# CS 43: Computer Networks

## 05:Network Services and Distributed Systems

September 17, 2019

# Reading Quiz

# Last class

- Inter-process communication using message passing
- How send and recv buffers work
- Concurrency

# Today

- Concurrency
- Application-layer communication paradigms:
  - Client-Server
  - Peer-to-peer architecture
- Distributed network applications: Sources of complexity

# Thread-based vs. Event-based Concurrency

# Five-Layer Internet Model

Application: the application (e.g., the Web, Email)

Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

Physical: 1's and 0's/bits across a medium (copper, the air, fiber)

# Where we are

Application: the application (e.g., the Web, Email)

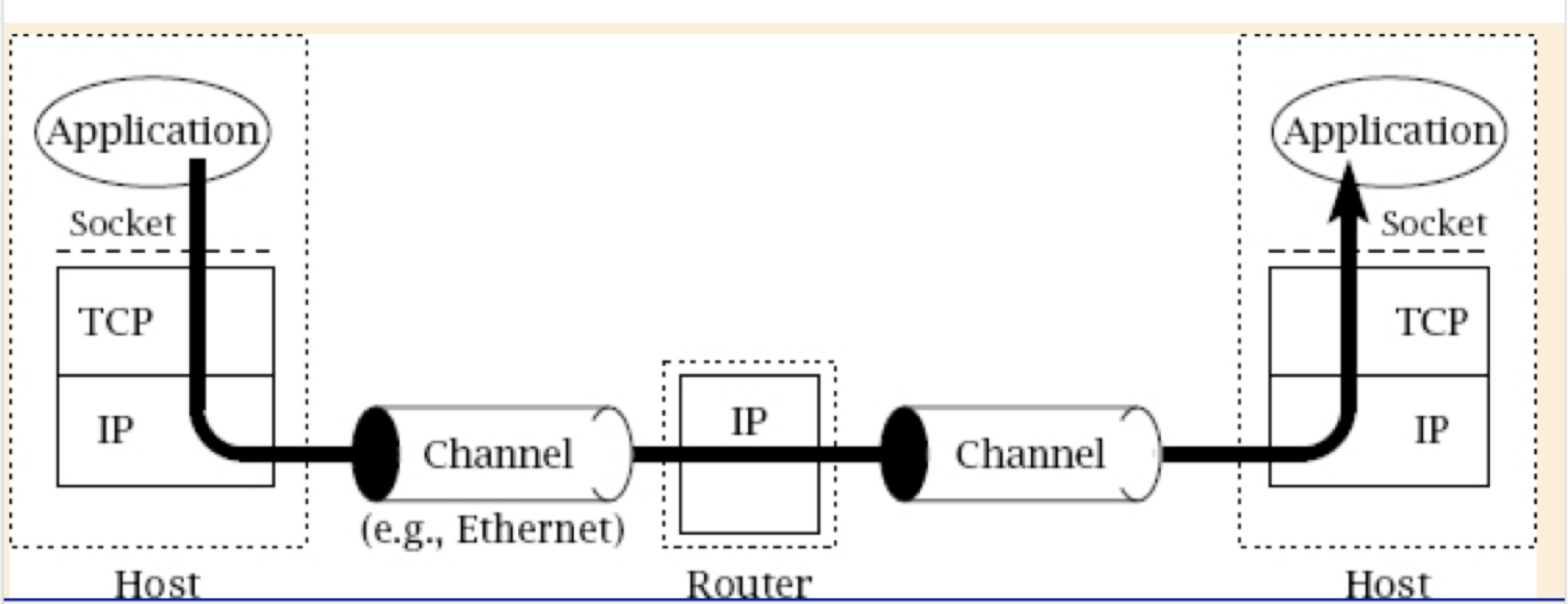
Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

