between **hosts**
- ❖ *transport layer*:
logical communication between **processes**
 - relies on, enhances, network layer services

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- ❖ hosts = houses
- ❖ processes = kids
- ❖ app messages = letters in envelopes
- ❖ transport protocol = Ann and Bill who demux to in-house siblings
- ❖ network-layer protocol = postal service

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



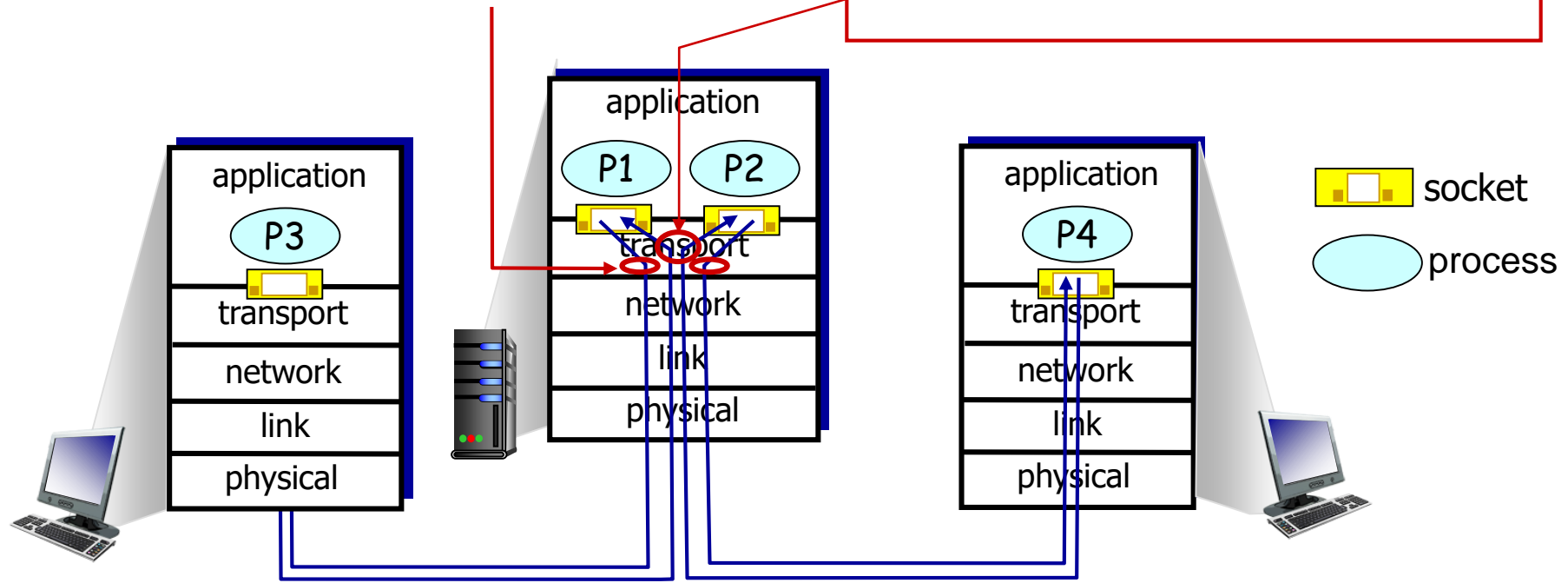
2.2 Multiplexing and demultiplexing

multiplexing at sender:

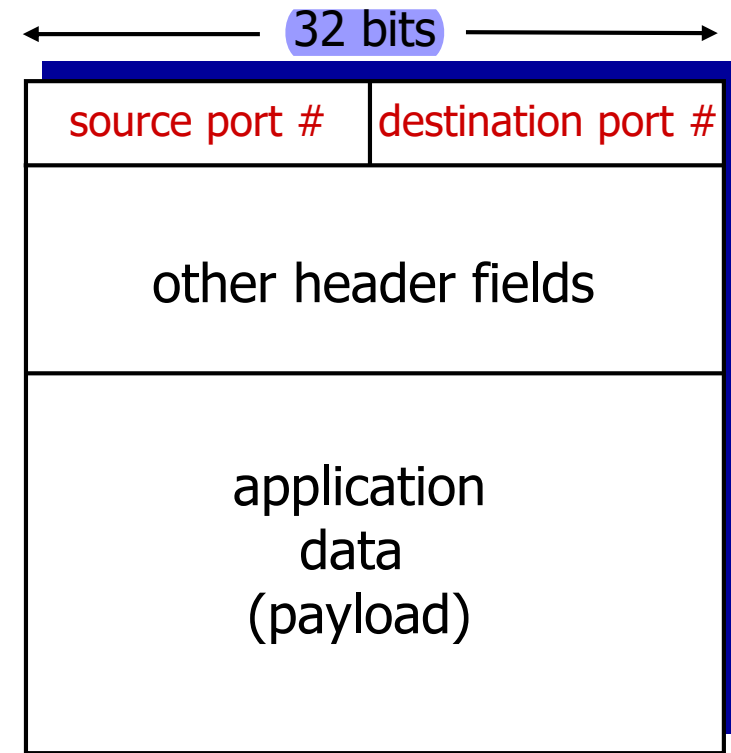
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

use header info to deliver received segments to correct socket



- ❖ host receives IP datagrams
 - each datagram has source IP address, destination IP address
 - each datagram carries one 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

Destination port number is 16-bit (0 – 65535). 0-1023 are well-known port numbers (restricted). For instance, HTTP port number is 80.

