CS 43: Computer Networks

04: HTTP: Methods, Cookies and Performance

September 17, 2020

SWARTHMORE COLLEGE

Last class

- End-to-end argument
- Five-layer protocol stack
  - Protocols at each layer
- Example HTTP Request

Today

- HTTP
  - GET vs. POST
  - response messages
  - Persistence vs. Non-persistence
- HTTP Performance and Cookies
- Server-side Socket Programming

Last class: 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)

HTTP request message

request line
(GET, POST,
HEAD, etc. commands)

carriage return character

line-feed character

```
GET /index.html HTTP/1.1\r\n
Host: web.cs.swarthmore.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

optional
header
lines

carriage return,
line feed

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:

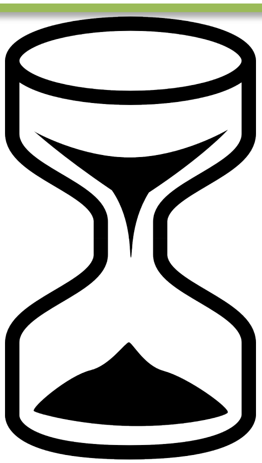
- web page often includes form input
- input is uploaded to server in request entity body

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

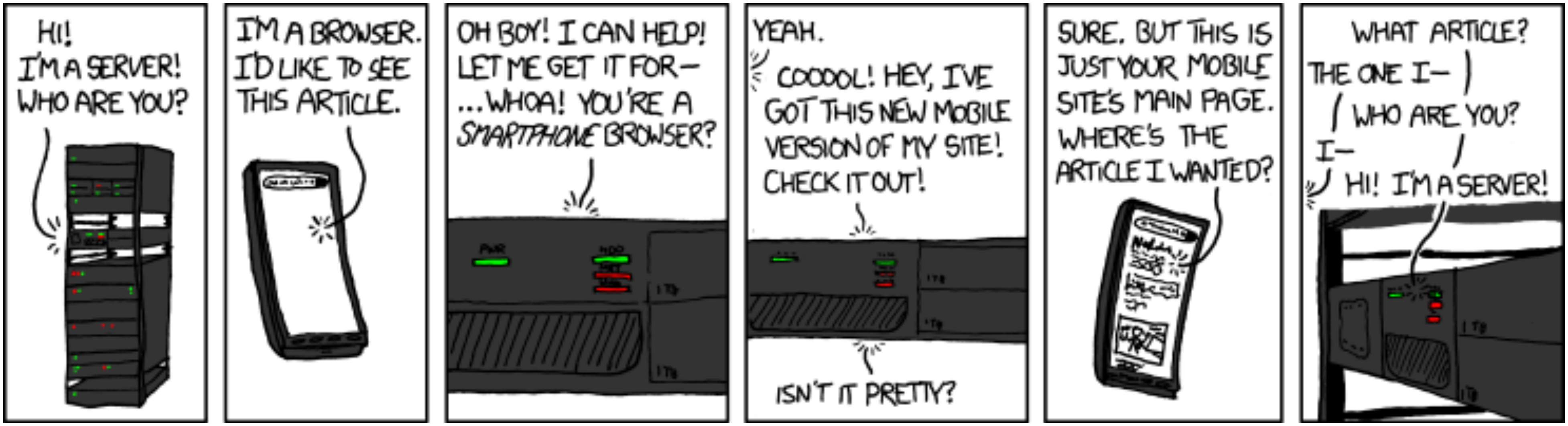
When might you use GET vs. POST?

	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

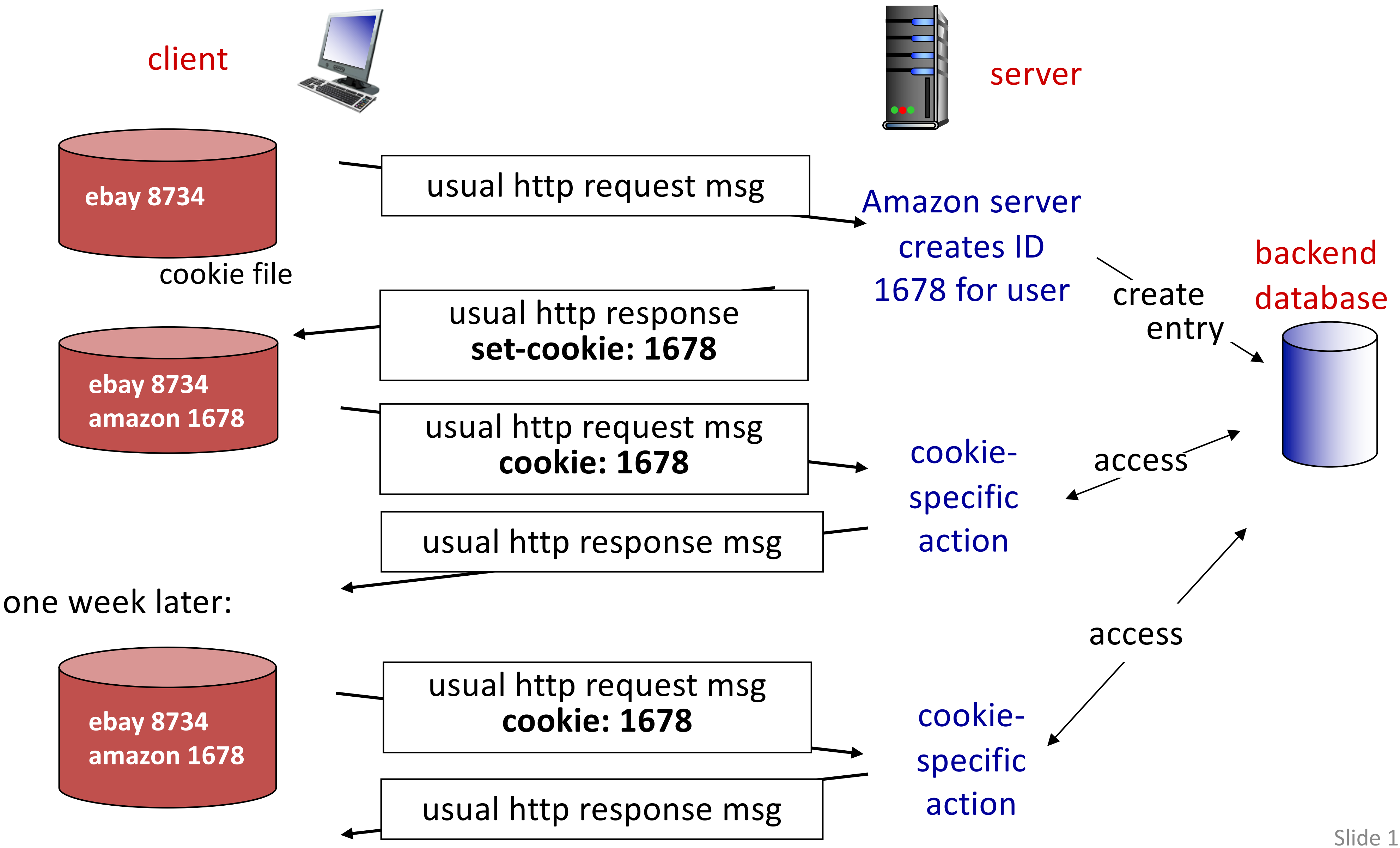
What cookies can be used for:

- 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

Cookies: keeping “state” (cont.)



