CS 43: Computer Networks

05: HTTP Concurrency and Performance

September 15, 2025

SWARTHMORE COLLEGE

Slides adapted from Kurose & Ross, Vasanta Chaganti, Kevin Webb

A Blocking system call

A. means the process is put to sleep while the event it is waiting on has not yet occurred
B. its running on the CPU and wasting CPU cycles waiting for the event (send/recv) to occur
C. means this process is blocking every other program from running

A Blocking system call

A. **means the process is put to sleep while the event it is waiting on has not yet occurred**
B. its running on the CPU and wasting CPU cycles waiting for the event (send/recv) to occur
C. means this process is blocking every other program from running

Client-Server communication

- Client:
  - initiates communication
  - must know the address and port of the server
  - active socket
- Server:
  - passively waits for and responds to clients
  - passive socket

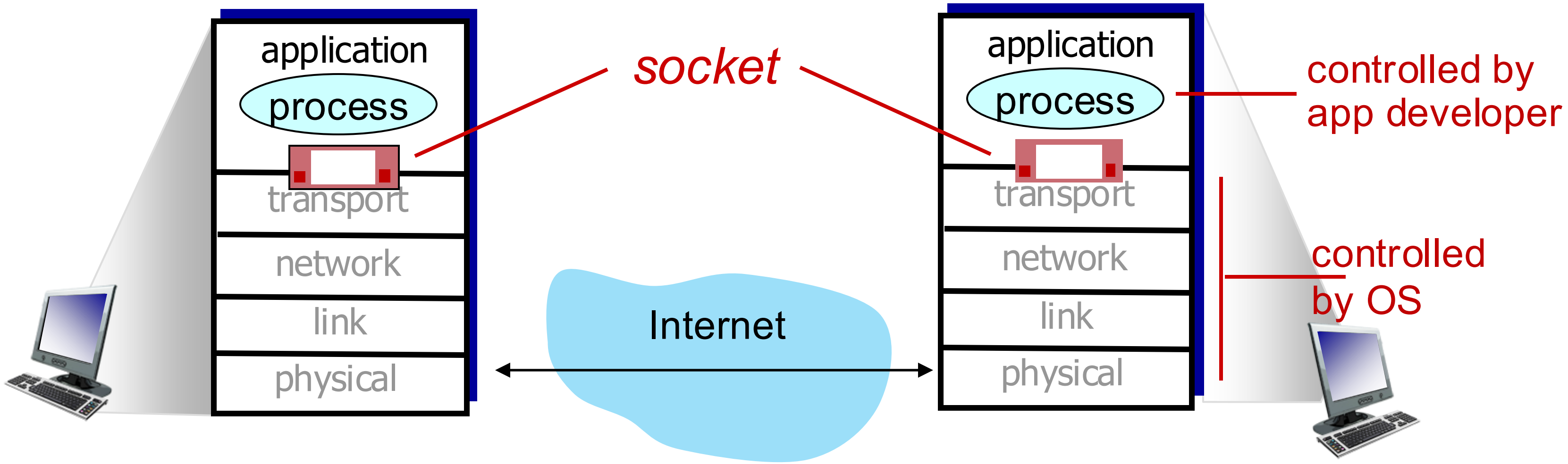
What is a socket?

An abstraction through which an application may send and receive data,

in the same way as a open-file handle or file pointer allows an application to read and write data to storage.

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
  - two sockets involved: one on each side



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!

Create a TCP socket: socket()

```
int socket(int domain, int type, int protocol)
```

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

- domain: communication domain of the socket: generic interface.
- type of socket: reliable vs. best-effort
- end-to-end protocol: TCP for a stream socket -
  - 0: default for specified domain and type.

Create a TCP socket: socket()

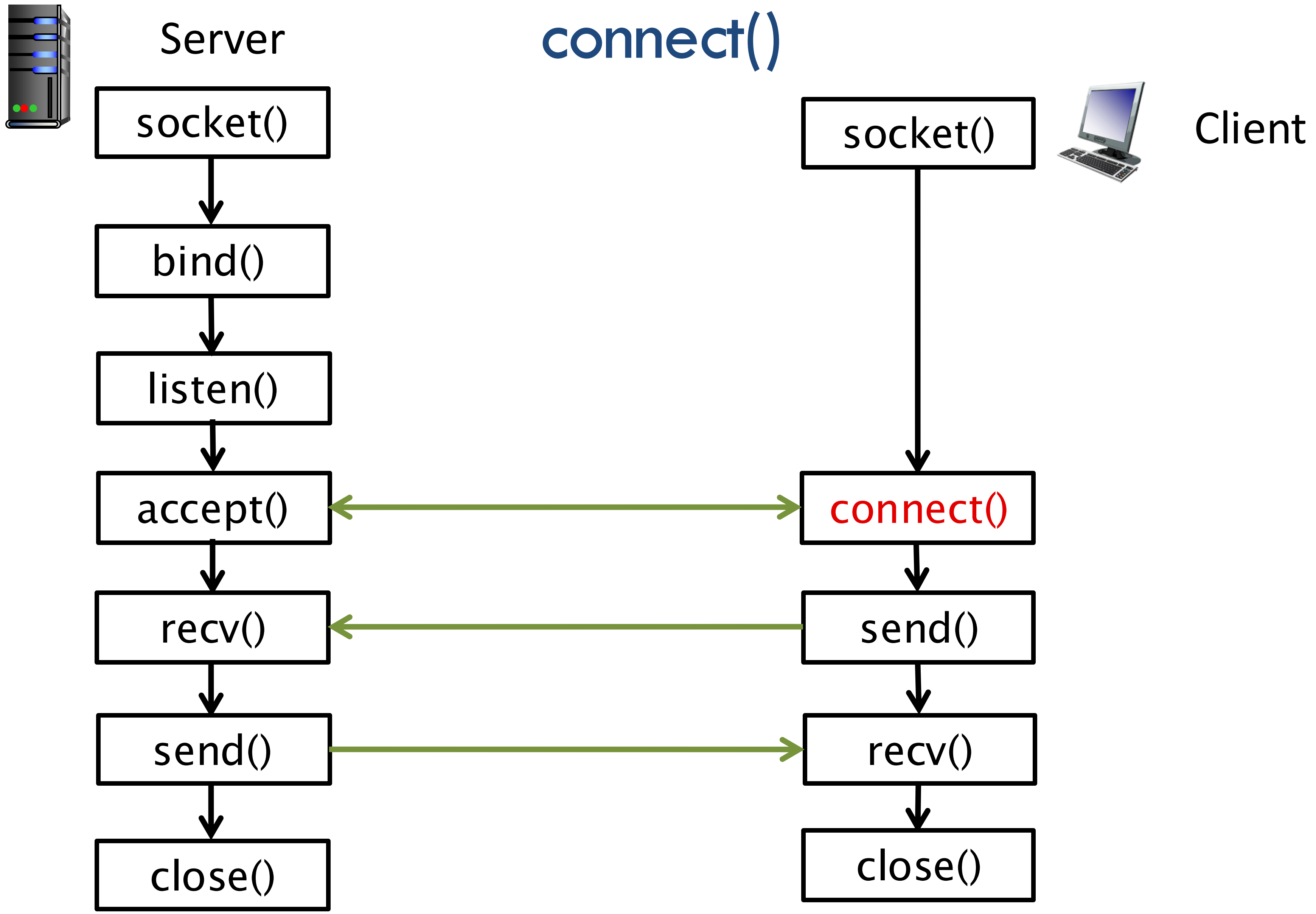
```
int socket(int domain, int type, int protocol)
int sock = socket(AF_INET, SOCK_STREAM, 0);
/* AF_INET: Communicate with IPv4 Address Family (AF),
   SOCK_STREAM: Stream-based protocol
   int sock: returns an integer-valued socket
descriptor or handle
*/
   if (sock < 0) { // If socket() fails, it returns -1
        perror("socket");
        exit(1);
  }
```

Close a socket: close()

```
int close(int socket)
if (close(sock)) {
        perror("close");
        exit(1);
  }
/* int socket: int socket descriptor is passed to close()*/
```

- Close operation similar to closing a file.
- initiate actions to shut down communication
- deallocate resources associated with the socket
- cannot send(), recv() after you close the socket.

connect()


Server
socket()
bind()
listen()
accept()
recv()
send()
close()
Client
socket()
connect()
send()
recv()
close()

connect()

- Before you can communicate, a connection must be established.
- Client Initiates, Server waits.
- Once connect() returns, socket is connected and we can proceed with send(), recv()