- ❖ *note*: created socket has host-local port #:

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

- ❖ *note*: when creating datagram to send into UDP socket, must specify

- destination IP address
 - destination port #
-

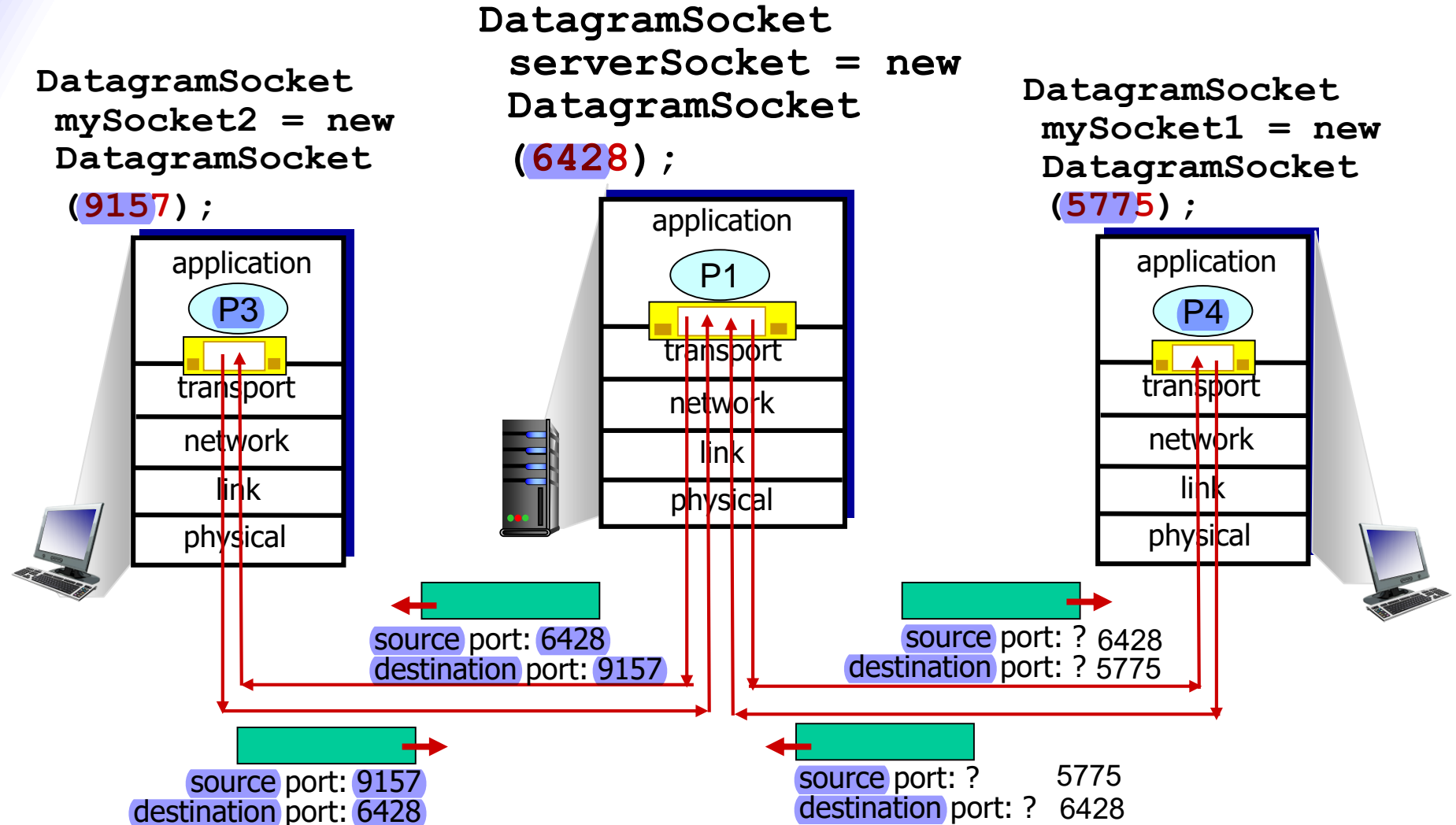
- ❖ when host receives UDP segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #



IP datagrams with *same destination port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at destination