client

server

ebay 8734

cookie file

usual http request msg

Amazon server creates ID 1678 for user

create entry

backend database

usual http response
set-cookie: 1678

ebay 8734
amazon 1678

usual http request msg
cookie: 1678

cookie-specific action

access

usual http response msg

one week later:

access

ebay 8734
amazon 1678

usual http request msg
cookie: 1678

cookie-specific action

usual http response msg

User-server state: cookies

Many web sites use cookies

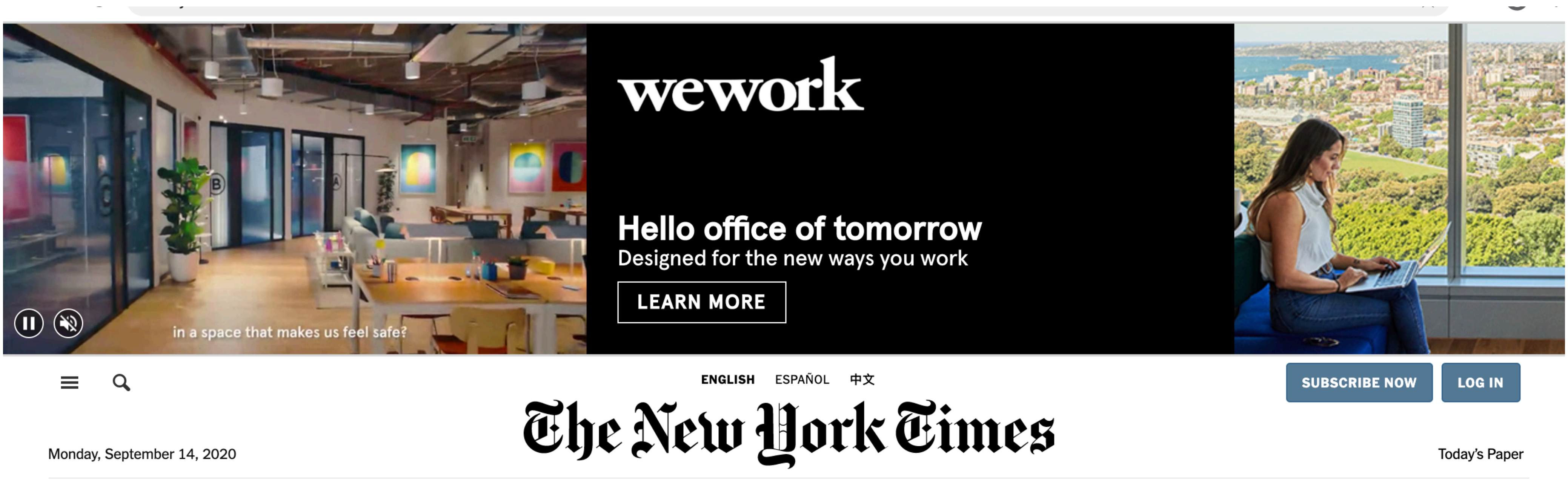
Four components:

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