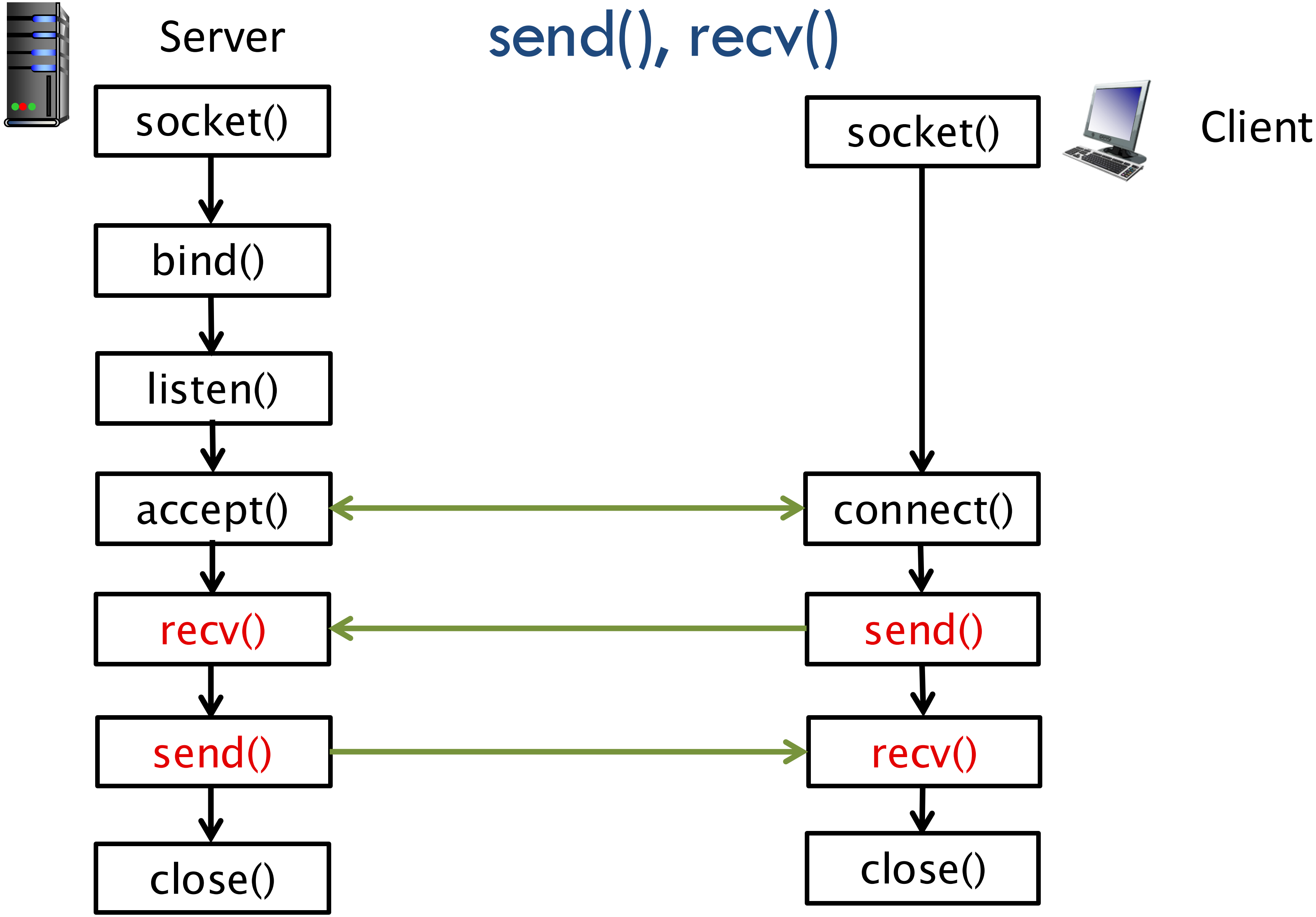
```
int connect(int socket, const struct sockaddr
*foreign_address, socklen_t address_length)
```

connect()

```
int connect(int socket, const struct sockaddr
*foreign_address, socklen_t address_length)
```

```
struct sockaddr_in addr;
int res = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
/* int socket: socket descriptor
    foreignAddress: pointer to sockaddr_in containing Internet
  address, port of server.
     addressLength: length of address structure
*/
```

send(), recv()


Server
socket()
bind()
listen()
accept()
recv()
send()
close()
Client
socket()
connect()
send()
recv()
close()

send(), recv()

Socket is connected when:

- client calls connect()
- connected socket is returned by accept() on server

```
ssize_t send(int socket, const void *msg, msgLength, int flags)
ssize_t recv(int socket, void *rcvBuffer, size_t bufferLength, int flags)

/* int socket: socket descriptor
     return: # bytes sent/received or -1 for failure.
```

send()

send():

```
ssize_t send(int socket, const void *msg, msgLength, int flags)
/* int socket: socket descriptor
     send(): msg: sequence of bytes to be sent
     send(): mesgLength: # bytes to send
```

recv()

recv():

```
ssize_t recv (int socket, void *rcvBuffer, size_t bufferLength, int flags)
int recv_count = recv(sock, buf, 255, 0);

/* int socket: socket descriptor
      void *rcvBuffer: generally a char array
      size_t bufferLength: length of buffer: max # bytes that can be
   received at once.
      flags: setting flag to zero specifies default behavior.
```

Place all send() and recv() calls in a loop, until you are left with no more bytes to send or receive. One call to send()/recv(), irrespective of the buffer does not necessarily mean all your data will be received at once.

Request Method Types ("verbs")

HTTP/1.0 (1996):

- GET:
  - Requests page.
- POST:
  - Uploads user response to a form.
- HEAD:
  - asks server to leave requested object out of response

HTTP/1.1 (1997 & 1999):

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field
- TRACE, OPTIONS, CONNECT, PATCH
- Persistent connections

Uploading form input

GET (in-URL) method:

- uses GET method
- input is uploaded in URL field of request line:

www.somesite.com/animalsearch?monkeys&banana

POST method:

- web page often includes form input
- input is uploaded to server in request entity body

GET vs. POST

GET can be used for idempotent requests

- Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

GET vs. POST

GET can be used for idempotent requests

- Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

Q: How many of the following operations are idempotent?

I. Incrementing a variable
II. Assigning a value to a variable
III. Allocating Memory
IV. Compiling a program

A. None of them
B. One of them
C. Two of them
D. Three of them
E. All of them

GET vs. POST

GET can be used for idempotent requests

- Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

Q: How many of the following operations are idempotent?

I. Incrementing a variable
II. Assigning a value to a variable
III. Allocating Memory
IV. Compiling a program

A. None of them
B. One of them
C. Two of them
D. Three of them
E. All of them

GET vs. POST

GET can be used for idempotent requests.

- Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

GET vs. POST

POST should be when:

- A request changes the state of the server or DB
- Sending a request twice would be harmful: (Some) browsers warn about sending multiple post requests
- Users are inputting non-ASCII characters
- Input may be very large
  - You want to hide how the form works/user input

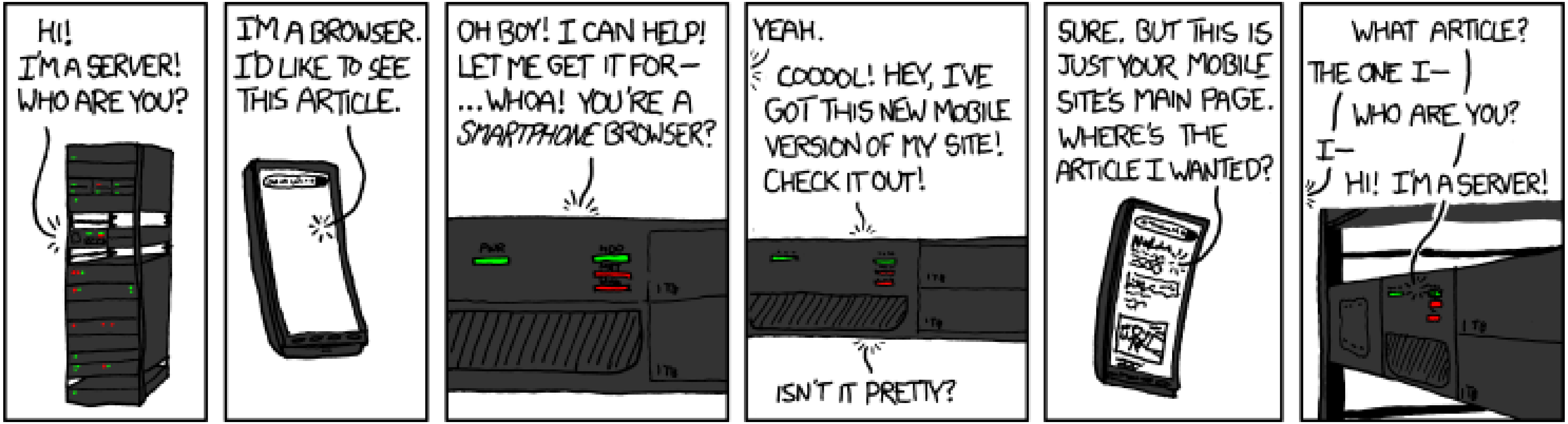
When might you use GET vs. POST?

	GET	POST
A.	Forum post	Search terms, Pizza order
B.	Search terms, Pizza order	Forum post
C.	Search terms	Forum post, Pizza order
D.	Forum post, Search terms, Pizza Order	
E.		Forum post, Search terms, Pizza Order

When might you use GET vs. POST?

	GET	POST
A.	Forum post	Search terms, Pizza order
B.	Search terms, Pizza order	Forum post
C.	**Search terms**	**Forum post, Pizza order**
D.	Forum post, Search terms, Pizza Order	
E.		Forum post, Search terms, Pizza Order

State(less)



(XKCD #869, “Server Attention Span”)

HTTP State

Does the HTTP protocol, allow for a server to keep track of every client?