Example

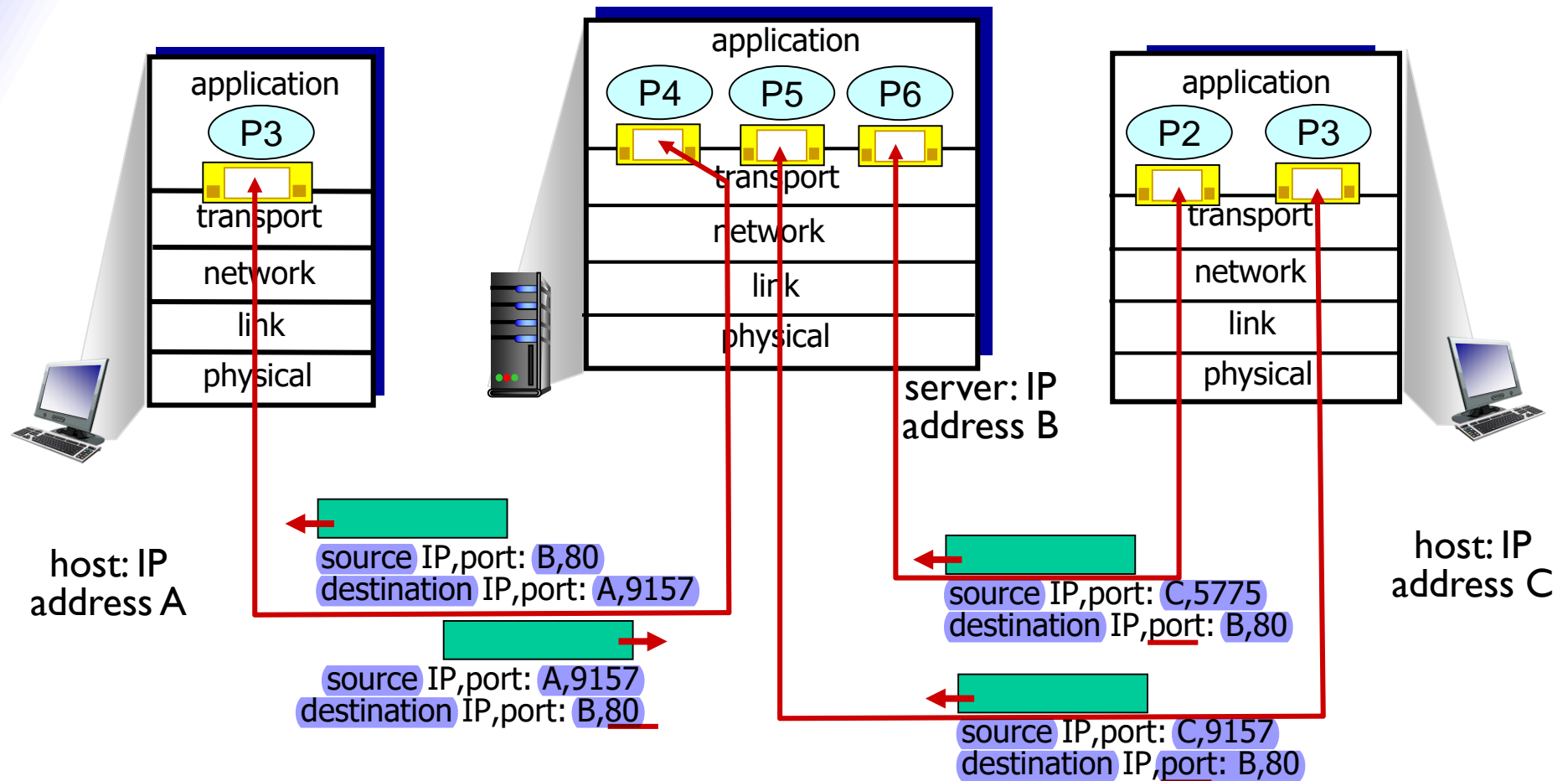


To reverse the communication direction, just reverse the source and destination port numbers!

- ❖ TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - destination IP address
 - destination port number
- ❖ demultiplexing:
receiver uses all four values to direct segment to appropriate socket