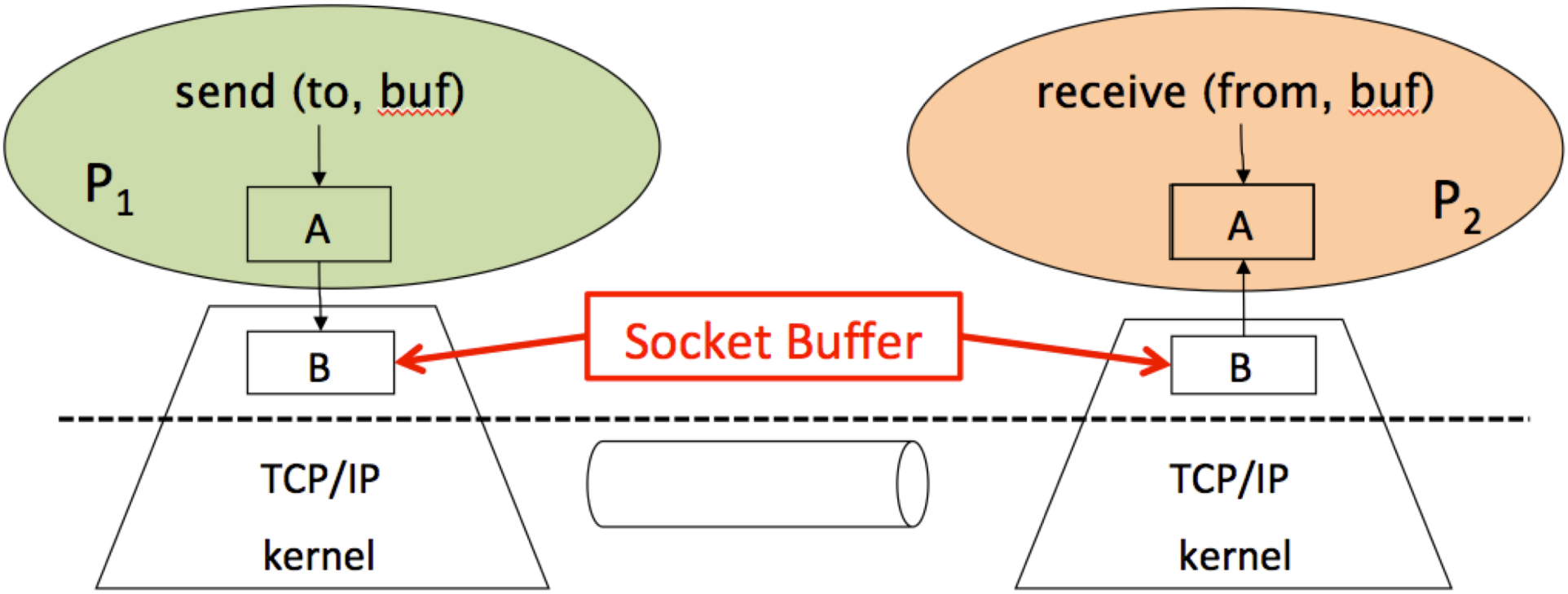
Physical: 1's and 0's/bits across a medium (copper, the air, fiber)

# Client-Server communication




Adapted from: Donahoo, Michael J., and Kenneth L. Calvert. TCP/IP sockets in C: practical guide for programmers. Morgan Kaufmann, 2009.

# Inter-process Communication (IPC): Network



# Blocking Summary

**send()**

- Blocks when socket buffer for sending is full
- Returns less than requested size when buffer cannot hold full size