A. Yes, it’s required to
B. No, it would not scale
C. That’s against privacy rules!
D. Something else

State(less)

- Original web: simple document retrieval
- Maintain State? Server is not required to keep state between connections

  ...often it might want to though
- Authentication: Client is not required to identify itself
  - server might refuse to talk otherwise though

User-server state: cookies

What cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

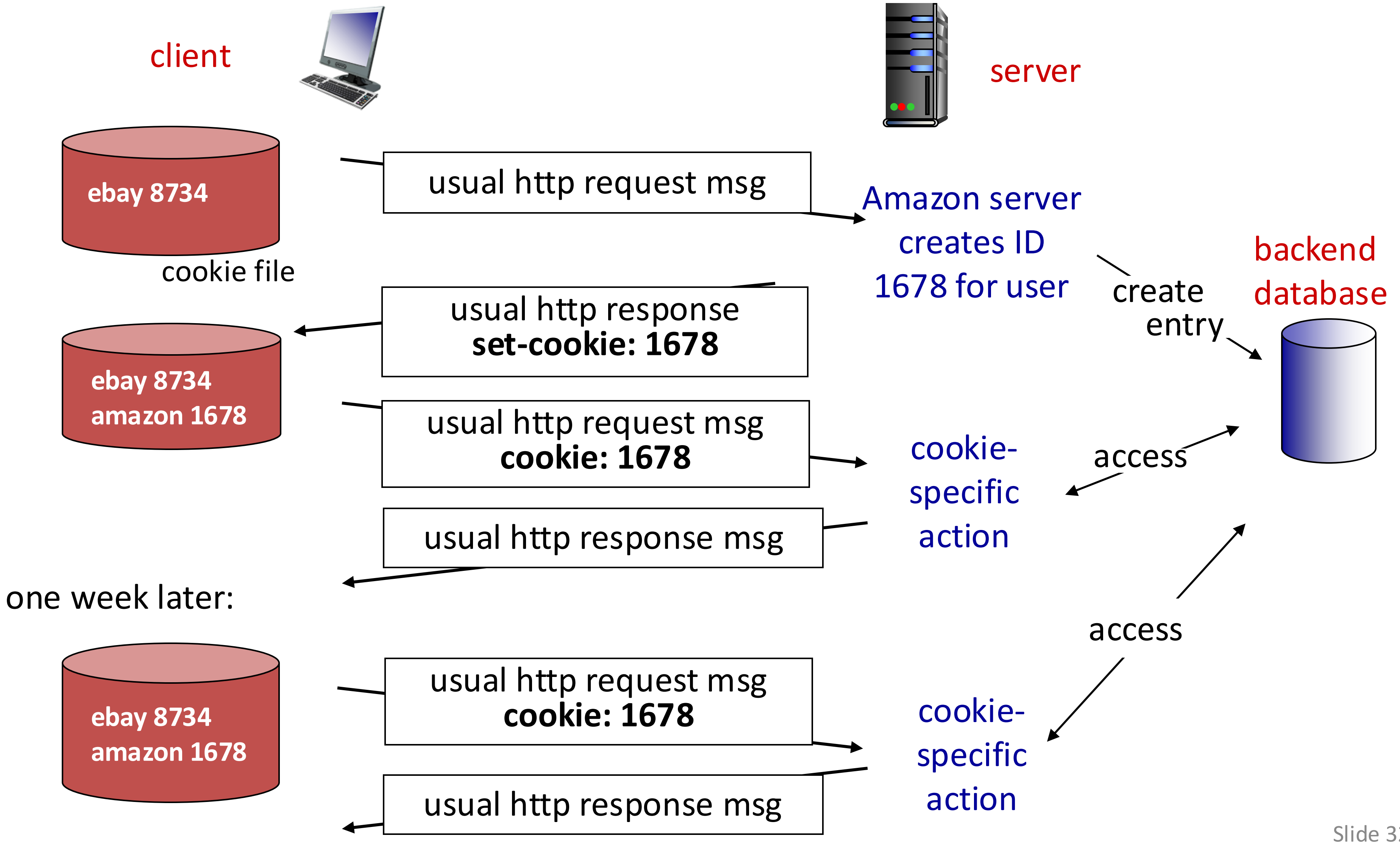
How to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

What Are Cookies Used For?

- Authentication
  - The cookie proves to the website that the client previously authenticated correctly
- Personalization
  - Helps the website recognize the user from a previous visit
- Tracking
  - Follow the user from site to site;
  - Read about iPads on CNN and see ads on Amazon 😱
  - How can an advertiser (A) know what you did on another site (S)?

Cookies: keeping "state" (cont.)



User-server state: cookies

Many web sites use cookies

Four components:

1) cookie header line of HTTP response message

2) cookie header line in next HTTP request message

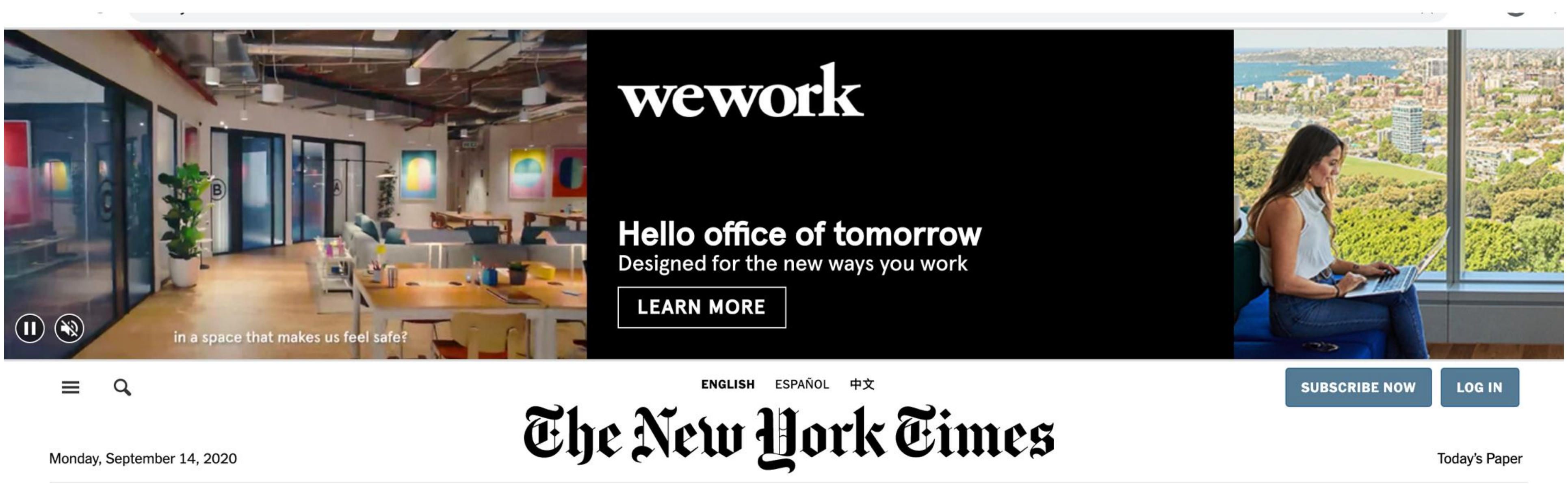
3) cookie file kept on user's host, managed by user's browser

4) back-end database at Web site

Cookies and Privacy

Cookies permit sites to learn a lot about you

- supply name and e-mail to sites (and more!)
- third-party cookies (ad networks) follow you across multiple sites.



Login Session

GET /loginform HTTP/1.1
cookies: []

HTTP/1.1 200 OK
cookies: []
<html><form>…</form></html>

POST /login HTTP/1.1
cookies: []
username: chaganti
password: swarthmore

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Login Success</h1></html>

GET /account HTTP/1.1
cookies: [session: e82a7b92]

GET /img/user.jpg HTTP/1.1

HTTP connections

Non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects requires multiple connections

Persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

object: image, script, stylesheet, etc.

Non-persistent HTTP

suppose user enters URL: contains references to 10 jpeg images

1a. HTTP client initiates TCP connection to HTTP server

1b. HTTP server "accepts" connection, notifying client

2. HTTP client sends HTTP request message: URL

3. HTTP server:
   - receives request
   - forms response message containing requested index.html
   - sends message

4. HTTP server closes TCP connection.

5. HTTP client receives response:
   - index.html
   - finds 10 referenced jpeg objects

time

6. Steps 1-5 repeated for each of 10 jpeg objects!!

Pseudocode Example

non-persistent HTTP

```
for object on web page:
    connect to server
    request object
    receive object
    close connection
```

persistent HTTP

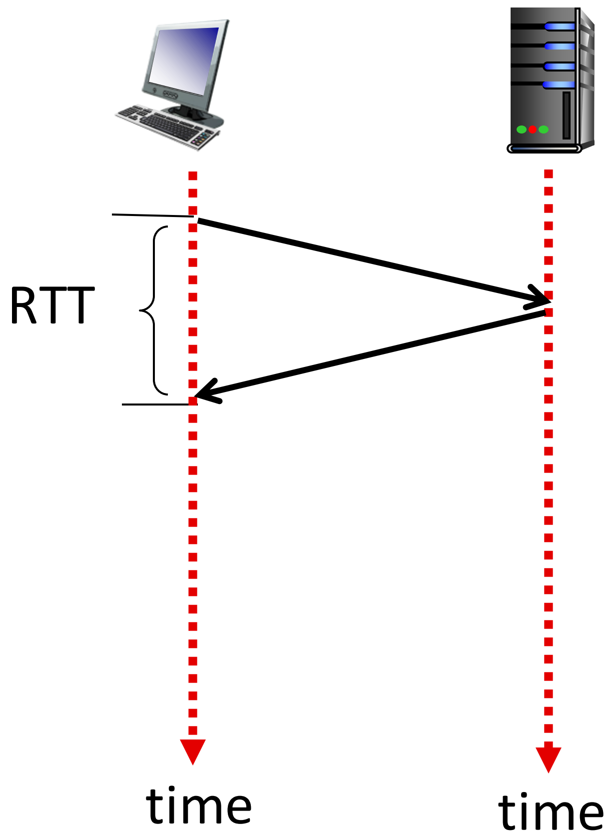
```
connect to server
for object on web page:
    request object
    receive object
close connection
```

Round Trip Time

Round Trip Time (RTT):

- time for a small packet to travel from client to server and response to come back.

- Connection establishment (via TCP) requires one RTT.