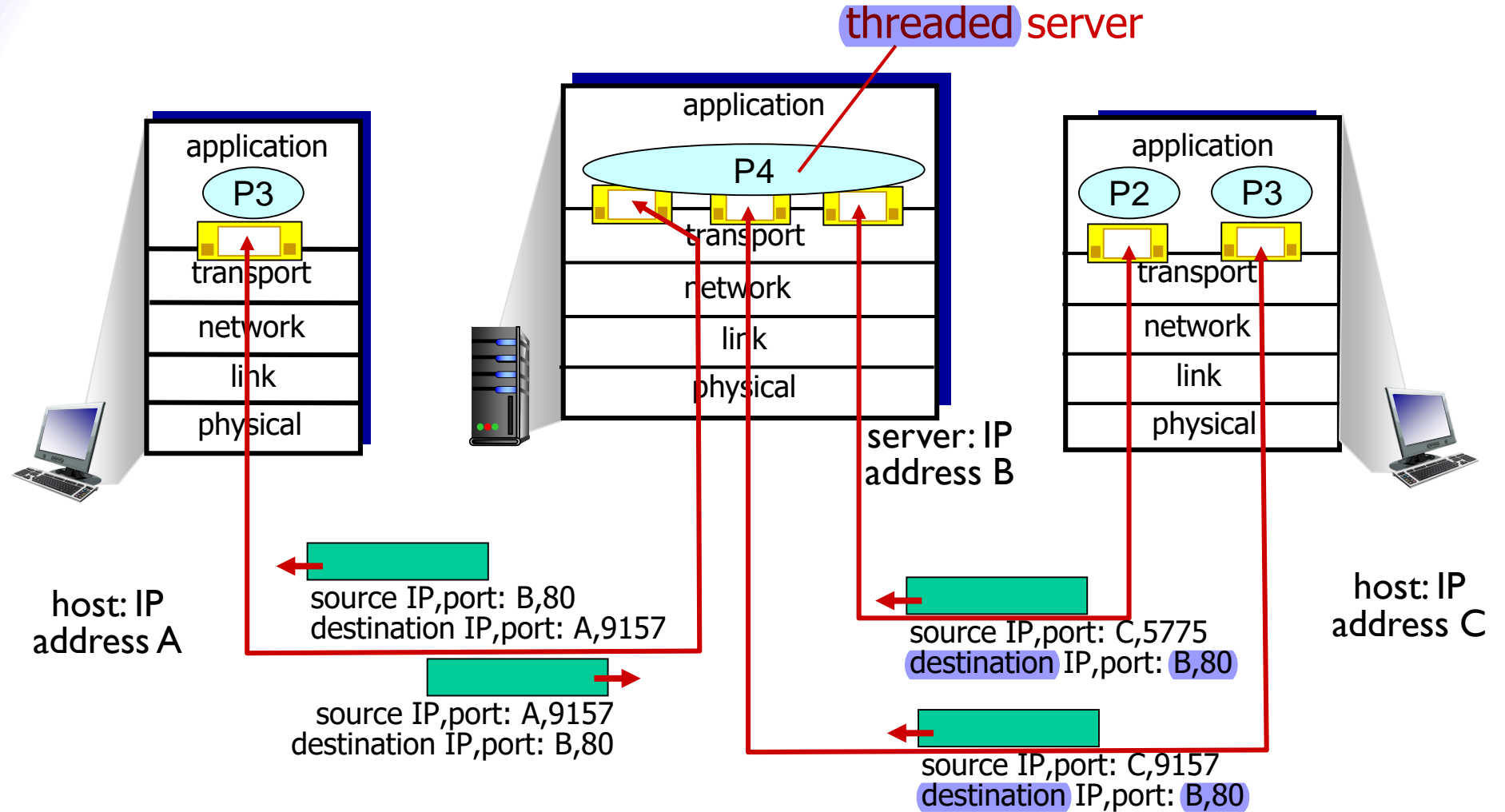
- ❖ server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
- ❖ web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

Example



three segments, all destined to IP address: B,
destination port: 80 are demultiplexed to *different* sockets

Example



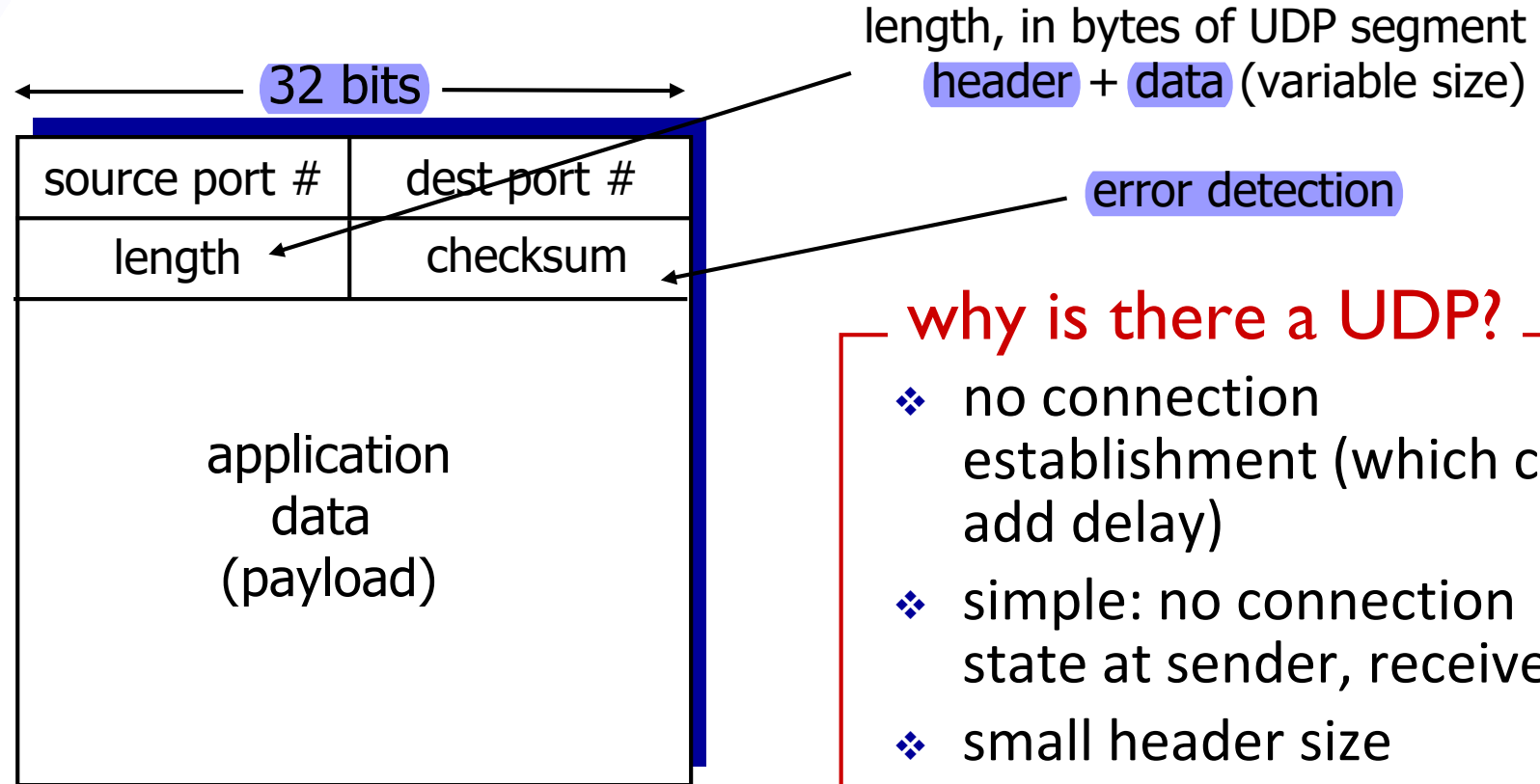
Only 1 process. A new thread (sub-process) is created with a new socket for each new client connection.