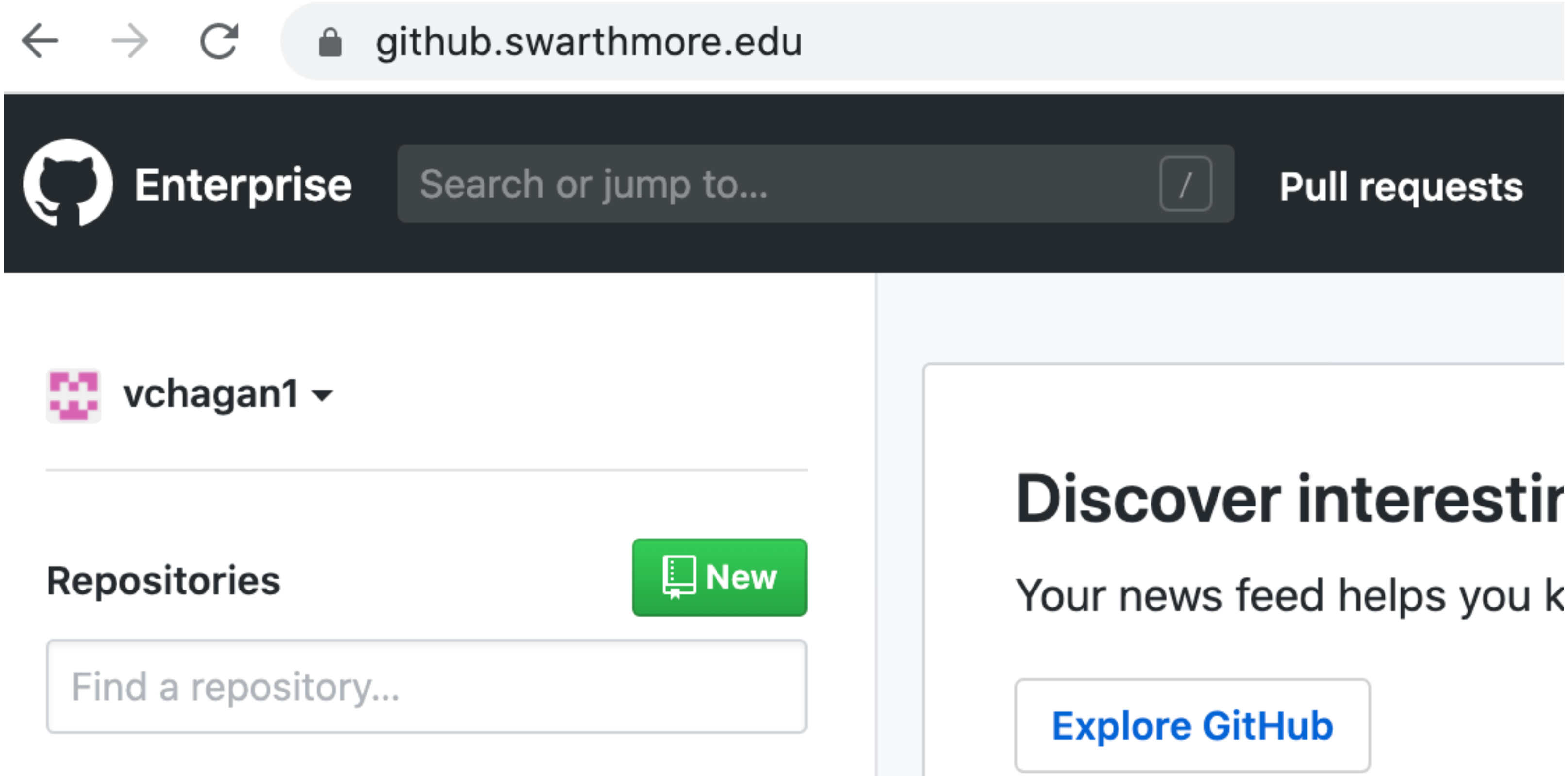
- supply name and e-mail to sites (and more!)
- third-party cookies (ad networks) follow you across multiple sites.



Cookies and Privacy

Cookies permit sites to learn a lot about you

You could turn them off …but good luck doing anything on the internet!



github.swarthmore.edu

Discover interestir

Your news feed helps you k

HTTP connections

Non-persistent HTTP

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

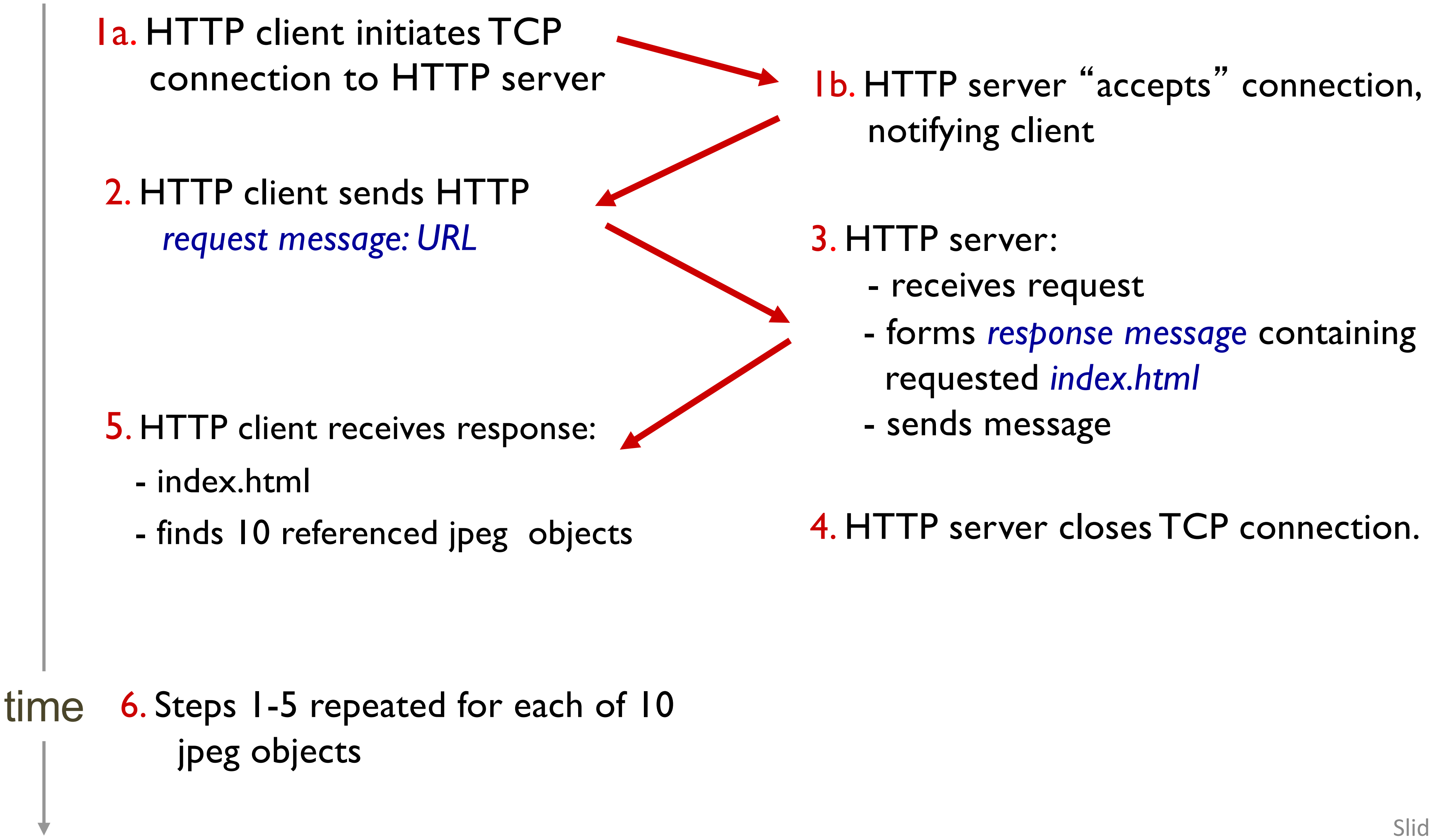
Persistent HTTP

- 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

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
```