Non-Persistent HTTP Connections can download a website with several objects in…

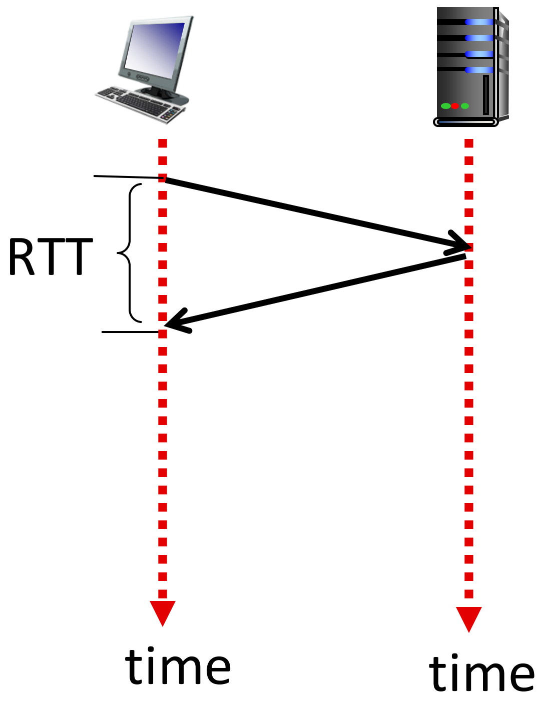
A. One RTT + (File transfer time per object)

B. (One RTT + File transfer time) per object

C. Two RTTs

D. Two RTTs + (File transfer time per object)

E. (Two RTTS + File transfer time) per object



Non-persistent HTTP: response time

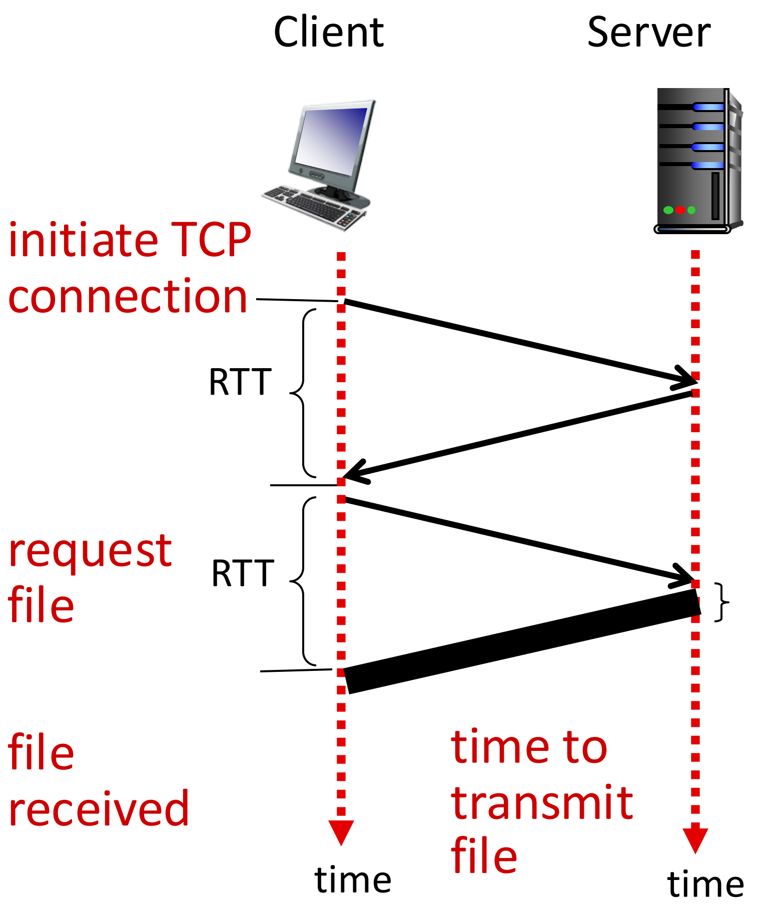
Round Trip Time (RTT): time for a small packet to travel from client to server and back

HTTP response time:

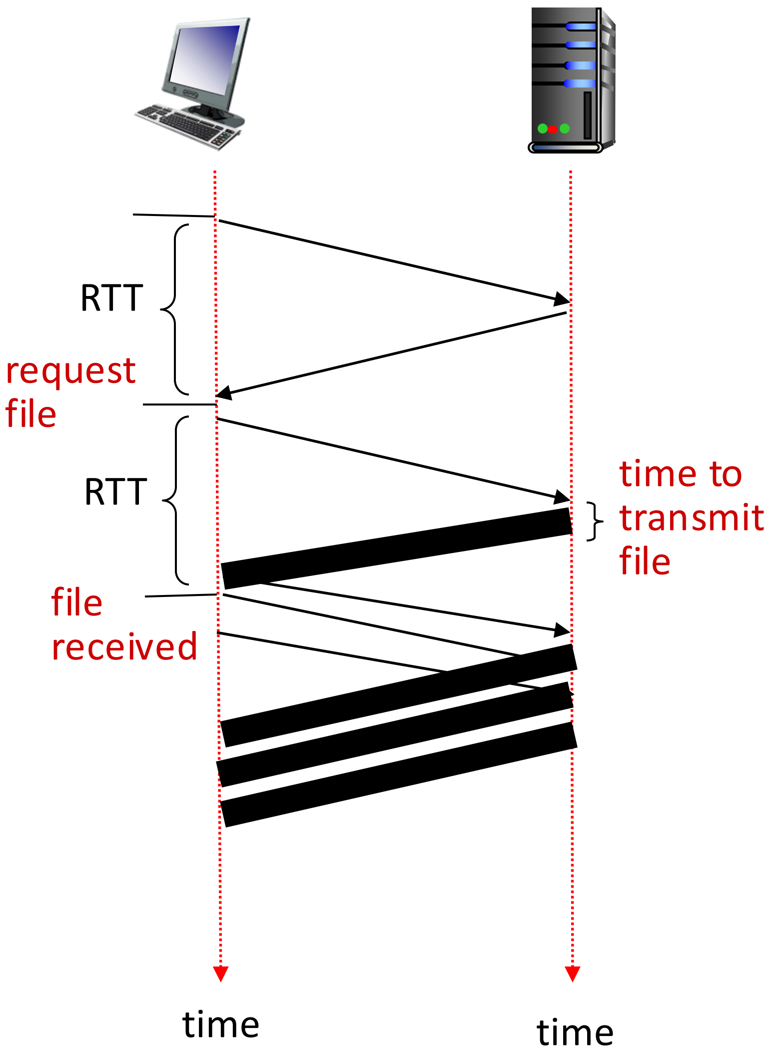
- 1-RTT to initiate TCP connection
- 1-RTT for HTTP request + first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2-RTT+ file transmission time

  For each object



Persistent Connection



Persistent HTTP

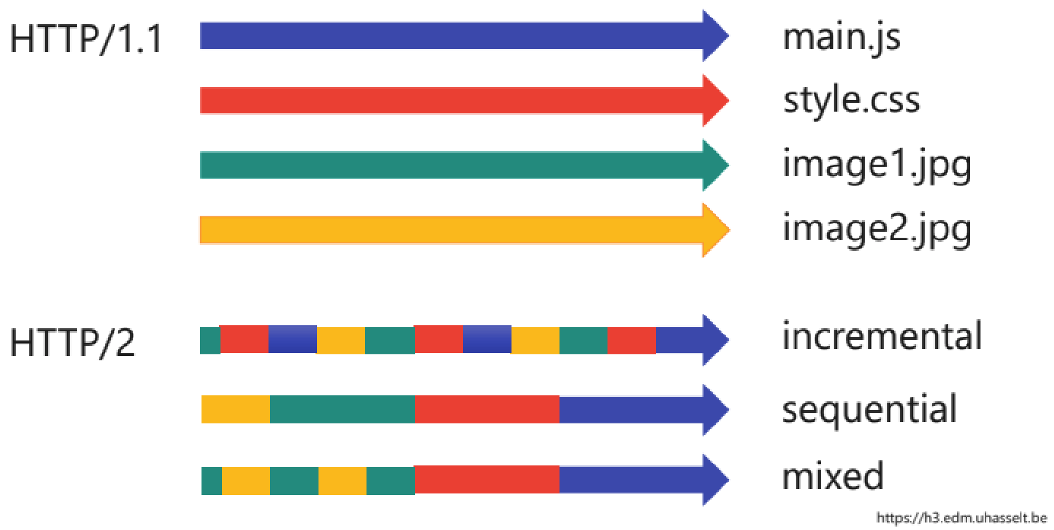
Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for each TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

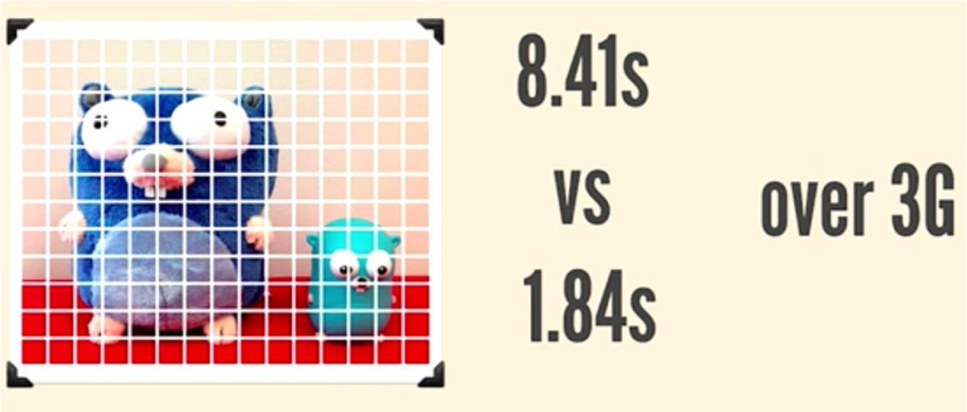
Persistent HTTP:

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP 1.x vs HTTP 2.0 vs. HTTP 3.0



- SPDY: protocol to speed up the web: Basis for HTTP 2.0
- Request pipelining
- Compress header metadata

