Systeme 3

11a: Networking

Winter 2019/20

Marcel Waldvogel

Chapter Goals

- How does networking work in the OS?
  - How does data flow in server apps?
  - How do packets flow? How can this be optimized?
  - What is demultiplexing? How does it work?
- How does the application interface to it?
  - Why is VFS not used?
  - How to make effective file servers?

Creating a network app

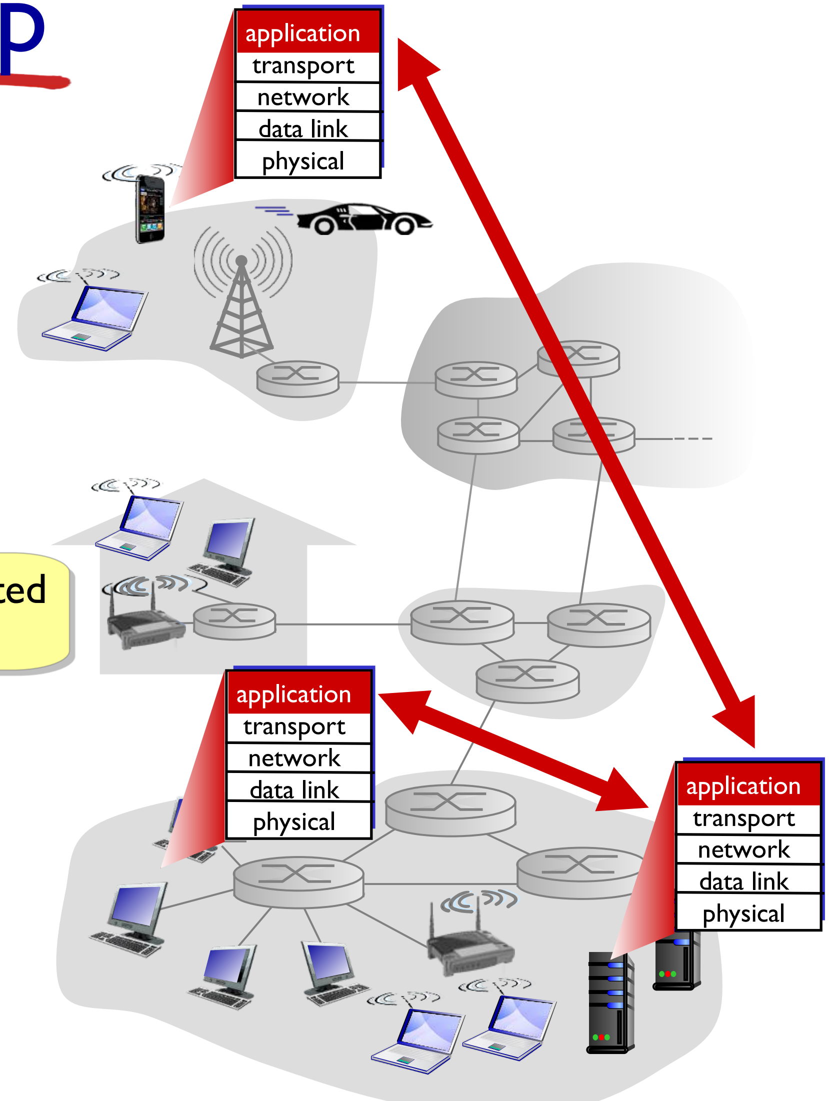
write programs that:

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

taken for granted today

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

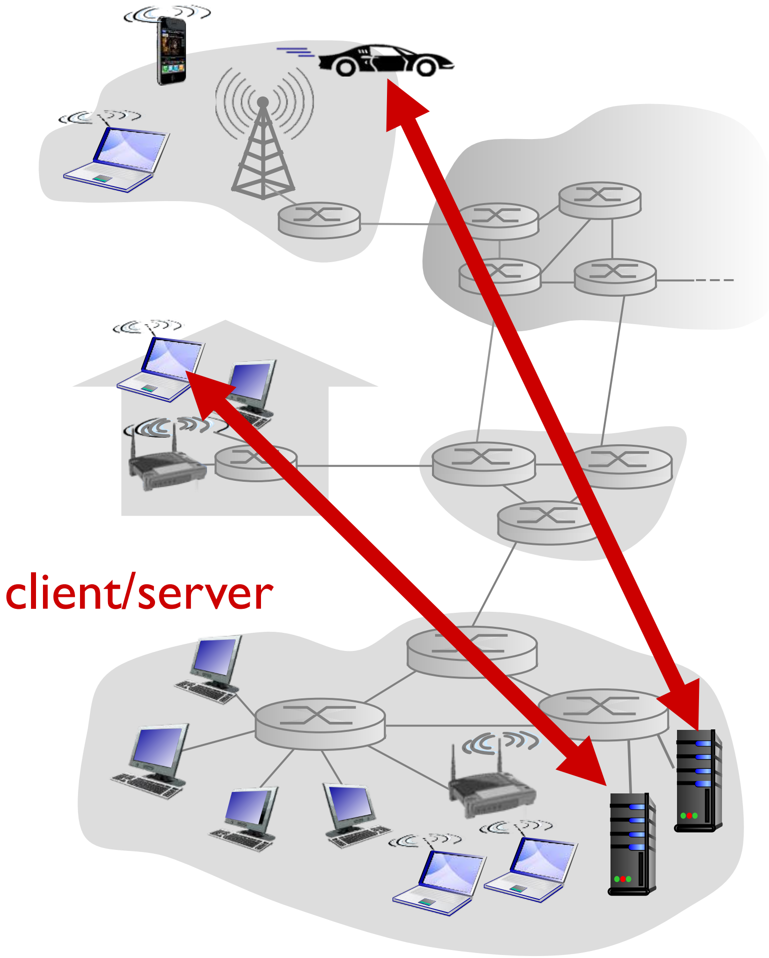


Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)
- federated

Client-server architecture



server:

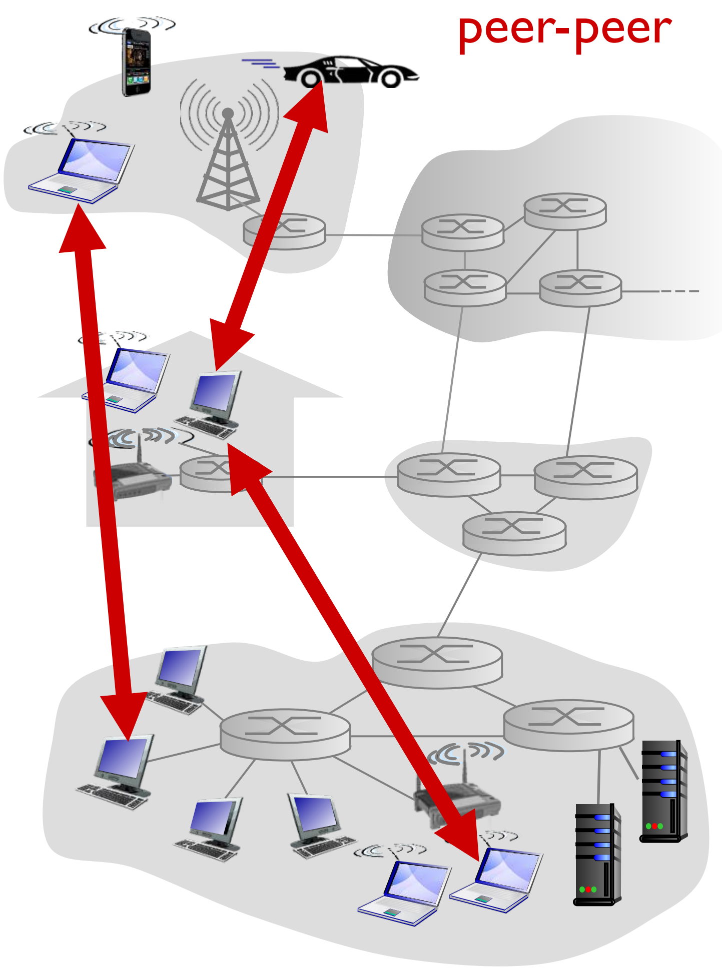
- always-on host
- permanent IP address
- data centers for scaling

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

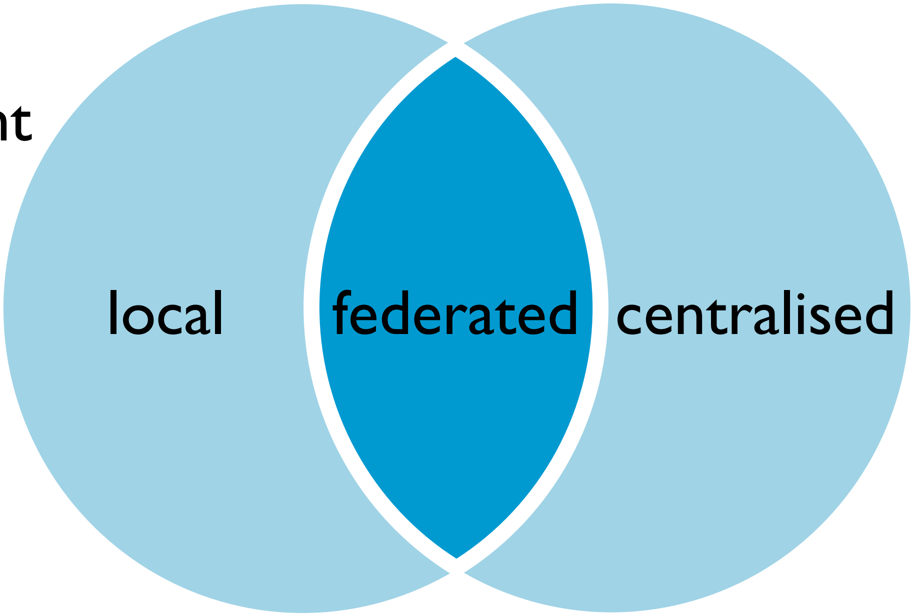
P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management



Federated Services

- Clients talk to their respective server: C/S
- Servers can freely talk among themselves:
  - like P2P
  - but always-on
  - with DNS entries to find each other
  - e.g., e-mail
  - resilient, independent