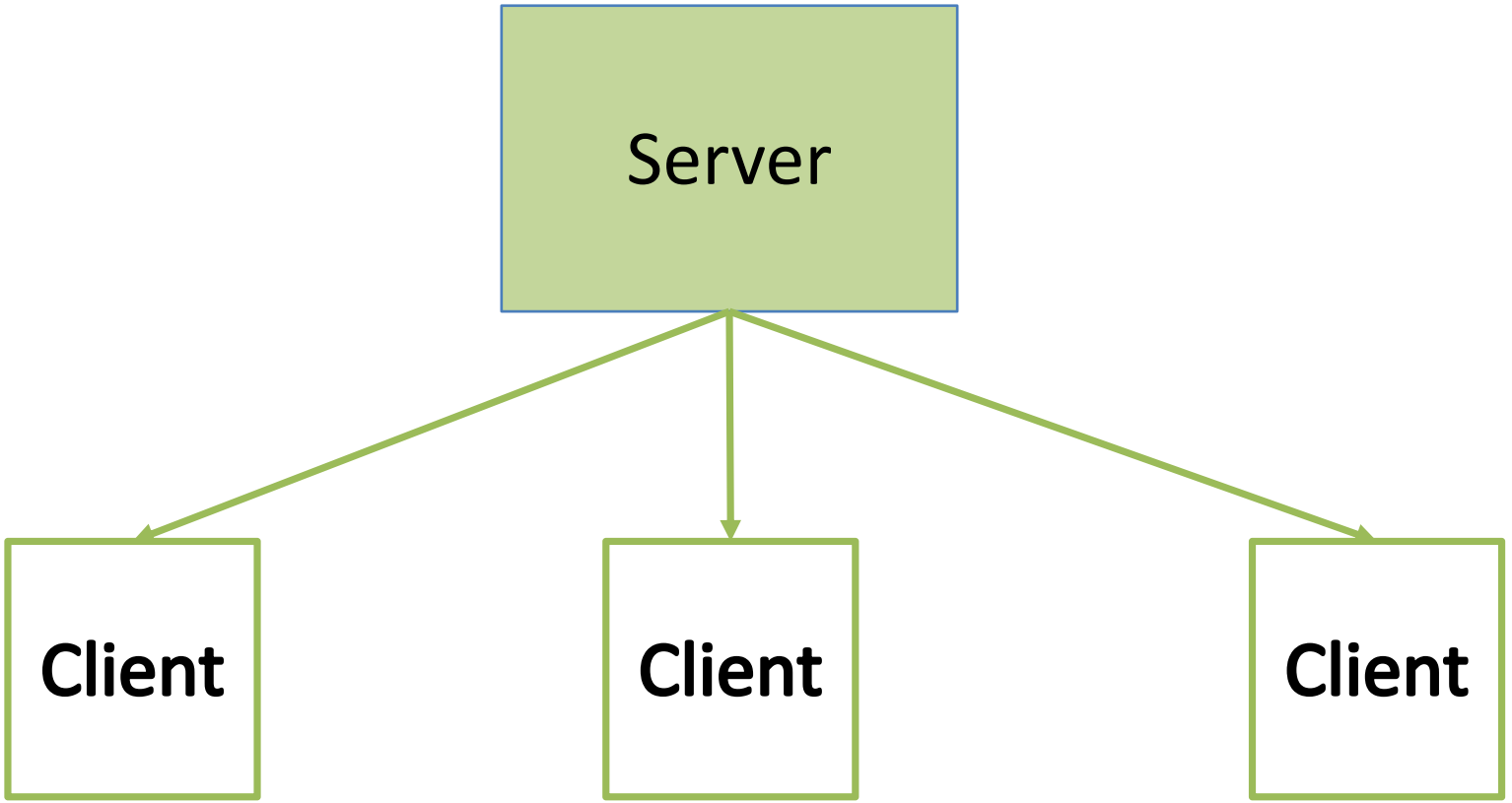
**recv()**

- Blocks when socket buffer for receiving is empty
- Returns less than requested size when buffer has less than full size

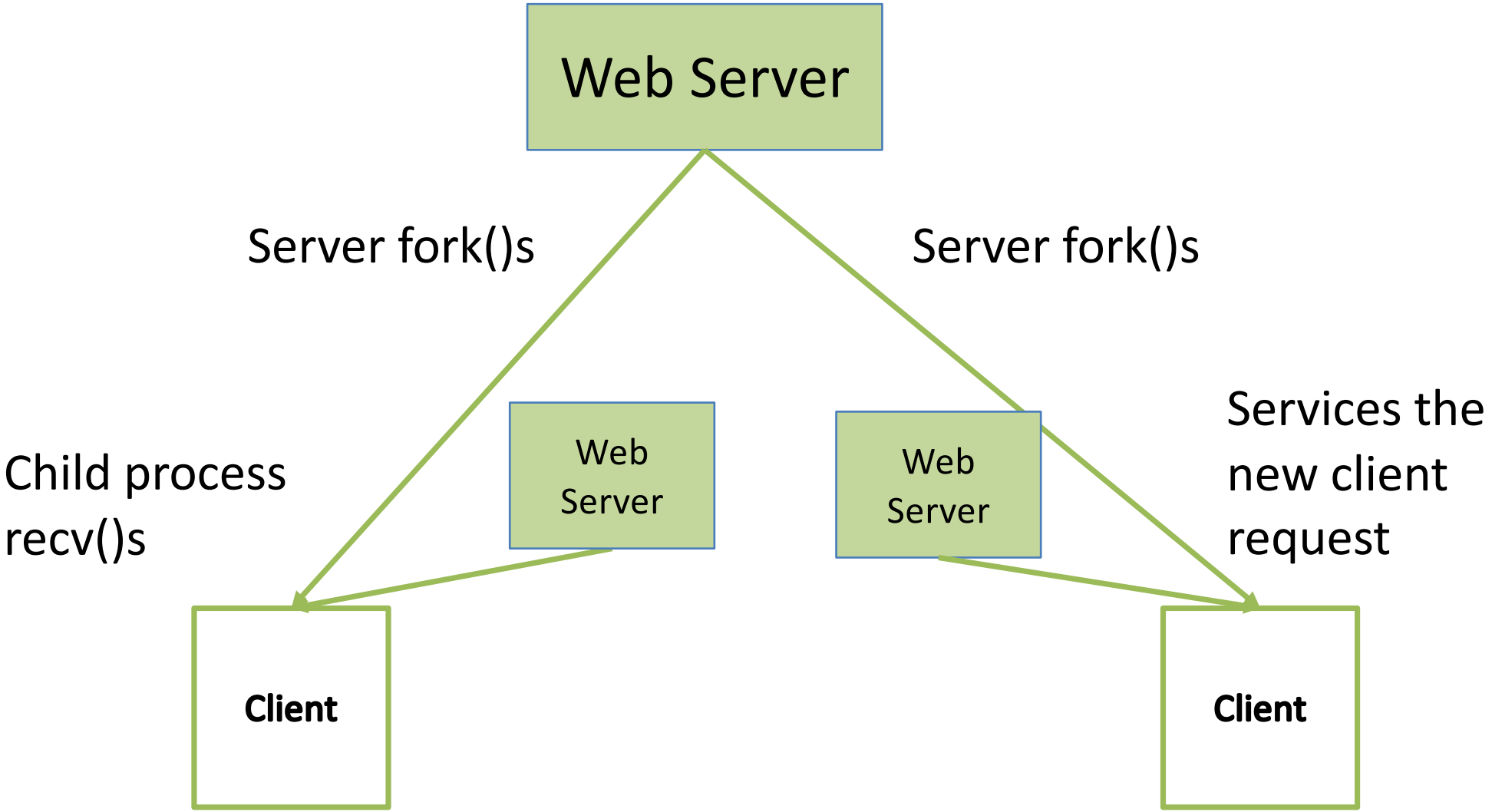
Always check the return value!

# Concurrency

- Think you're the only one talking to that server?