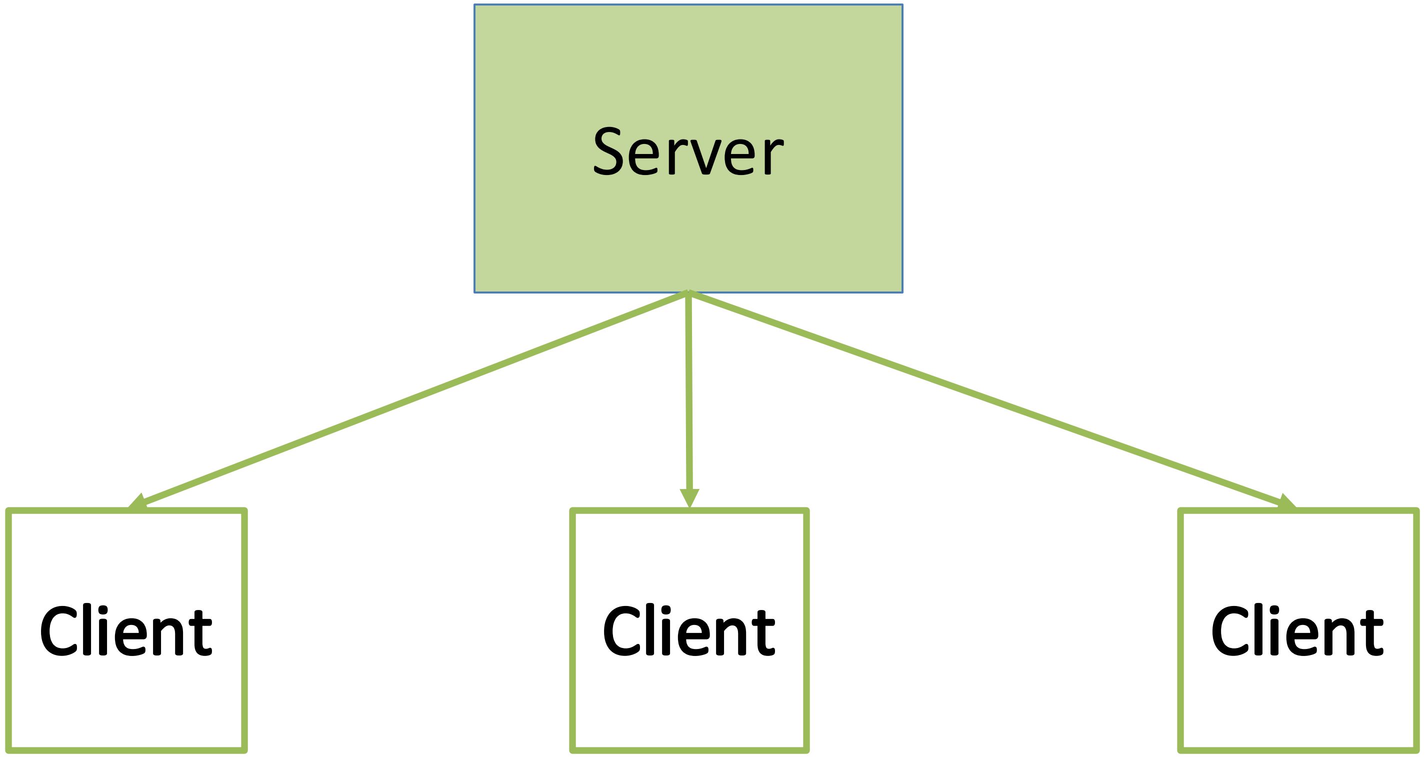


Learn more: https://http2.github.io/

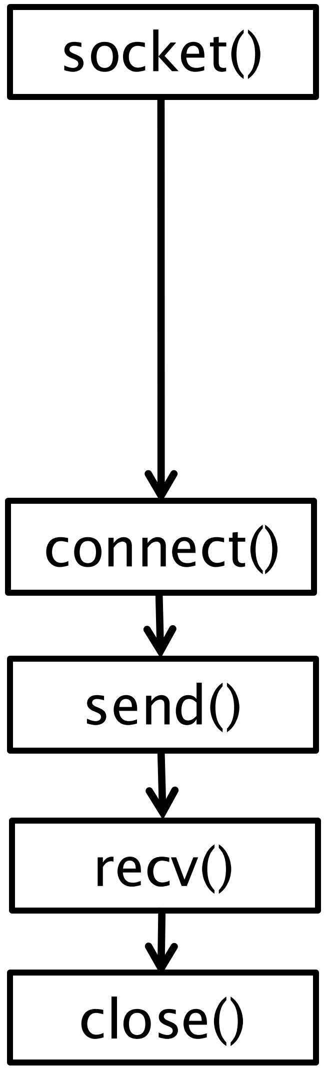
HTTP/2 101 Chrome Dev Summit 2015, Robin Marx – ”Fixing HTTP/2 and Preparing for HTTP/3 over QUIC

Concurrency

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



TCP Socket Procedures: for a Web Client



socket: create a new communication endpoint

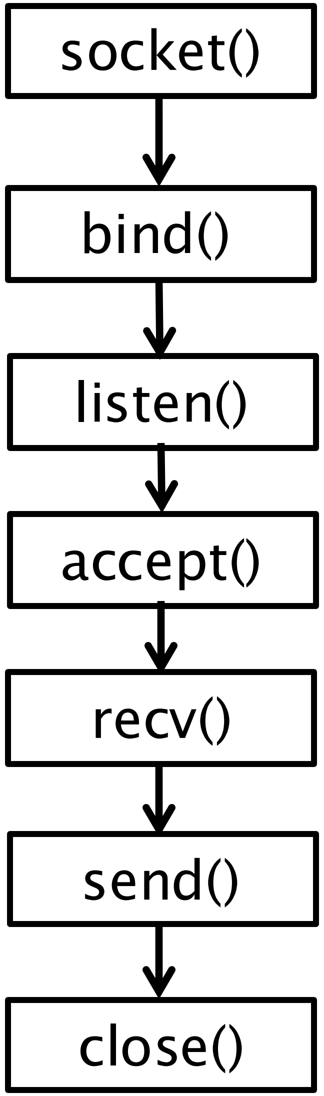
connect: actively attempt to establish a connection

send: receive some data over a connection

receive: send some data over a connection

close: release the connection

TCP socket procedures for a web server



socket: create a new communication endpoint

bind: attach a local address to a socket

listen: announce willingness to accept connections

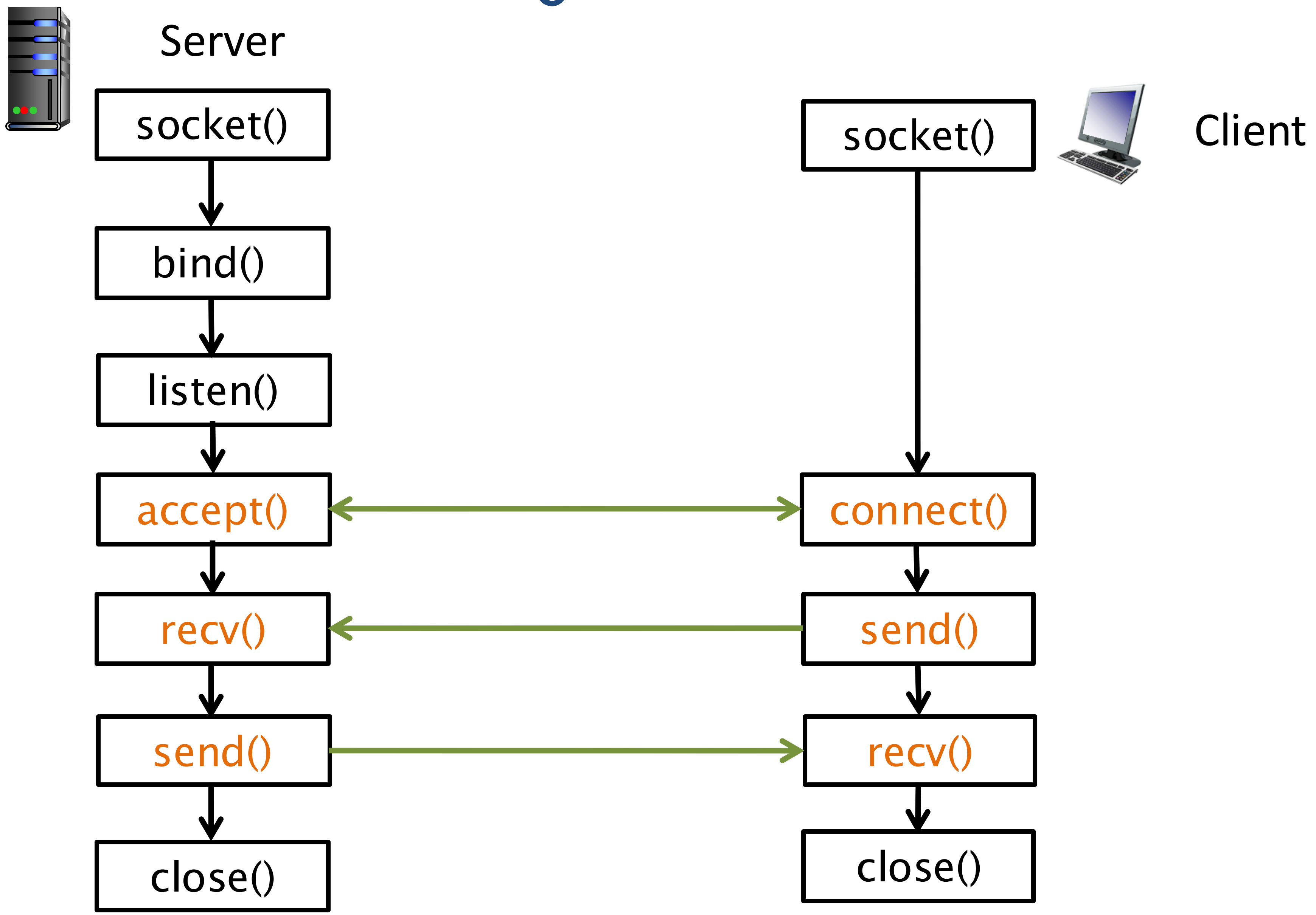
accept: block caller until a connection request arrives