Processes communicating

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

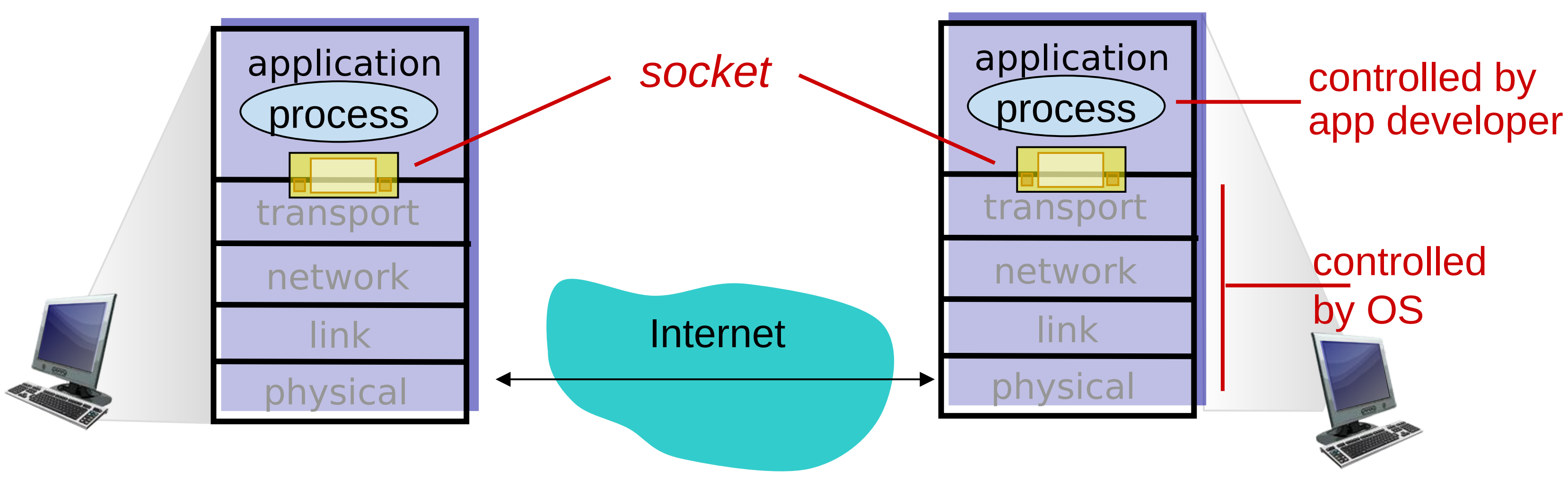
client process: process that initiates communication

server process: process that waits to be contacted

- aside: applications with P2P architectures have client processes & server processes

Sockets

- 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



Addressing processes

- 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 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 www.uni.kn web server:
  - IP address: 134.34.240.80
  - port number: 80
- more shortly…

Internet transport protocols services

TCP service:

- reliable transport between sending and receiving process
- flow control: sender won't overwhelm receiver
- congestion control: throttle sender when network overloaded
- does not provide: timing, minimum throughput guarantee, security
- connection-oriented: setup required between client and server processes

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup

Q: why bother? Why is there a UDP?

Securing TCP

TCP & UDP

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext

TLS (formerly SSL)

- provides encrypted TCP connection
- data integrity
- end-point authentication

TLS is at app layer

- apps use TLS libraries, that "talk" to TCP

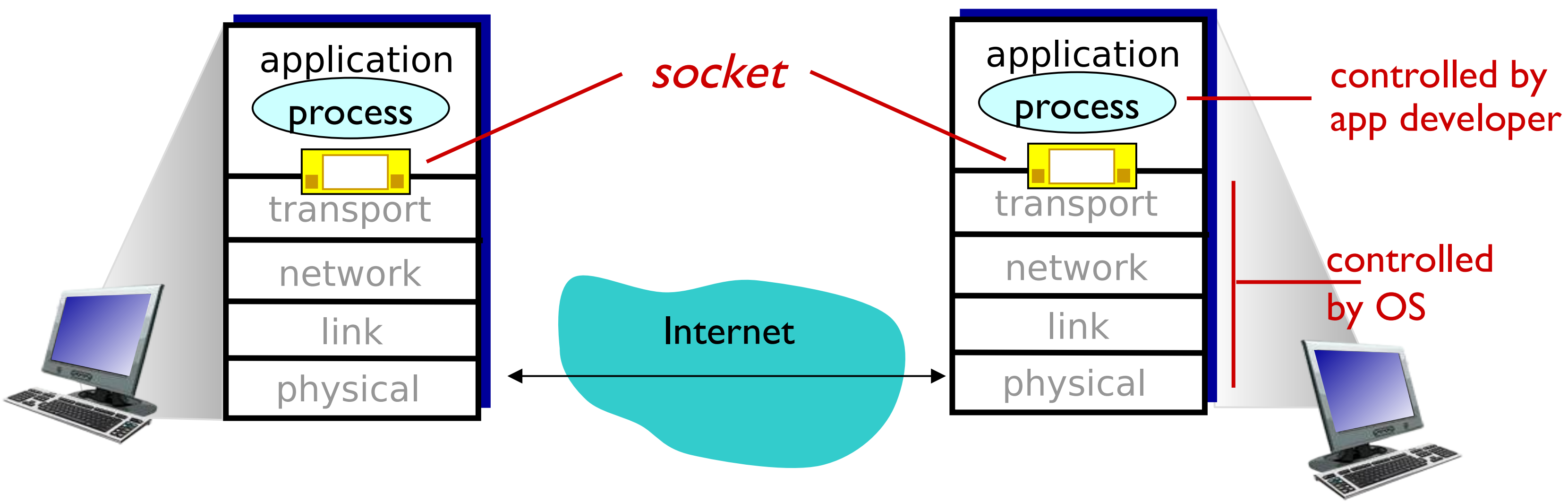
TLS socket API

- cleartext passwords sent into socket traverse Internet encrypted
- see Chapter 8

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- UDP: unreliable datagram
- TCP: reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming *with UDP*