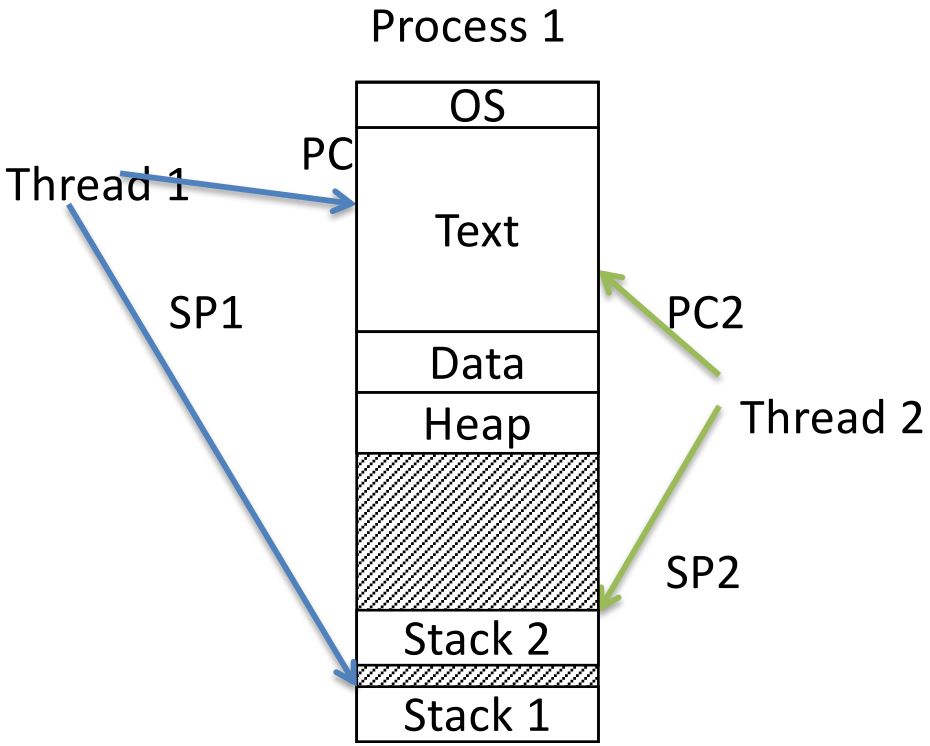


# Concurrent Webserver.

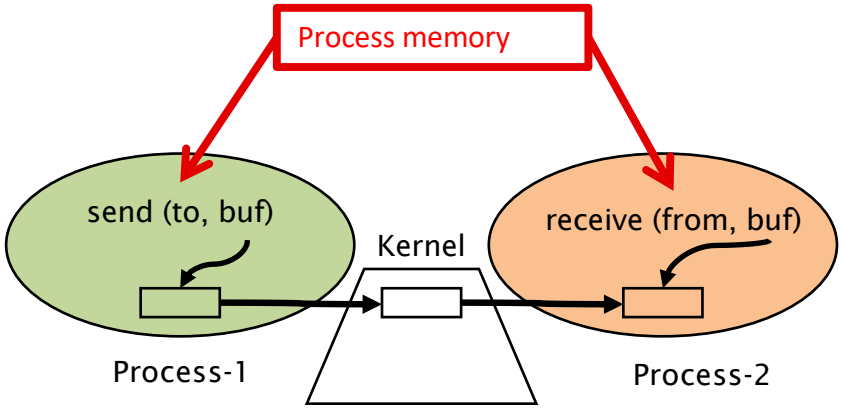


# Concurrent Web-servers with multiple threads/processes

- Threads (shared memory)




- Message Passing (locally)



# Which benefit is most critical?

A. Modular code/separation of concerns.

B. Multiple CPU/core parallelism.

C. I/O overlapping.

D. Some other benefit.

# How would you use threads to achieve faster performance in the following programs? (answer in the worksheet)

- A program that displays the squares of numbers 1 to 10 (in any order).
- A program that serves data to multiple clients.
- A program that reads data from files and sends them over the network.
- Program that reads files from the disk and prints the output.

# Event-based concurrency

- Blocking: synchronous programming
  - wait for I/O to complete before proceeding
  - control does not return to the program
- Non-blocking: asynchronous programming
  - control returns immediately to the program
  - perform other tasks while I/O is being completed.
  - notified upon I/O completion

# Non-blocking I/O

One operation: add a flag to send/recv

- Permanently, for socket: fcntl() – "file control"
  - Allows setting options on file/socket descriptors

```
int sock, result, flags = 0;
sock = socket(AF_INET, SOCK_STREAM, 0);
result = fcntl(sock, F_SETFL, flags|O_NONBLOCK)
```

*always check the result!*

# Will this work?

```
server_socket = socket(), bind(), listen() //non-blocking
connections = []
while (1)
  new_connection = accept(server_socket)
  if new_connection != -1,
      add it to connections
  //end if
  for connection in connections:
     recv(connection, …)  // Try to receive
     send(connection, …) // Try to send, if needed
 //end for
//end of while
```

# Will this work?

A. Yes, this will work.
B. No, this will execute too slowly.
C. No, this will use too many resources.
D. No, this will still block.

```
server_socket = socket(), bind(), listen() //non-blocking
connections = []
while (1)
  new_connection = accept(server_socket)
  if new_connection != -1,
     add it to connections
  for connection in connections:
     recv(connection, …)  // Try to receive
     send(connection, …) // Try to send, if needed
```

# Event-based concurrency: select()

- Rather than checking over and over, let the OS tell us when data can be read/written
- Create set of file/socket descriptors we want to read and write
- Tell system to block until at least one of those is ready for us to use. The OS worries about selecting which one(s).