recv: receive some data over a connection

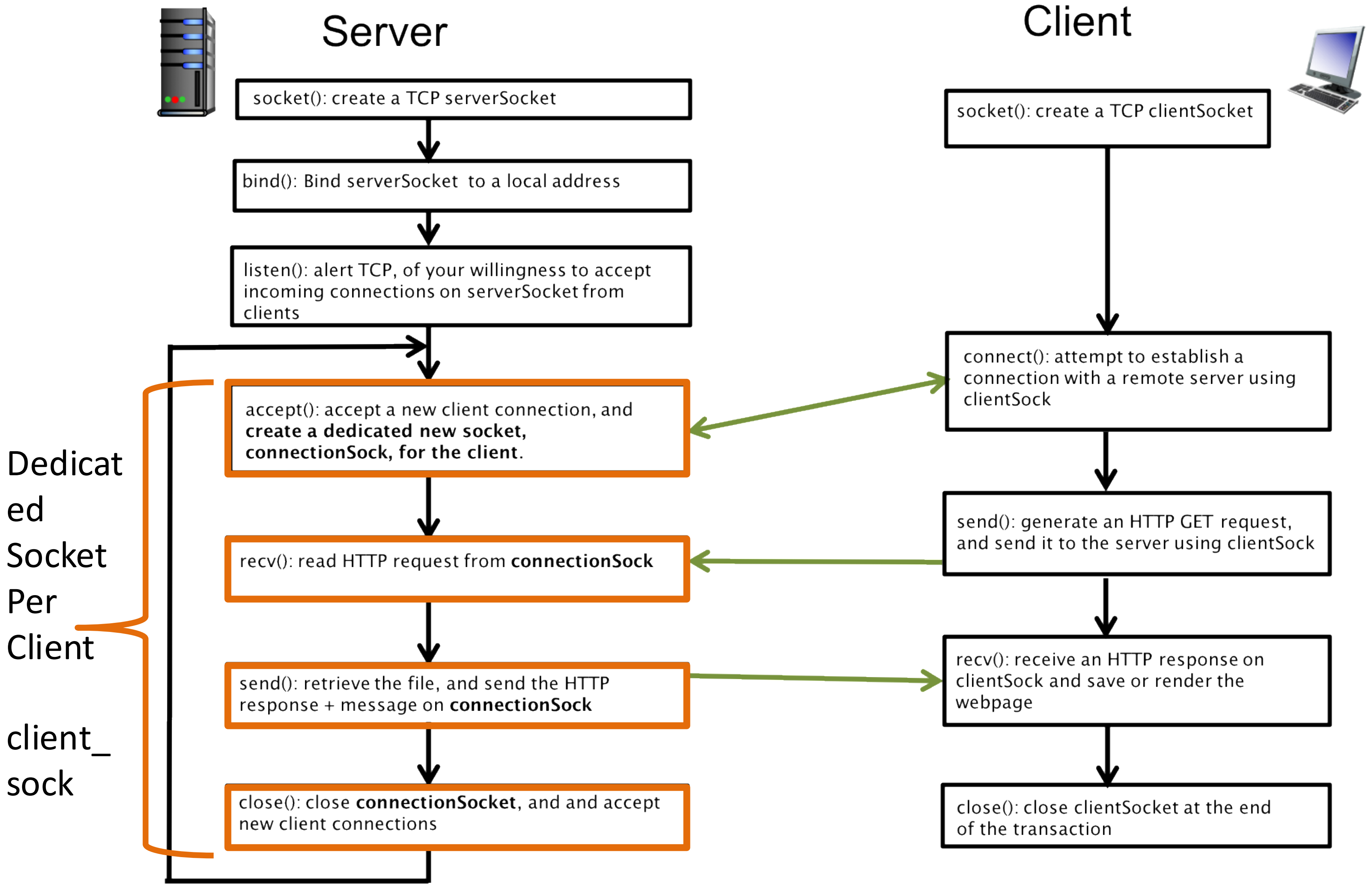
send: send some data over a connection

close: release the connection

Running a Web Server over TCP

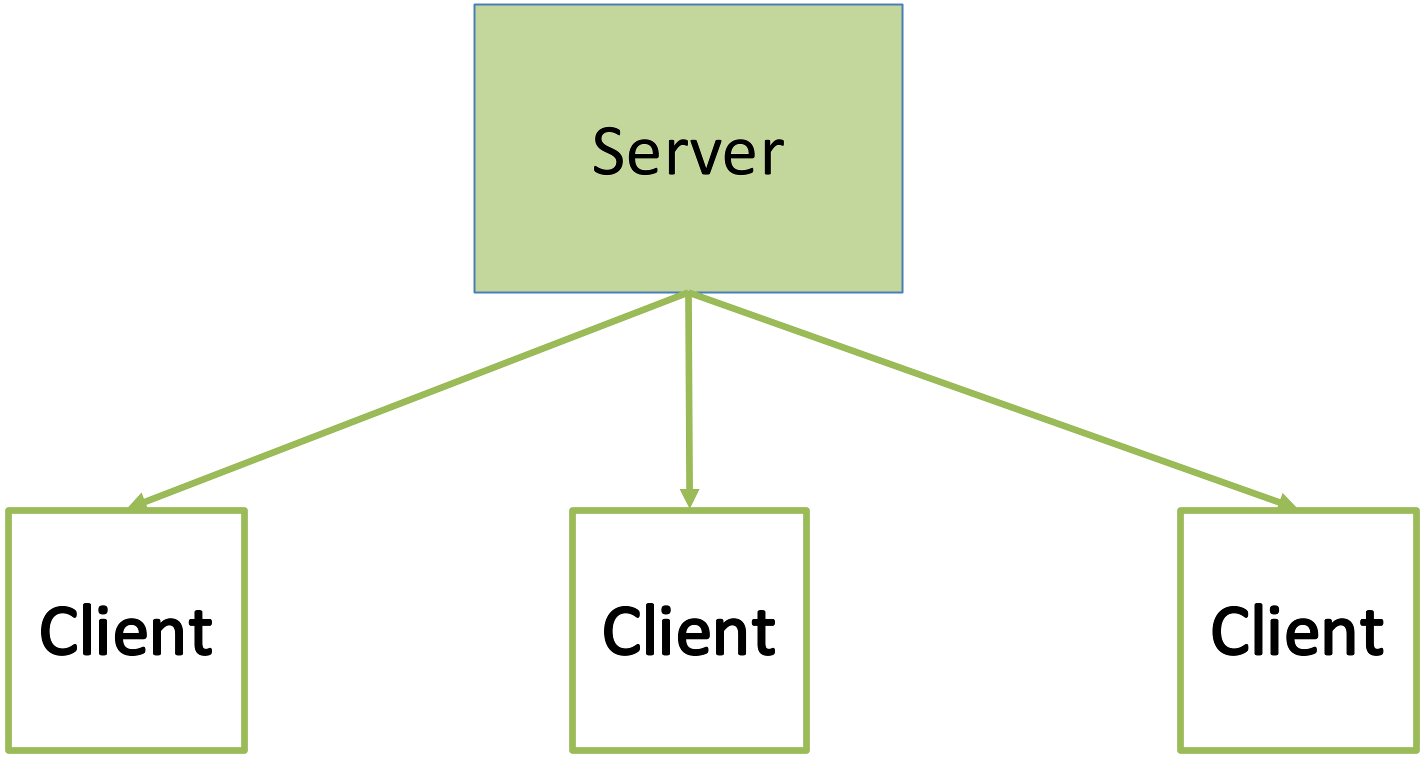


Running a Web Server



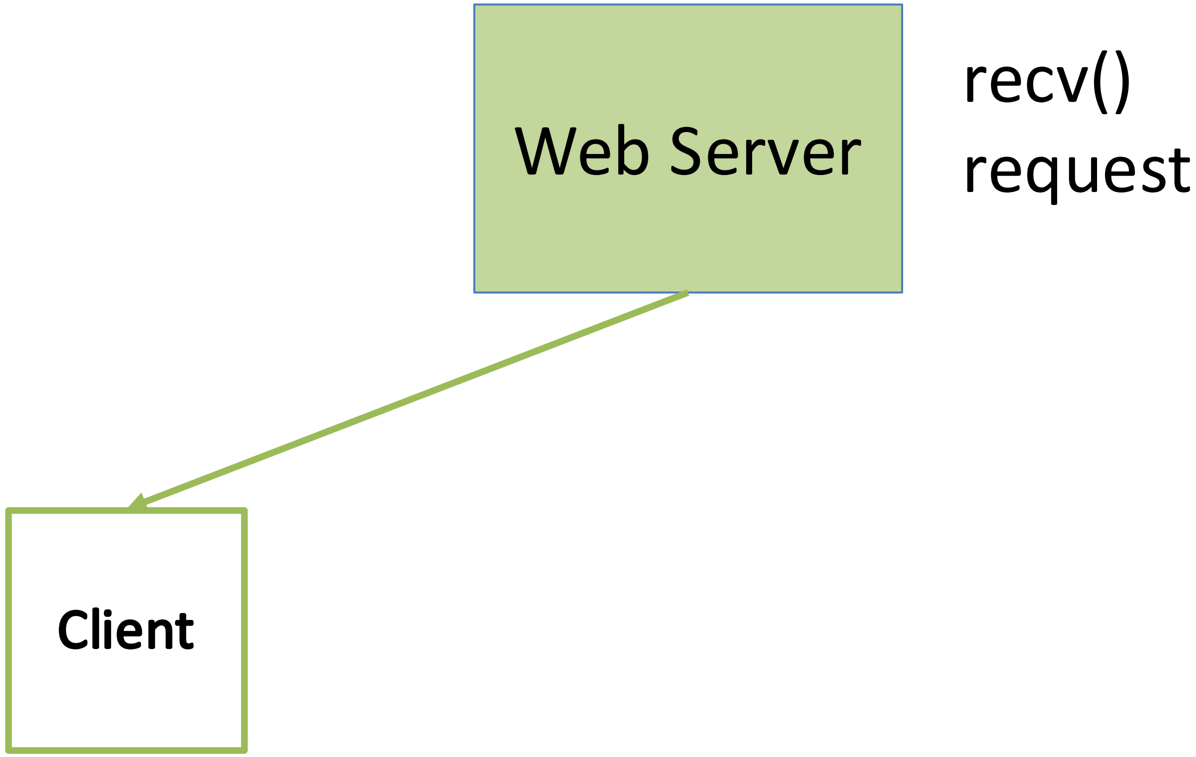
Concurrency

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