UDP: no "connection" between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP

server (running on serverIP)

create socket, port= 7:

```
struct sockaddr_in6 sa;
sock_fd = socket(AF_INET6,
                 SOCK_DGRAM);
memset(&sa, 0, sizeof(sa));
sa.sin6_family = AF_INET6;
sa.sin6_port = MAGIC(port);
err = bind(serverSocket,
          &sa, sizeof(sa));
```

htons(): Host to Network, Short

```
char buf[MAX];
struct sockaddr_in6 remote_addr;
len = recvfrom(sock_fd, buf, sizeof(buf), 0,
      &remote_addr, sizeof(remote_addr));
```

```
sent = sendto(sock_fd, buf, len, 0,
      &remote_addr, sizeof(remote_addr));
```

client

create socket:

```
sock_fd = socket(AF_INET6,
                 SOCK_DGRAM);
```

```
struct addrinfo *ai;
struct addrinfo hints = {
      .ai_flags = AI_V4MAPPED
          | AI_ADDRCONFIG,
      .ai_family = AF_INET6,
      .ai_socktype = SOCK_DGRAM };
err = getaddrinfo("servername", "echo",
      &hints, &ai);
sendto(sock_fd, msg, msglen, 0,
      (struct sockaddr_in6*)ai->ai_addr,
      sizeof(struct sockaddr_in6));
```

read datagram from clientSocket

close clientSocket

`freeaddrinfo(…)`

Avoid IPv4-only code

- Even when your network/system does not support IPv6 yet, always write IPv6-capable code
  - Because it will, soon
- Tool at hand: IPv6-mapped IPv4 addresses
  - IPv4 address
    - `192.0.2.10`, `127.0.0.1`, `0.0.0.0`
  - IPv6 address
    - `2001:0db8:1234:5678:0123:45ff:fe01:2345`, `::1 (= 0:0:0:0:0:0:0:1 = 0000:0000:0000:0000:0000:0000:0000:0001)`, `::`
  - IPv6-mapped IPv4 address
    - `::192.0.2.10`
    - Can be used like an IPv6 address in the application, will behave as an IPv4 address on the network.

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:

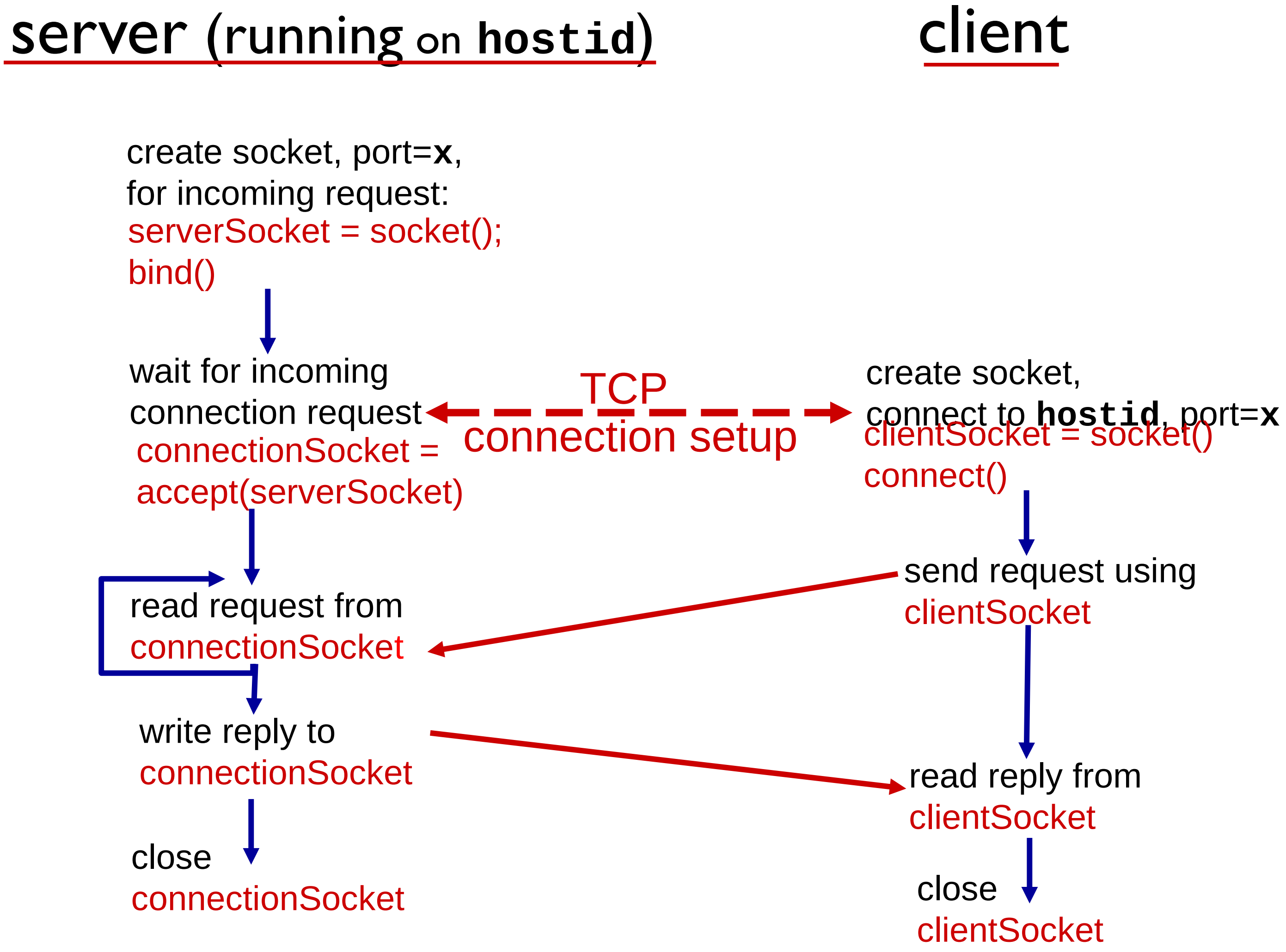
- Creating TCP listening socket, specifying IP address, port number of server process
- when client creates socket: client TCP establishes connection to server TCP

- when contacted by client, server TCP creates new (connected/accepted) socket for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chapter 3)

application viewpoint:

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

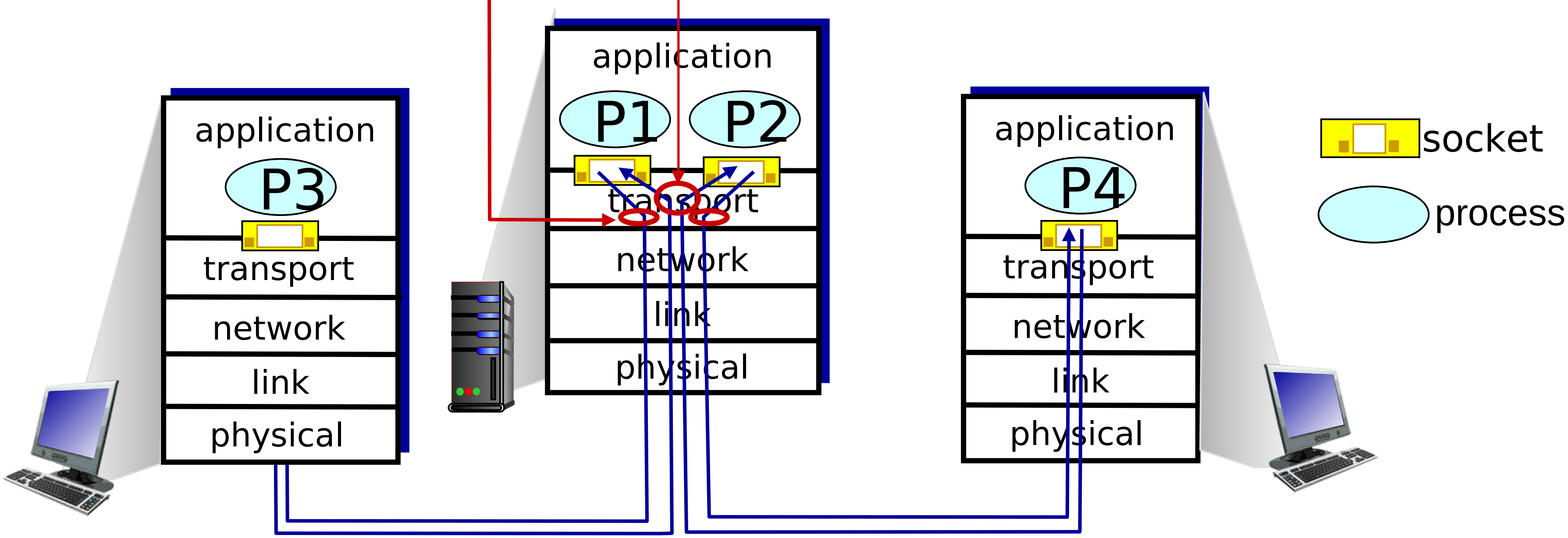
Client/server socket interaction: TCP



Multiplexing/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

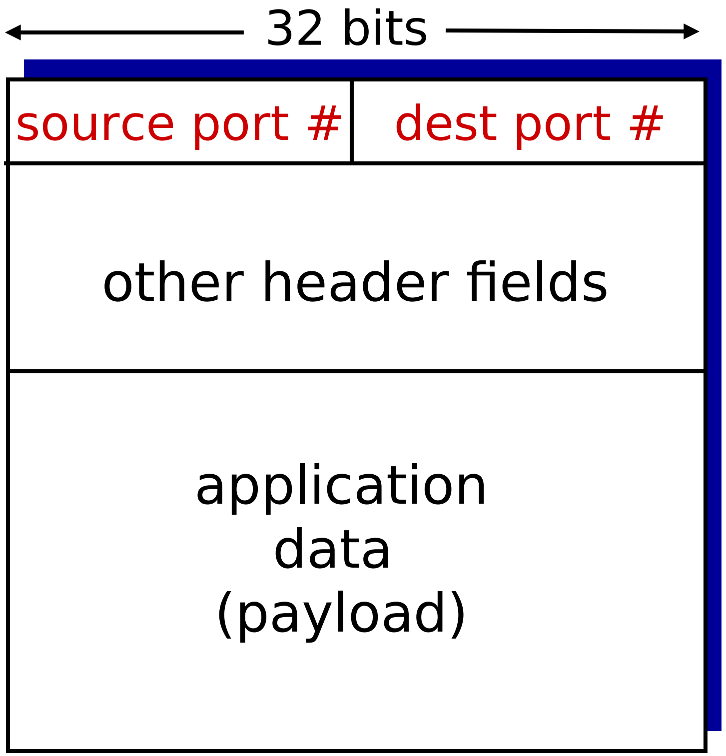


How demultiplexing works

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

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