2.3 Connectionless transport: UDP

- ❖ “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 use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - Domain Name System (DNS)
 - Simple Network Management Protocol (SNMP)
- ❖ reliable transfer over UDP:
 - add reliability at application layer
 - application-specific error recovery!

Why build an application over UDP rather than over TCP?

- Real-time applications require minimum delay, can tolerate some data loss
- Can support more active clients
- Small packet header overhead – TCP (20 bytes), UDP (8 bytes)



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

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.

Note: detail of checksum is discussed in link layer

- the problem of implementing reliable data transfer occurs not only at the transport layer
- it is also a fundamentally important problem at the application layer, and link layer as well
- we may have a reliable data transfer protocol at the transport layer
- but the lower network layer is not reliable!

Application layer



Transport layer



Network layer



- to provide reliable data transfer, TCP relies on:

error detection

re-transmissions

cumulative acknowledgments

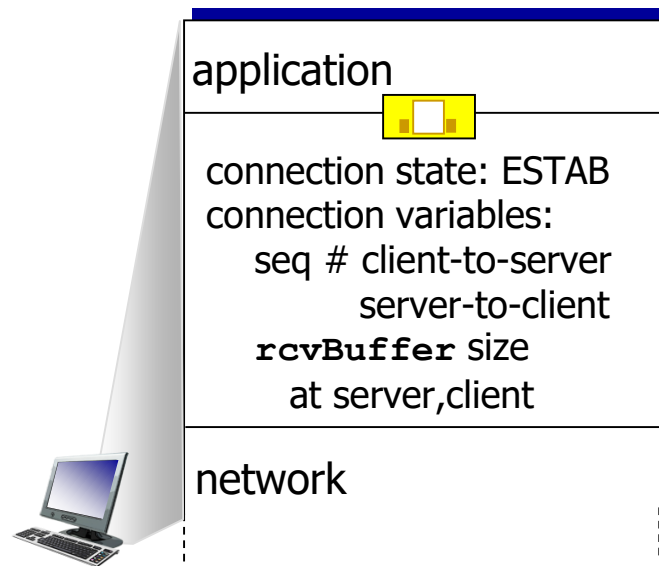
timers

header fields for sequence and acknowledgment numbers

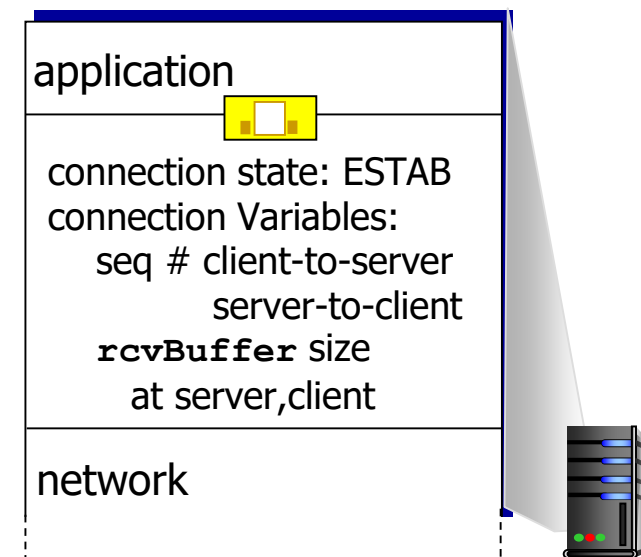
2.4 Connection-oriented transport: TCP

before exchanging data, sender/receiver “handshake”:

- ❖ agree to **establish connection** (each knowing the other willing to establish connection)
- ❖ agree on connection parameters

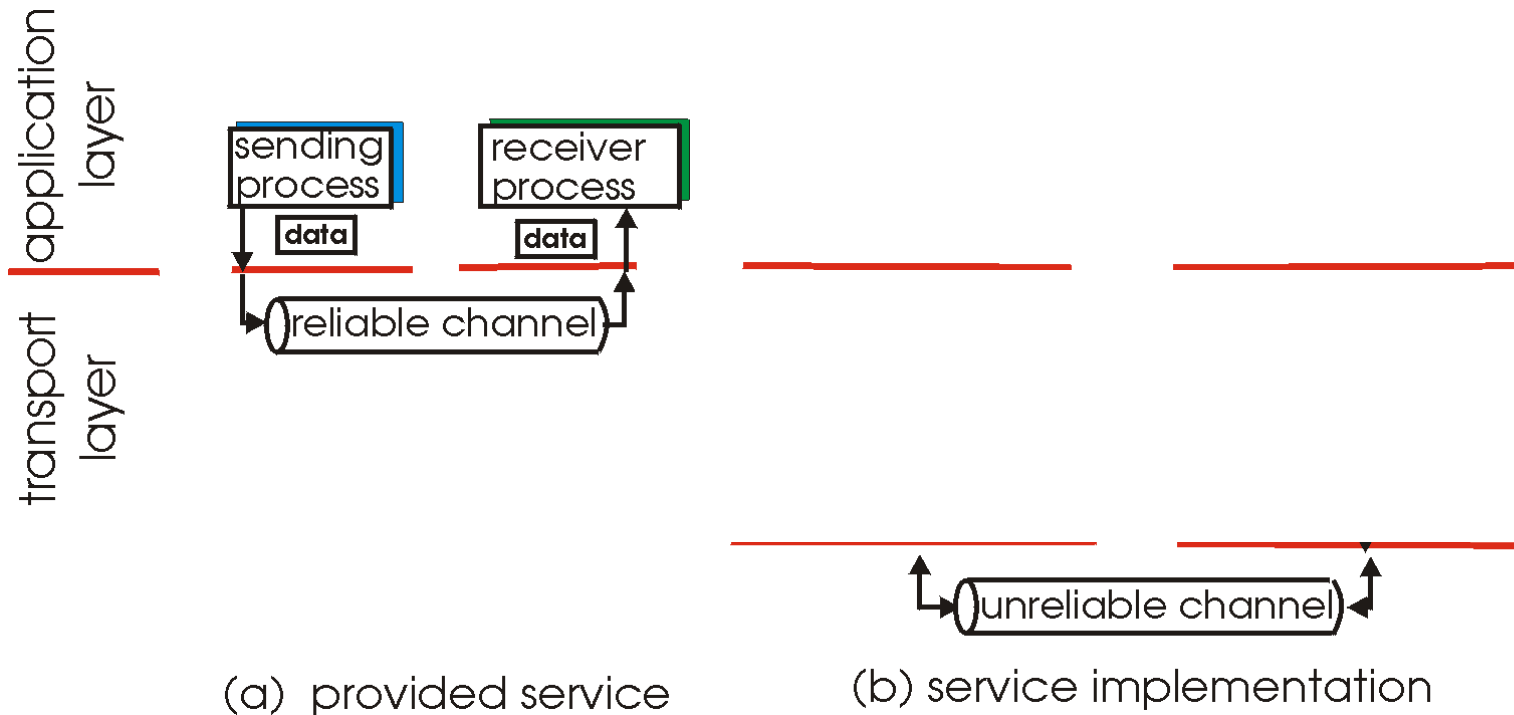


```
Socket clientSocket =  
    newSocket("hostname", "port  
    number");
```