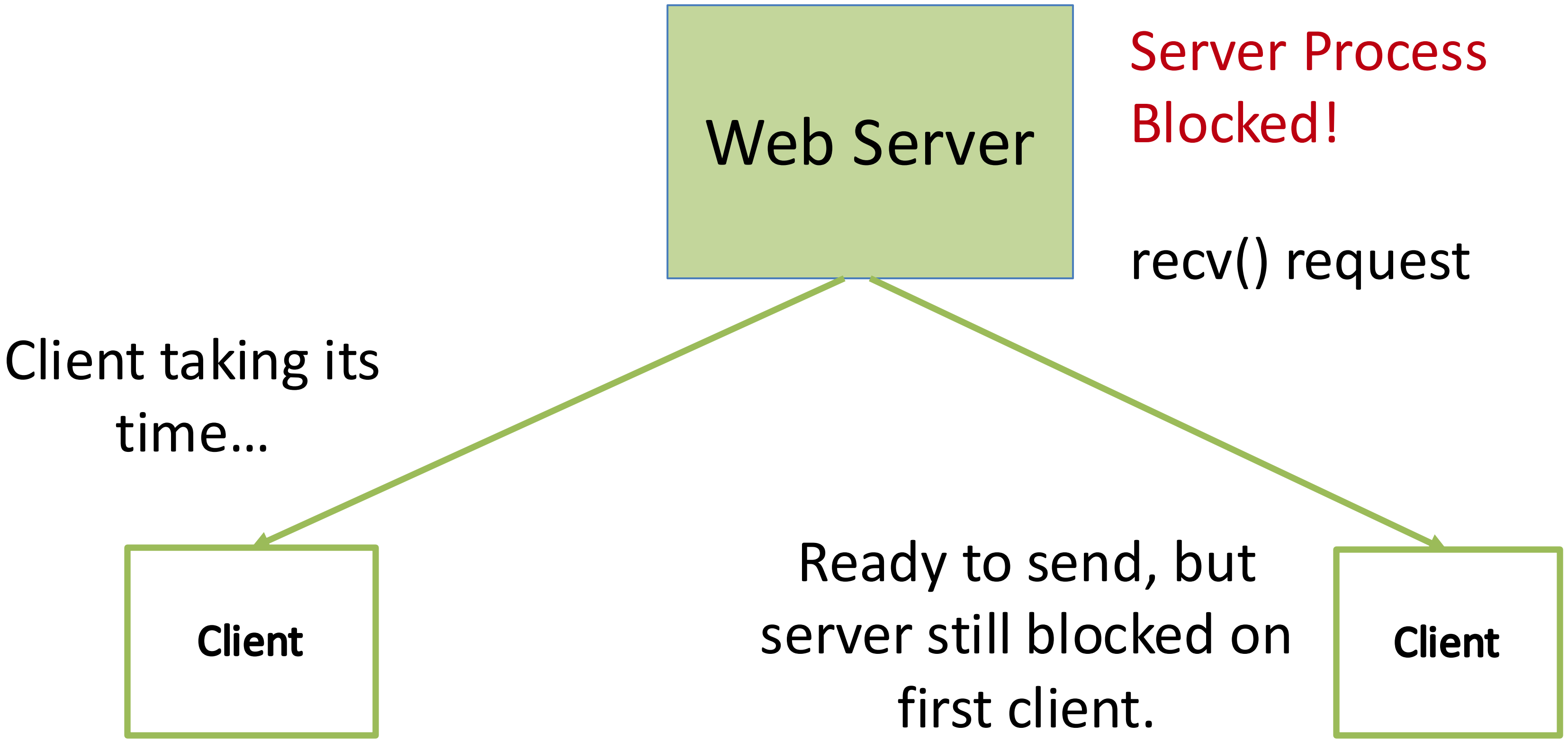
Without Concurrency

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



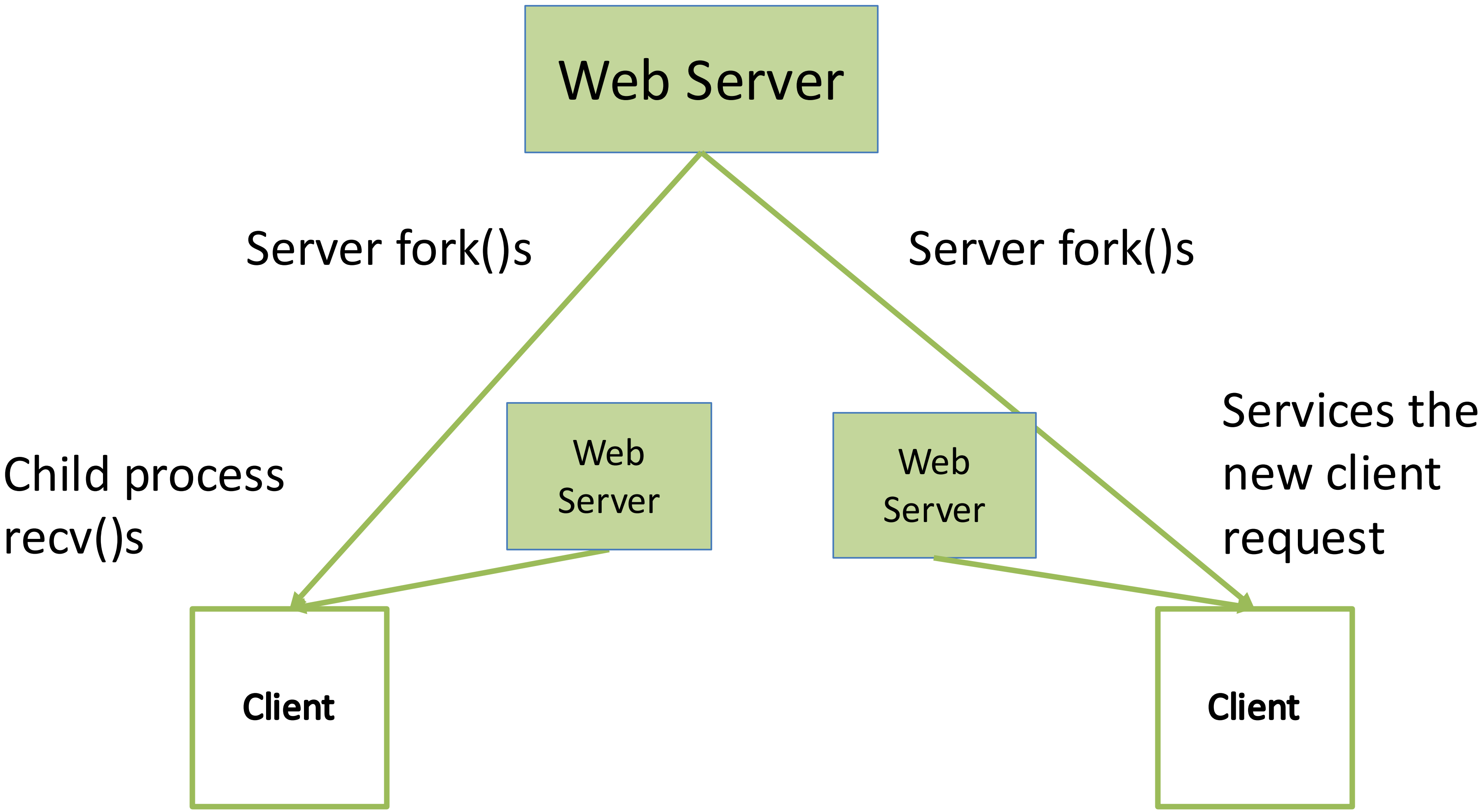
Without Concurrency

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



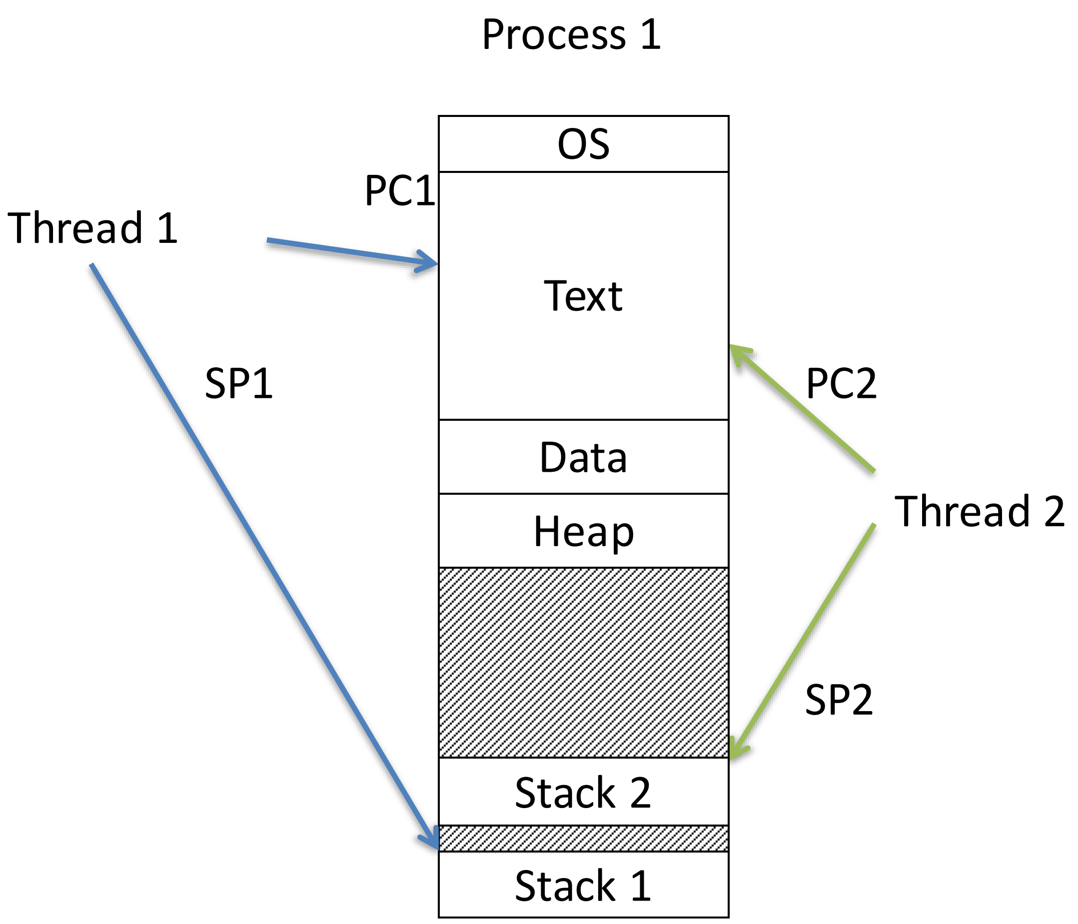
If only we could handle these connections separately…