# Event-based concurrency

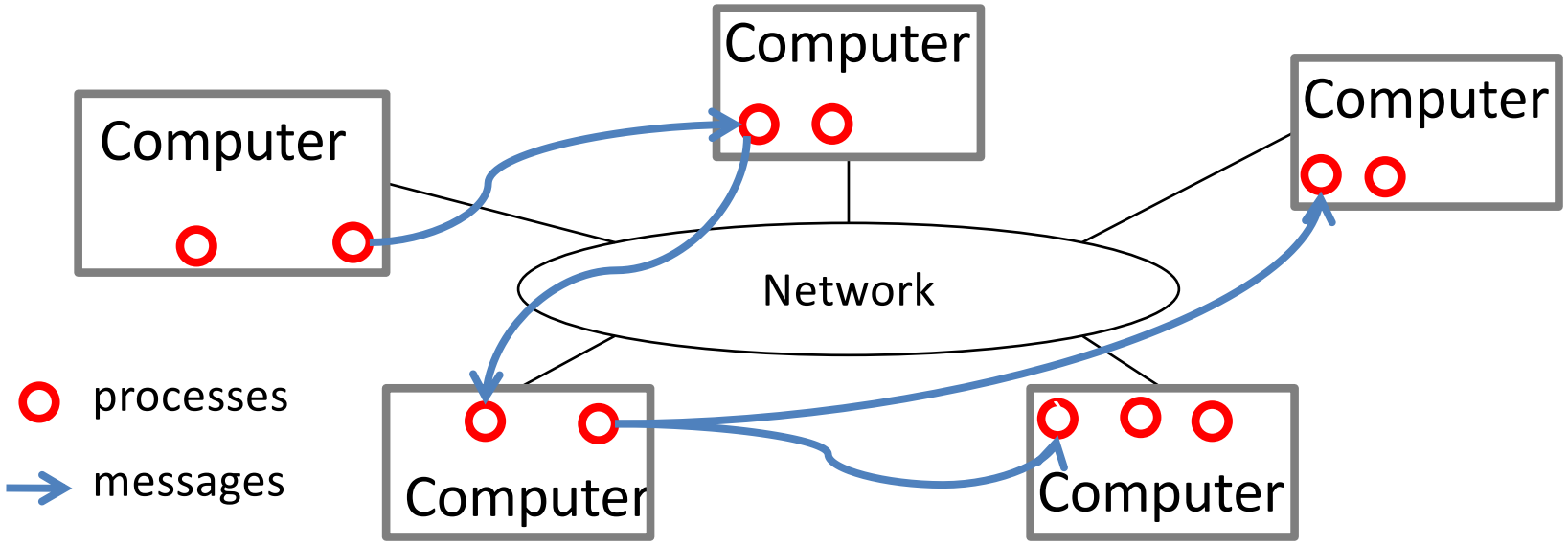
- Only one process/thread (or one per core)
  - No time wasted on context switching
  - No memory overhead for many processes/threads

# How would you design a real-world concurrent server? (Explain)

A. Event-based for serving clients.
B. Thread-based for serving clients.
C. Combination: thread-based for some tasks and event-based for others (how would you divide tasks)?

# Distributed Network Applications

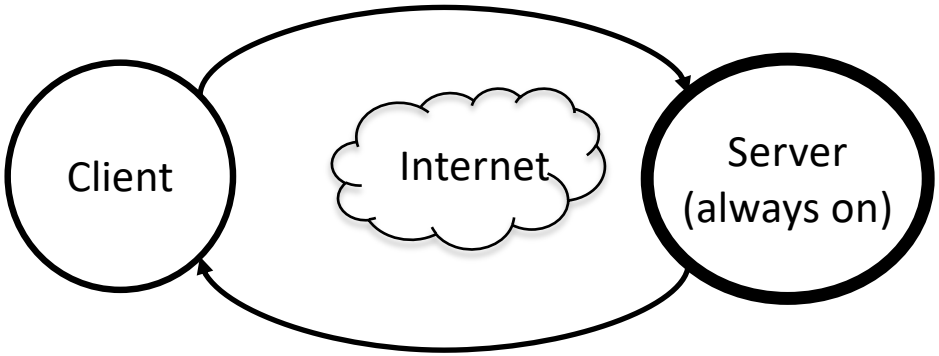
# What is a distributed application?