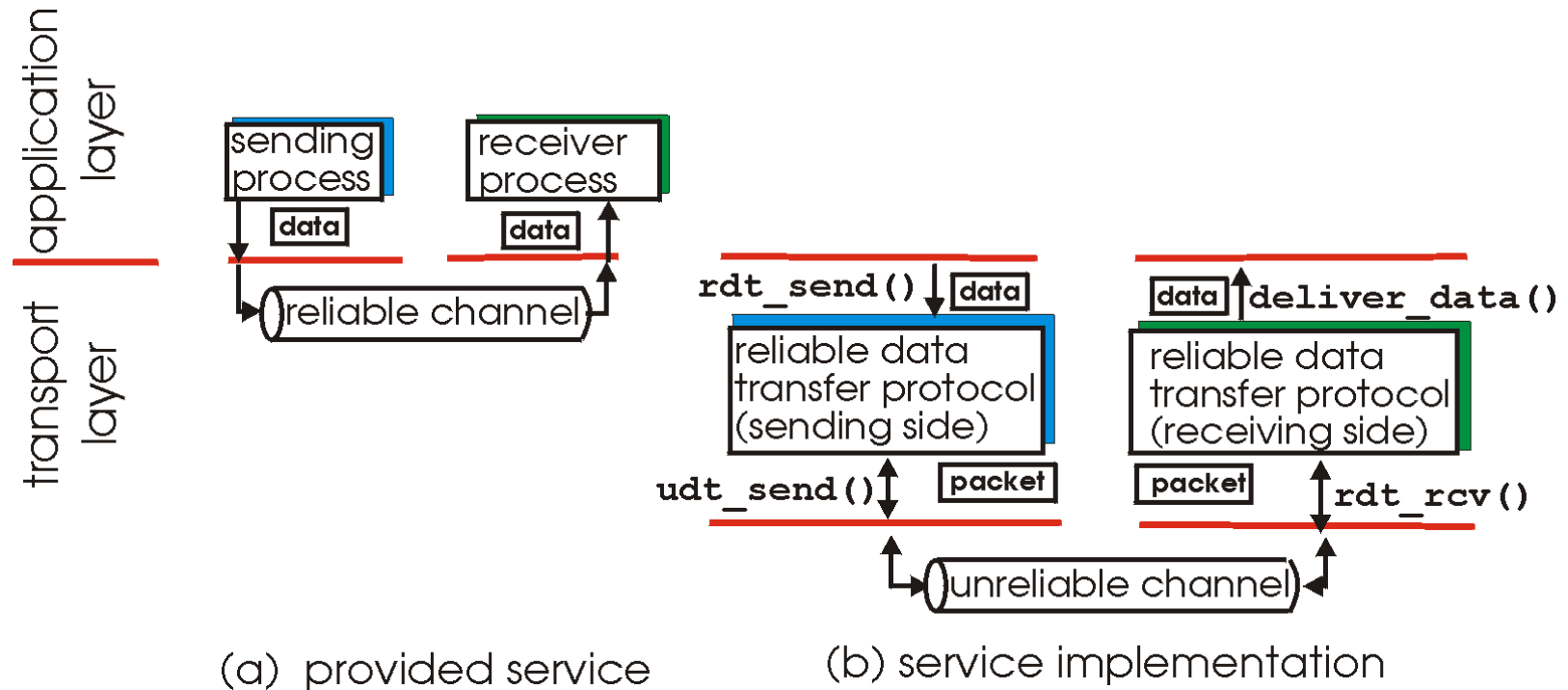
```
Socket connectionSocket =  
    welcomeSocket.accept();
```

- ❖ important in application, transport, link layers
 - top-10 list of important networking topics!



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

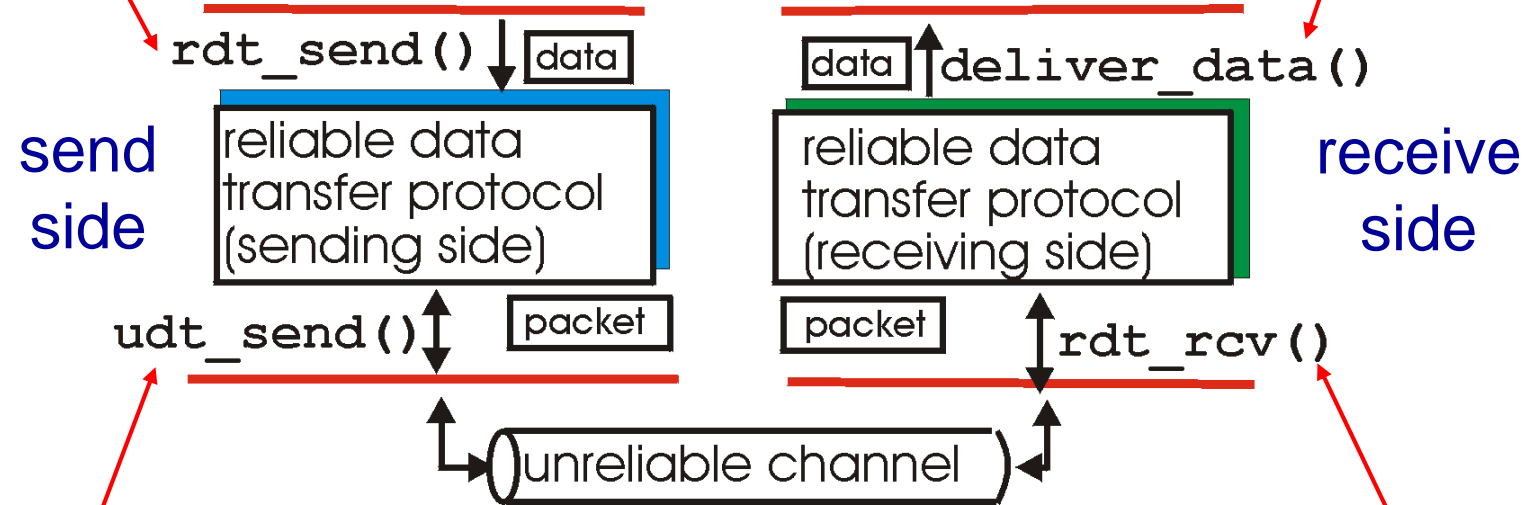
- ❖ important in application, transport, link layers
 - top-10 list of important networking topics!



- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol

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

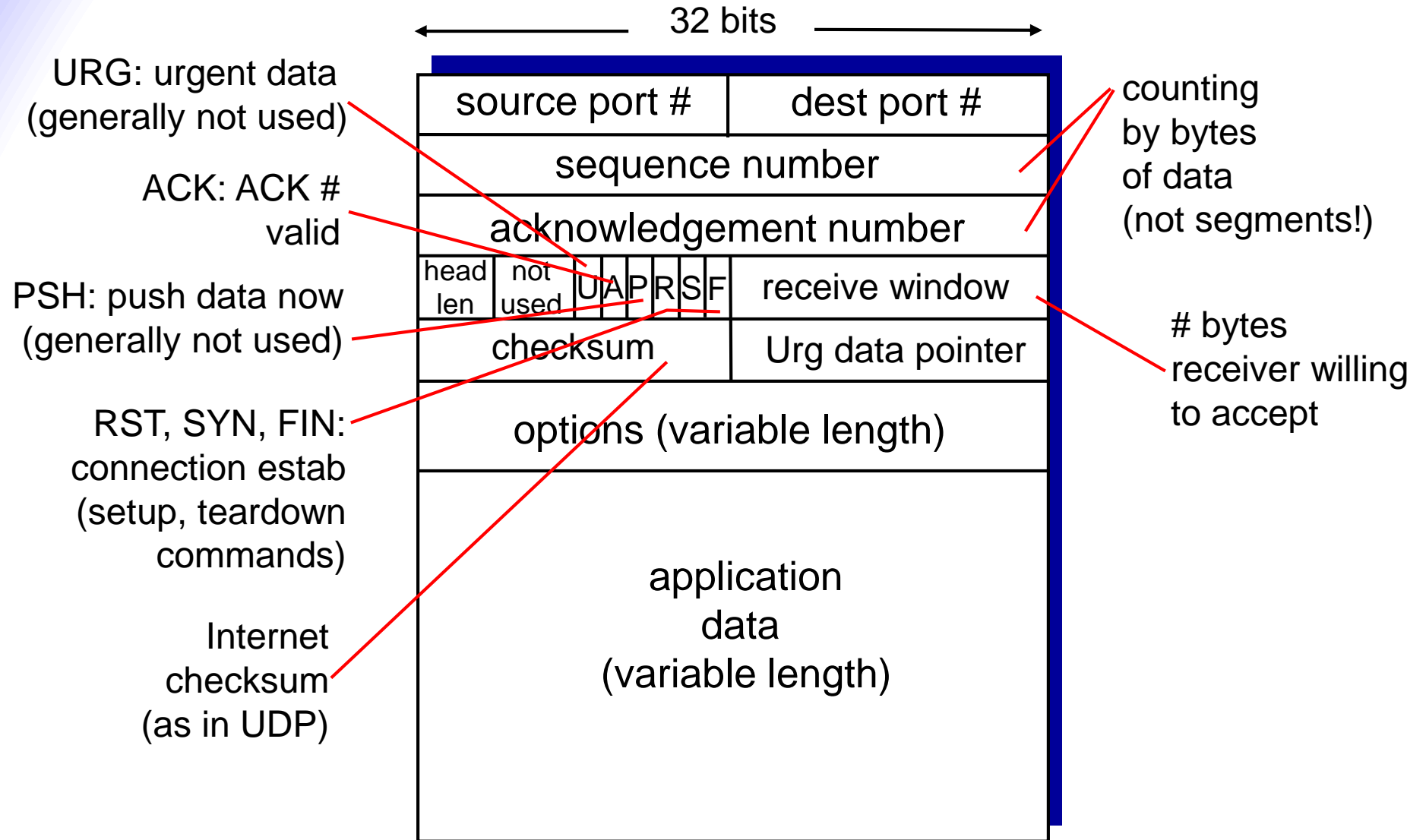
deliver_data() : called by
rdt to deliver data to upper



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

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

*Note: detail of reliable data transfer
is discussed in link layer*



- TCP views data as an unstructured, but ordered, stream of bytes
- sequence number is the byte-stream number of the first byte in the segment
- if host A sends 500,000 bytes to host B, maximum segment size is 1,000 bytes
- sequence number of the first segment is 0
- sequence number of the second segment is 1,000, and so on

- TCP connection provides full-duplex service
- host A receives data from host B, while it sends data to host B
- segment from host B has a sequence number
- the acknowledgment number that host A puts in its segment is the sequence number of the next byte host A is expecting from host B
- TCP connection is also always point-to-point (*multicasting is not possible*)

Summary

- ◆ transport-layer protocol – provides services to network applications
- ◆ simple service – multiplexing and demultiplexing
- ◆ UDP
- ◆ reliable data transfer - TCP

Reference

Chapter 3, Computer Networking: A Top-Down Approach