Multiple processes

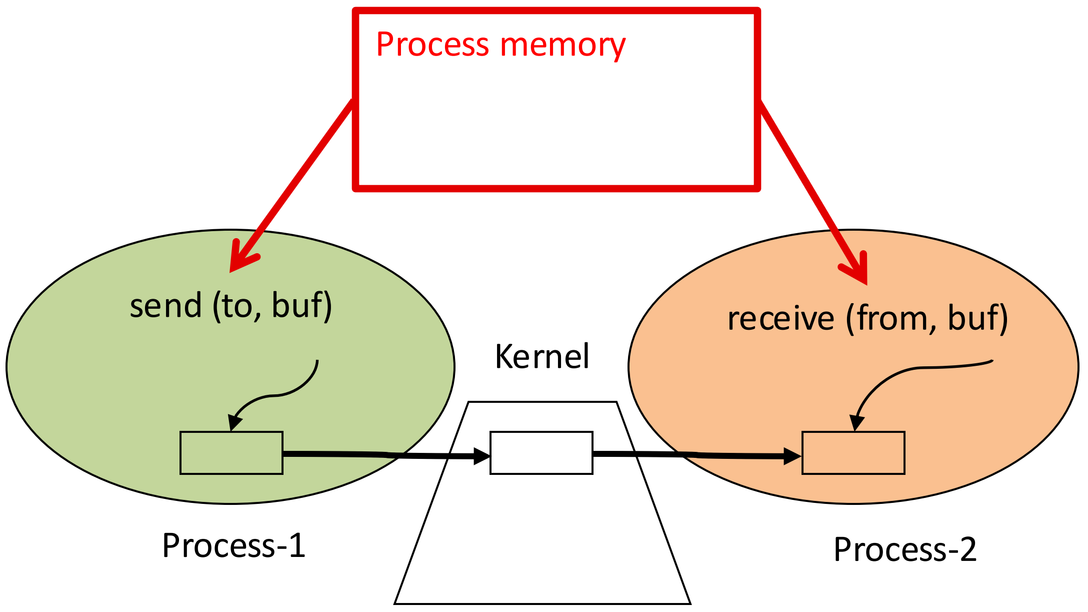


Concurrent Web-servers with multiple threads/processes

- Threads (shared memory)



- Message Passing (locally)



Processes/Threads vs. Parent

Spawned Process

- Inherits descriptor table
- Does not share memory
  - New memory address space
- Scheduled independently
  - Separate execution context
  - Can block independently

Spawned Thread

- Shares descriptor table
- Shares memory
  - Uses parent's address space
- Scheduled independently
  - Separate execution context
  - Can block independently

Processes/Threads vs. Parent
(More details in an OS class…)

Spawned Process

- Inherits descriptor table
- Does not share memory
  - New memory address space
- Scheduled independently
  - Separate execution context
  - Can block independently

Spawned Thread

- Shares descriptor table
- Shares memory
  - Uses parent's address space
- Scheduled independently
  - Separate execution context
  - Can block independently

Often, we don't need the extra isolation of a separate address space. Faster to skip creating it and share with parent – threading.

Threads & Sharing

- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete
- Local variables are not shared
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack

Which benefit of threads most critical in the context of running a web server?

A. Modular code/separation of concerns.

B. Multiple CPU/core parallelism.

C. I/O overlapping.

D. Some other benefit.

Both processes and threads:

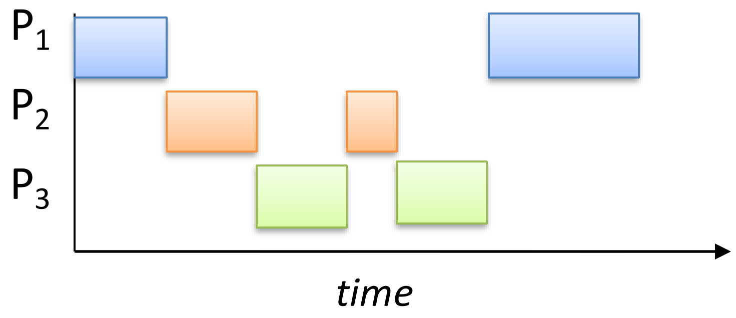
Several benefits

- Modularizes code: one piece accepts connections, another services them
- Each can be scheduled on a separate CPU
- Blocking I/O can be overlapped

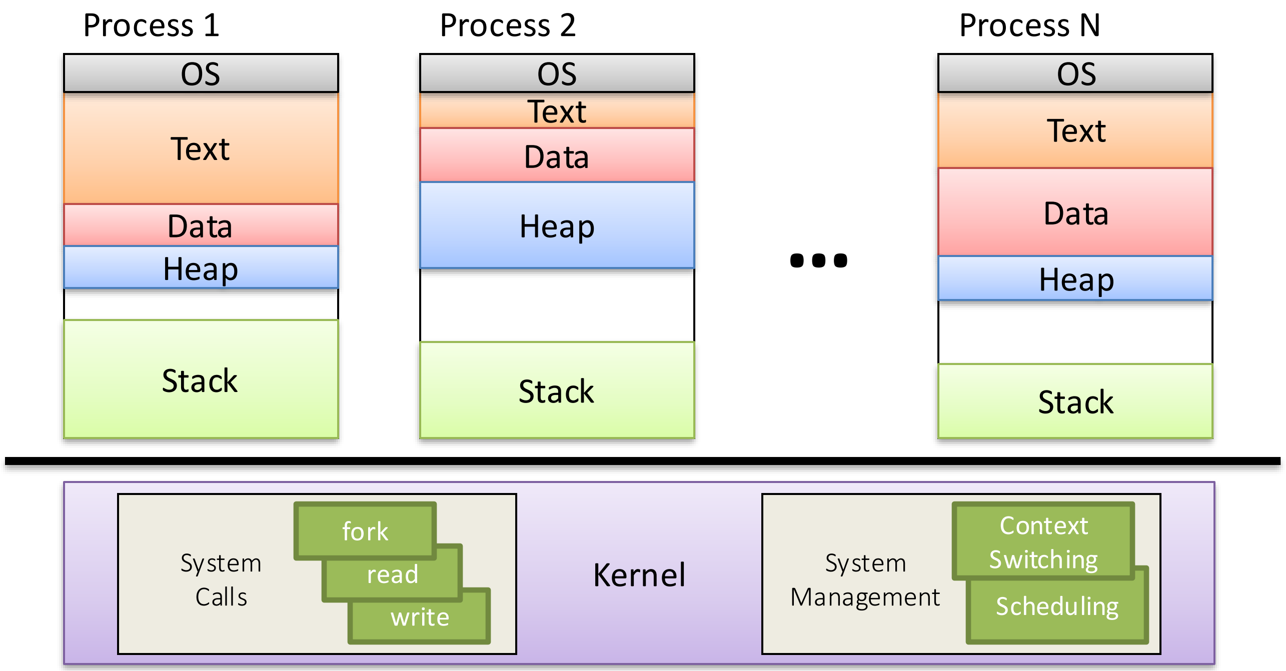
Both processes and threads

Still not maximum efficiency…

- Creating/destroying threads takes time
- Requires memory to store thread execution state
- Lots of context switching overhead



CPU: Time
Single core



Context Switching

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

Event Driven I/O processing!

- Permanently for socket flag O_NONBLOCK
- With O_NONBLOCK set on a socket: No operations will block!

Non-blocking I/O

- With O_NONBLOCK set on a socket
  - No operations will block!

- On recv(), if socket buffer is empty:
  - returns -1

- On send(), if socket buffer is full:
  - returns -1

Will this work?

A. Yes, this will work efficiently.
B. Yes but this will execute too slowly.
C. Yes but 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
```

Will this work?

A. Yes, this will work efficiently.
B. Yes but this will execute too slowly.
C. Yes but 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
```

Non-blocking I/O

- With O_NONBLOCK set on a socket
  - No operations will block!
- On recv(), if socket buffer is empty:
  - returns -1
- On send(), if socket buffer is full:
  - returns -1

So... keep checking send and recv until they return something – waste of CPU cycles?

Event-based concurrency: select()

- Create set of file/socket descriptors we want to send and recv
- Tell the O.S to block the process until at least one of those is ready for us to use.
- The OS worries about selecting which one(s).

Event-based concurrency: select()

Rather than checking over and over, let the OS tell us when data can be read/written

```
client_sockets[10];
FD_SET(client_sockets) //ask OS to watch all client sockets and select those that are
select(client_sockets)      are ready to recv() or send() data
for every client in client_socket:
        FD_ISSET(client, read) //return true if this client socket has any data to be received
        FD_ISSET(client, write) //return true if this client socket has any data to be sent
```

- ✓ OS worries about selecting which sockets (s) are ready.
- ✓ Process blocks if no socket is read to send or receive data.

Event-based concurrency: advantages

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