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

- *recall:* 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 #

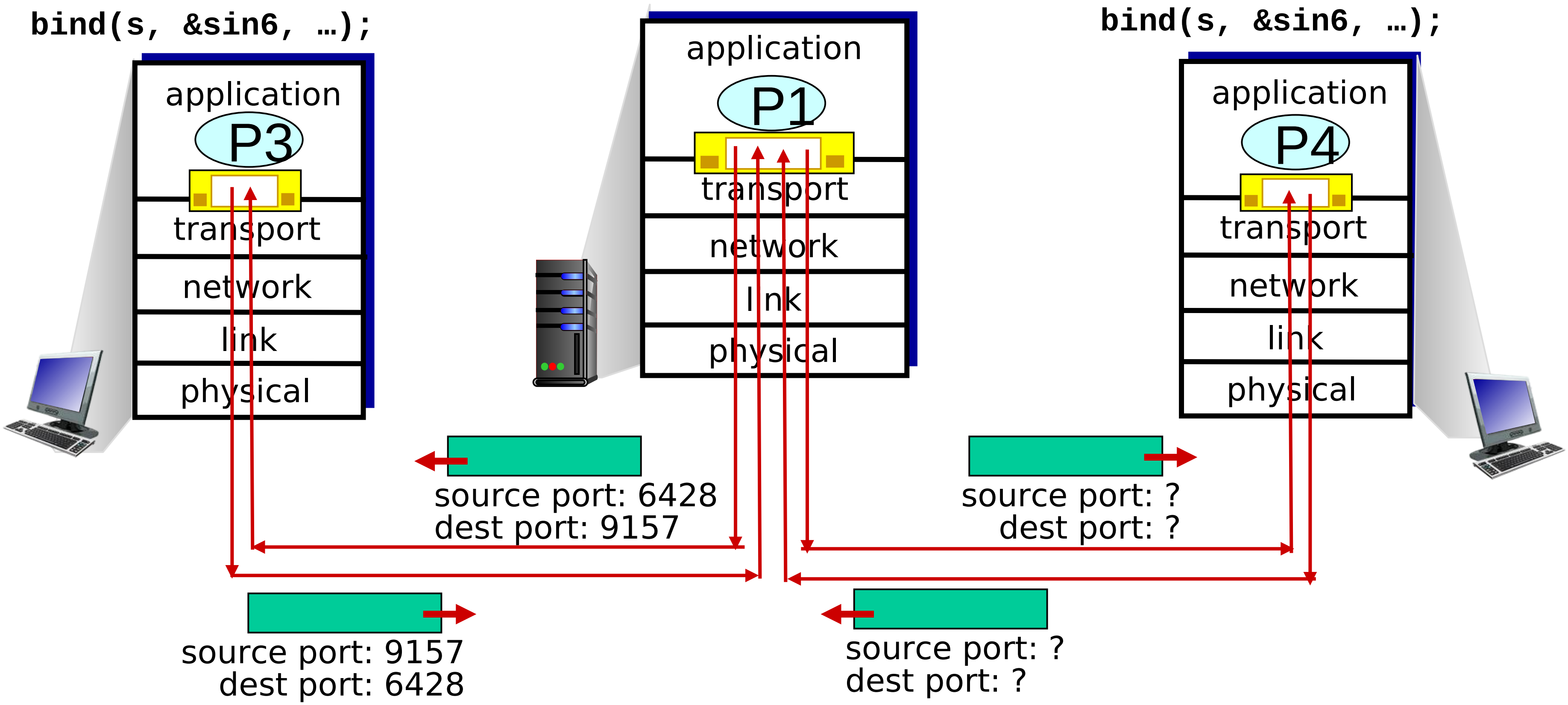
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
s = socket(AF_INET6,
 SOCK_DGRAM, 0);
…
sin6.in6_port = 9157;
bind(s, &sin6, …);
```

```
s = socket(AF_INET6,
 SOCK_DGRAM, 0);
…
sin6.in6_port = 6428;
     bind(s, &sin6, …);
```