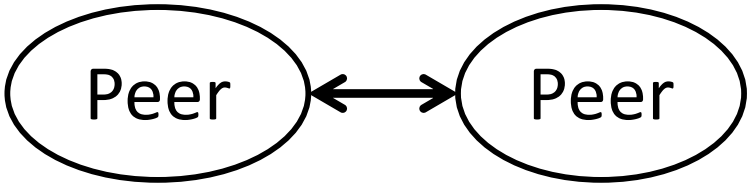

- Cooperating processes in a computer network
- Varying degrees of integration
  - Loose: email, web browsing
  - Medium: chat, Skype, remote execution, remote file systems
  - Tight: process migration, distributed file systems

# Designating roles to an endpoint

Client-server architecture




Peer-to-peer architecture



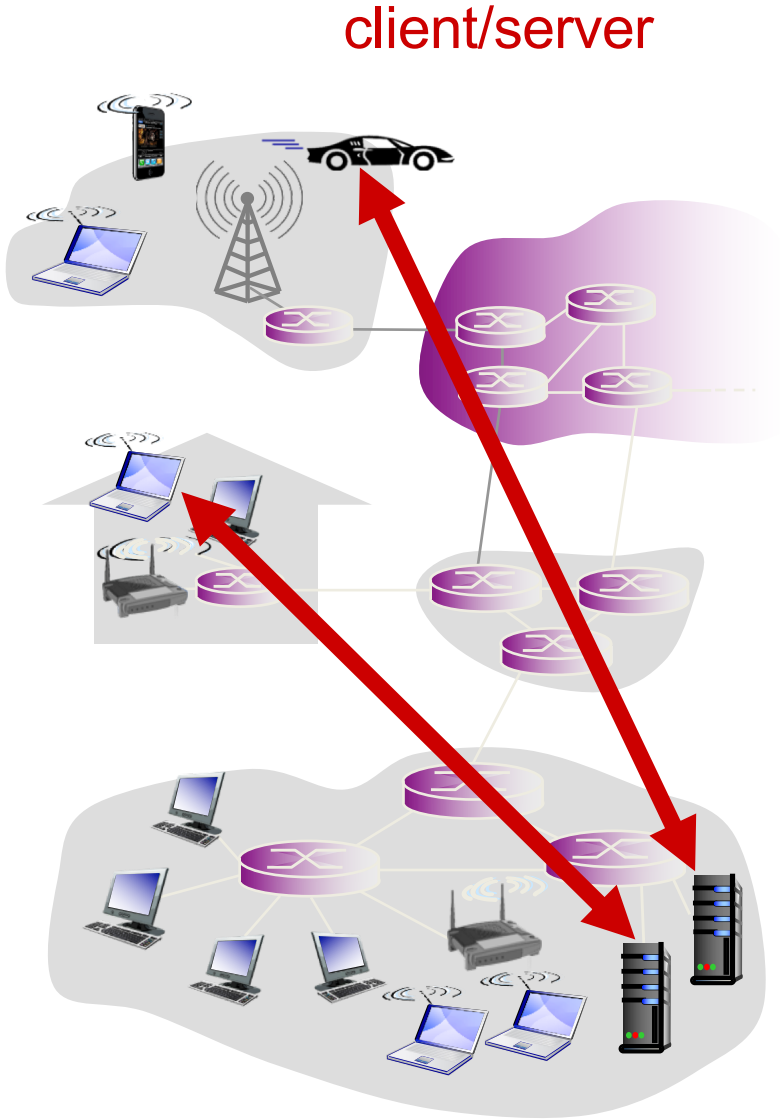
# Client-server architecture

server:

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

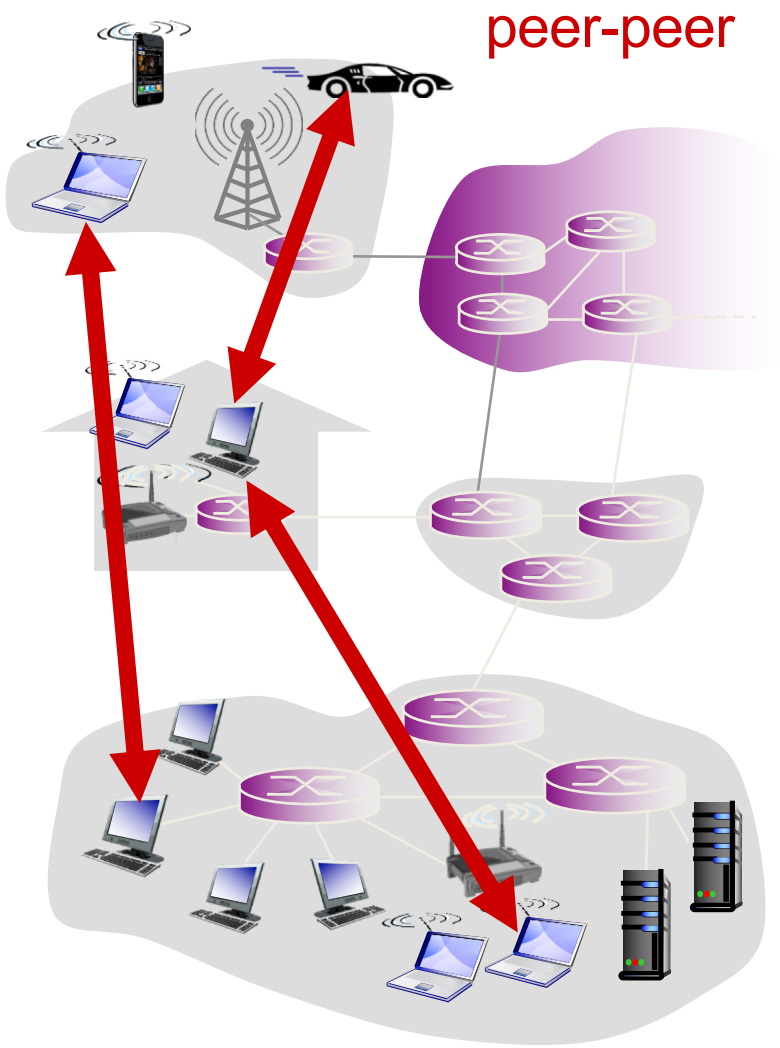
clients:

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



# Peer-to-Peer

- no always-on server
- A peer talks directly with another peer
  - Symmetric responsibility (unlike client/server)
- 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

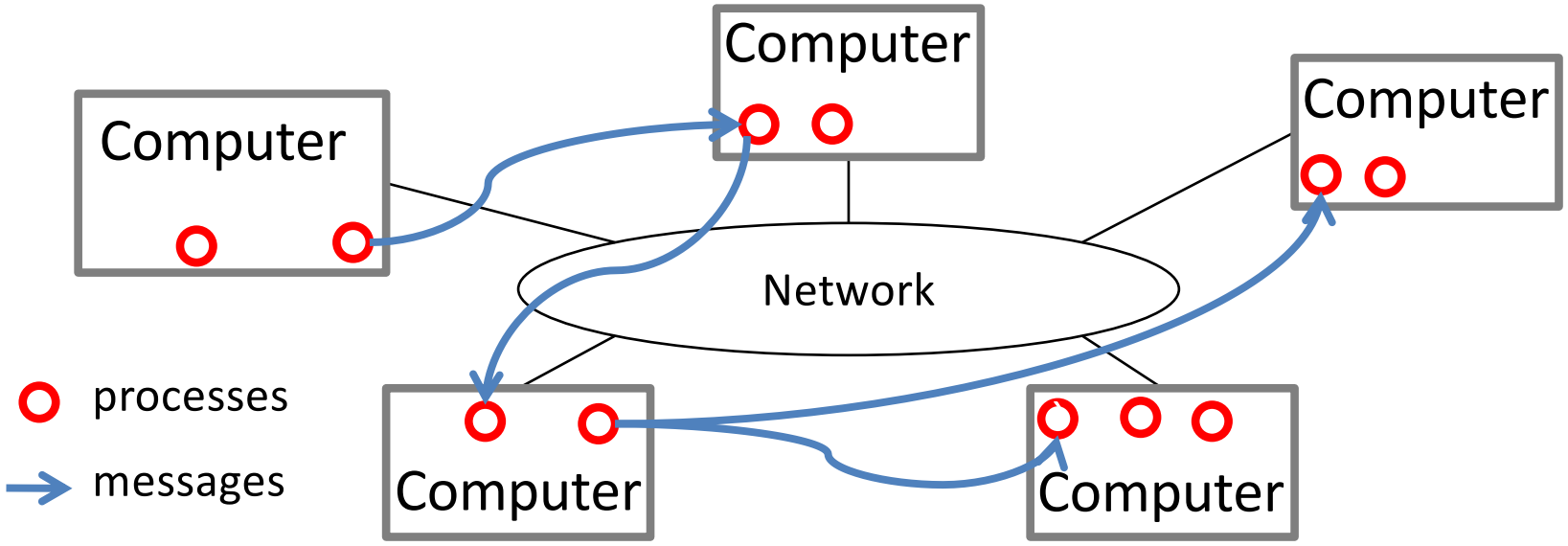


# In a peer-to-peer architecture, are there clients and servers?

A. Yes

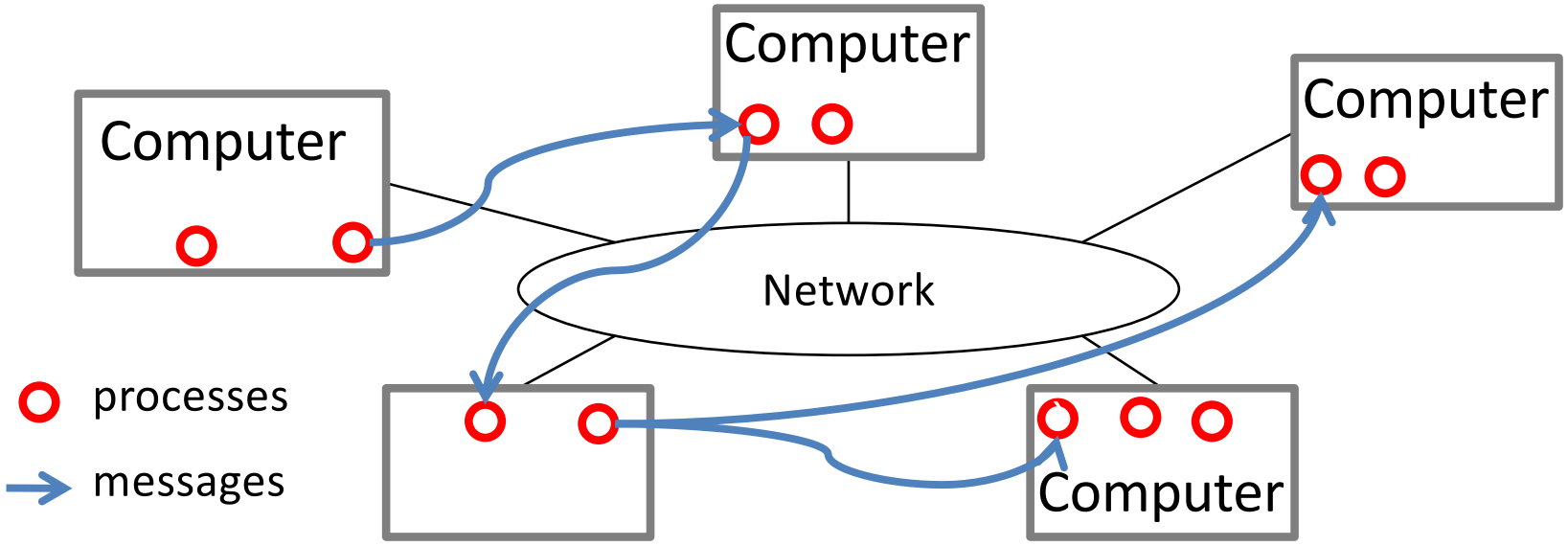
B. No

# Distributed Systems: Advantages




- Speed: parallelism, less contention
- Reliability: redundancy, fault tolerance
- Scalability: incremental growth, economy of scale
- Geographic distribution: low latency, reliability

# Distributed Systems: Disadvantages




- Fundamental problems of decentralized control
  - State uncertainty: no shared memory or clock
  - Action uncertainty: mutually conflicting decisions
- Distributed algorithms are complex

# If one machine can process requests at a rate of X per second, how quickly can two machines process requests?

A. Slower than one machine (<X)
B. The same speed (X)
C. Faster than one machine, but not double (X-2X)
D. Twice as fast (2X)
E. More than twice as fast(>2X)

# On a single system…

- You have a number of components
  - CPU
  - Memory
  - Disk
  - Power supply

- If any of these go wrong, you're (usually) toast.

# On multiple systems…

- New classes of failures (**partial failures**).
  - A link might fail
  - One (of many) processes might fail
  - The network might be partitioned

# On multiple systems…

- New classes of failures (**partial failures**).
  - A link might fail
  - One (of many) processes might fail
  - The network might be partitioned

Introduces major complexity!

If a process sends a message, can it tell the difference between a slow link and a delivery failure?

If a process sends a message, can it tell the difference between a slow link and a delivery failure?

A. Yes

B. No

# What should we do to handle a partial failure? Under what circumstances, or what types of distributed applications?

A. If one process fails or becomes unreachable, switch to a spare.
B. Pause or shut down the application until all connectivity and processes are available.
C. Allow the application to keep running, even if not all processes can communicate.
D. Handle the failure in some other way.

# Desirable Properties

- Consistency
  - Nodes agree on the distributed system's state
- Availability
  - The system is able and willing to process requests