```
s = socket(AF_INET6,
 SOCK_DGRAM, 0);
…
sin6.in6_port = 5775;
bind(s, &sin6, …);
```



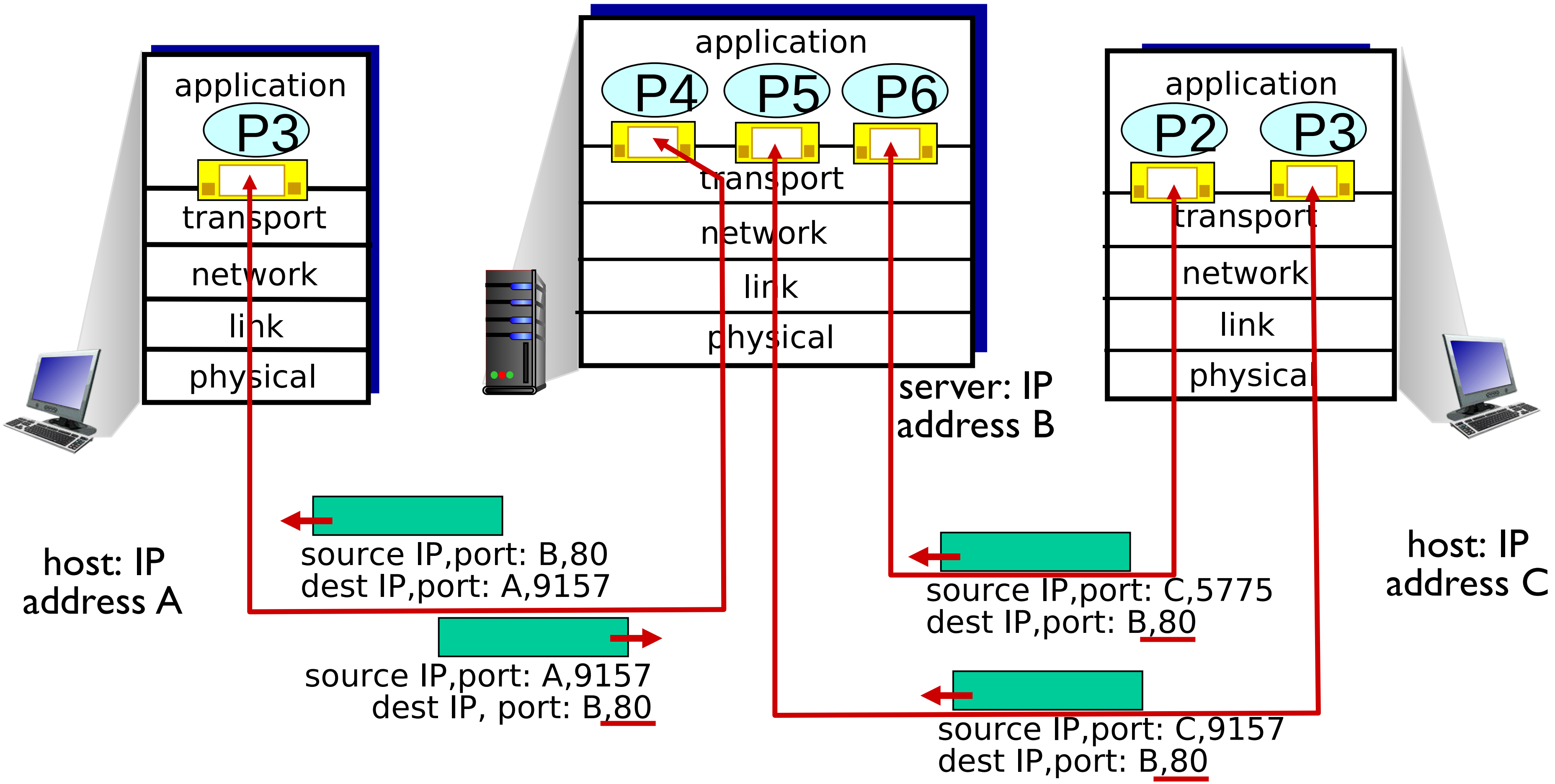
Connection-oriented demux

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

5-tuple
incl. protocol

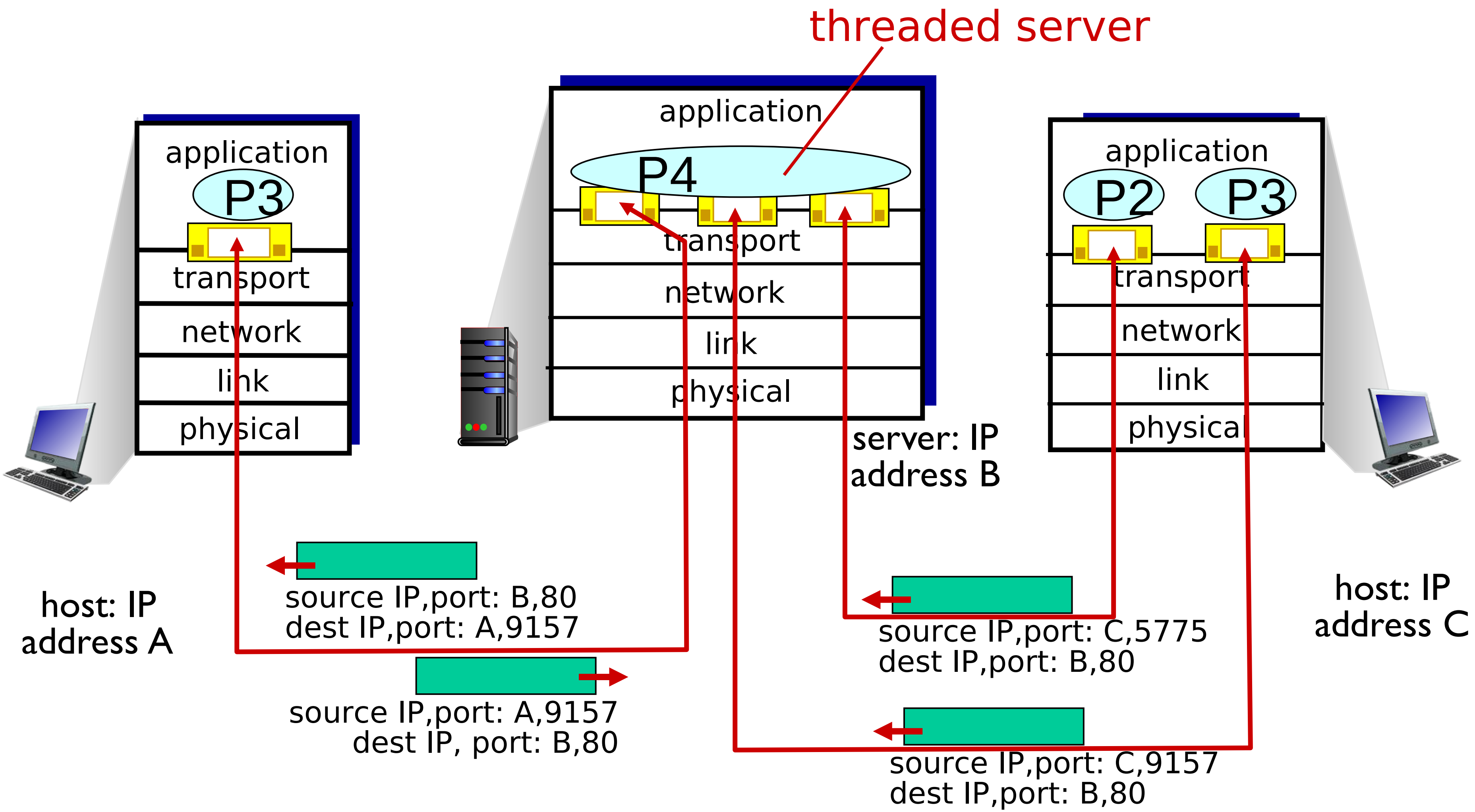
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



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

Connection-oriented demux: example



Demux rules

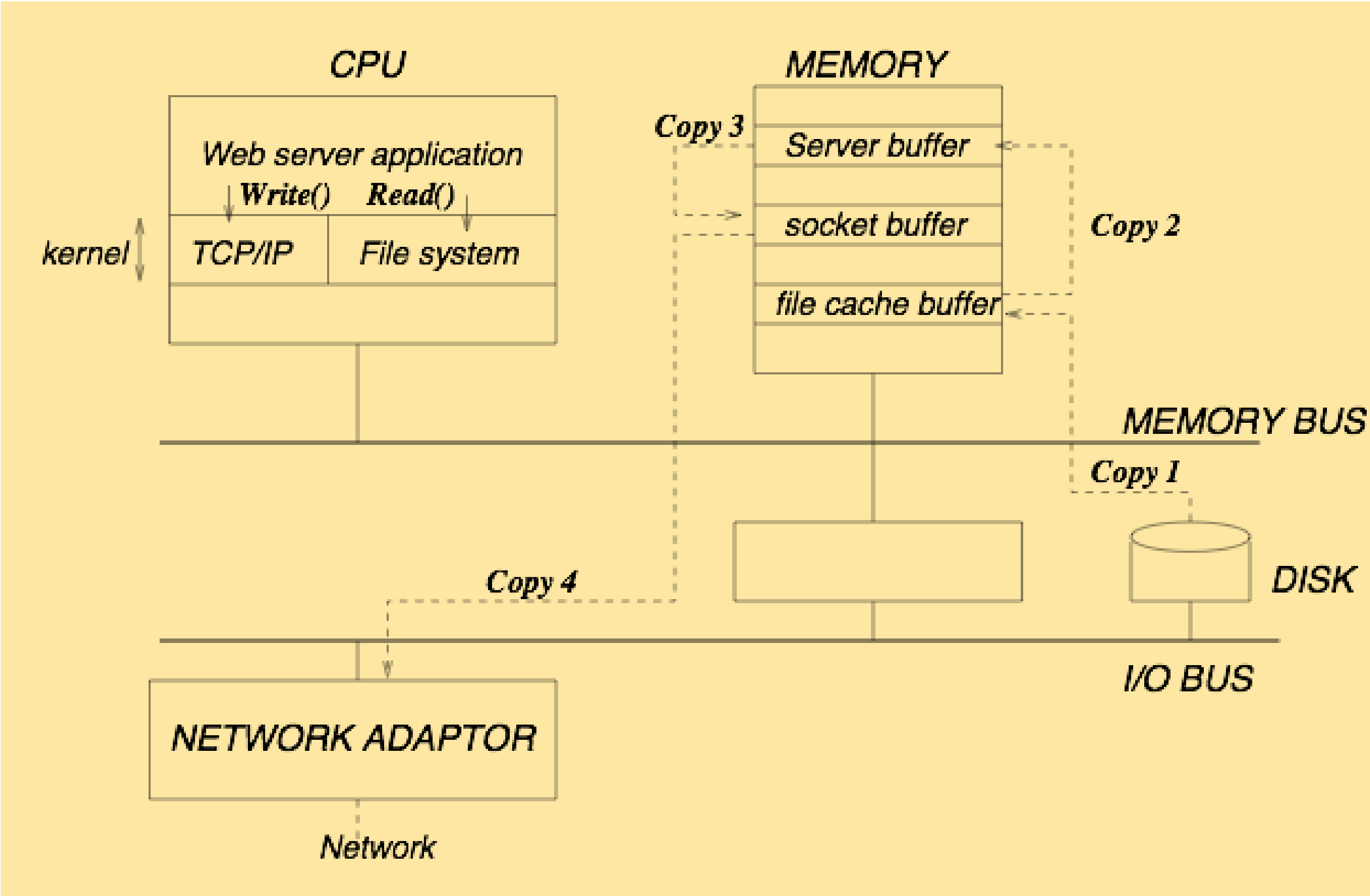
- TCP
  - Acceptor socket
    - source (remote): `*:*` (connection requests only)
    - destination (local): `*:port` or `local-addr:port`
  - Connection socket
    - source (remote): `rem-addr:rem-port` (data packets only)
    - destination (local): `local-addr:port`
- UDP
  - bound-only socket
    - source (remote): `*:*` (all data packets)
    - destination (local): `*:port` or `local-addr:port`
  - bound and connected socket
    - source (remote): `rem-addr:rem-port` (all data packets)
    - destination (local): `local-addr:port`

- Demultiplexing
  - Hash table with 3-tuples/5-tuples
  - (Firewall rules)
- Packet sending/reception
  - Headers (IP addresses, port numbers, flags, sequence numbers, ...) need to be
    - prepended on send,
    - removed on receive
  - `packet = header + packet` is not efficient
  - mbuf (memory buffer)/skbuf (socket buffer) has space reserved in front of the user data
    - (pointer to) data, start offset, length

Buffering in the Kernel

- File Buffers
  - Every read() from disk causes that disk block to be stored in the buffer cache
  - If sequential read()s are detected, the kernel also does prefetching
- Networking
  - TCP sending: Copies of all outgoing packets are kept until they have been acknowledged by the remote side (reliable transfer!)
  - TCP reception: Incoming packets are kept until all holes are filled (dealing with network loss/reordering)
  - TCP/UDP reception: Incoming packets are kept until the application is ready to receive them

File/web server



Speeding up copying

- Traditional API: diverse workload, mixed operations
- Frequent patterns should be optimized, e.g. with additional APIs: Exploit…
  - Degrees of freedom
  - „Intelligent“ hardware
  - Free yourself from reference implementations
- New abstractions, e.g. Linux sendfile(2)

```
SENDFILE(2)              Linux Programmer's Manual              SENDFILE(2)

NAME
       sendfile - transfer data between file descriptors

SYNOPSIS
       #include <sys/sendfile.h>

       ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```