- Partition tolerance
  - The system is robust to network (dis)connectivity

# The CAP Theorem

- **C**onsistency
  - Nodes agree on the distributed system's state
- **A**vailability
  - The system is able and willing to process requests
- **P**artition tolerance
  - The system is robust to network (dis)connectivity

- Choose Two
- "CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare."*

* Brewer, Eric. "CAP twelve years later: How the" rules" have changed." Computer 45.2 (2012): 23-29.

# Event Ordering

- It's very useful if all nodes can agree on the order of events in a distributed system

- For example: Two users trying to update a shared file across two replicas

If two events occur (digitally or in the "real world"), can we always tell which happened first?

A. Yes

B. No

# If two events occur (digitally or in the "real world"), can we always tell which happened first?

A. Yes

B. No

"Relativity of simultaneity"

- Example: observing car crashes
- Exception: causal relationship

# Event Ordering

- It's very useful if all nodes can agree on the order of events in a distributed system
- For example: Two users trying to update a shared file across two replicas
- "Time, Clocks, and the Ordering of Events in a Distributed System" by Leslie Lamport (1978)
  - Establishes causal orderings
  - Cited > 8000 times

# Causal Consistency Example

- Suppose we have the following scenario:
  - Sally posts to Facebook, “Billy is missing!”
  - (Billy is at a friend’s house, sees message, calls mom)
  - Sally posts new message, “False alarm, he’s fine”
  - Sally’s friend James posts, “What a relief!”

# Causal Consistency Example

- NOT causally consistent:
  - Third user, Henry, sees only:
  - Sally posts to Facebook, “Billy is missing!”
  - Sally’s friend James posts, “What a relief!”

# Causal Consistency Example

- Suppose we have the following scenario:
  1. Sally posts to Facebook, “Billy is missing!” (Billy is at a friend’s house, sees message, calls mom)
  2. Sally posts new message, “False alarm, he’s fine”
  3. Sally’s friend James posts, “What a relief!”

- Causally consistent version:
  - Because James had seen Sally’s second post (which caused his response), Henry must also see it prior to seeing James’s.

# Summary

- Client-server vs. peer-to-peer models
- Distributed systems are hard to build!
  - Partial failures
  - Ordering of events
- Take CS 87 for more details!