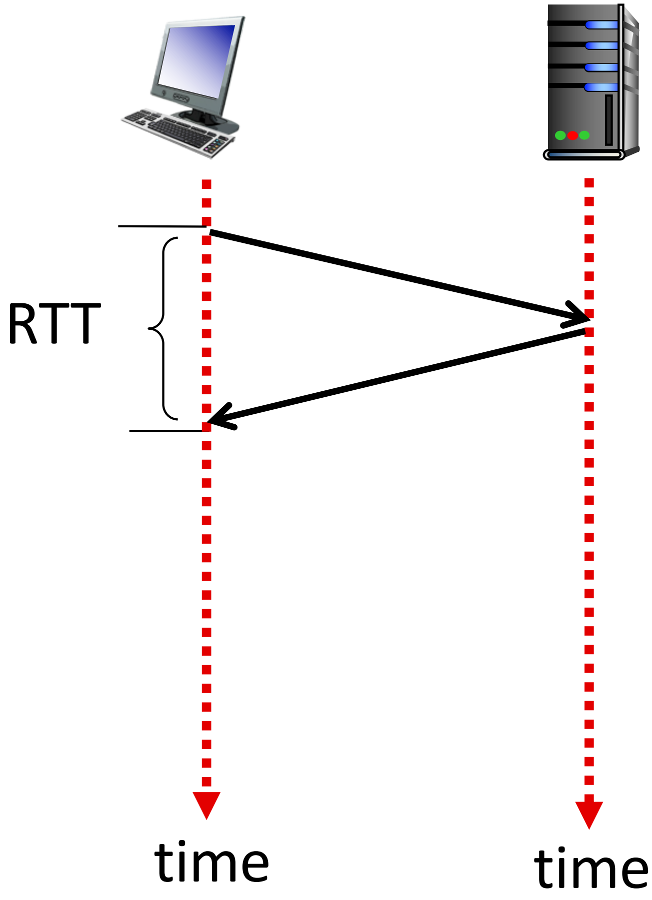
persistent HTTP

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

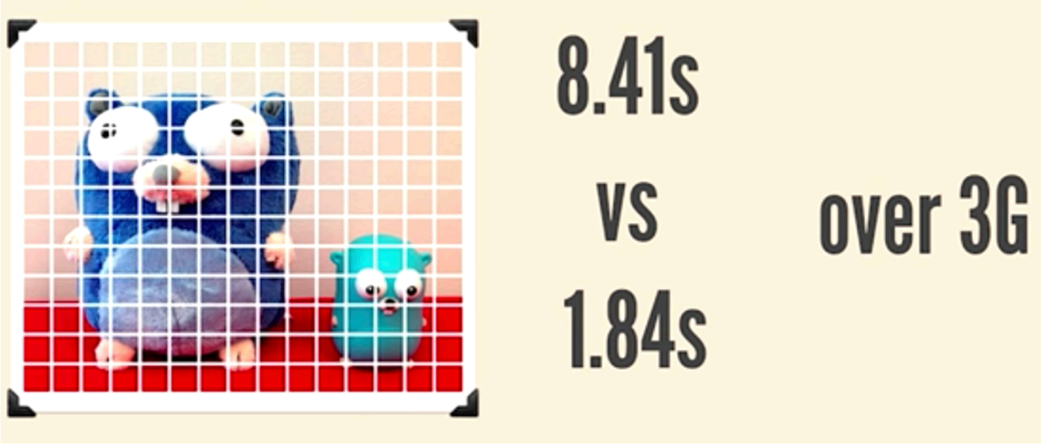
Round Trip Time

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.



HTTP 1.x vs HTTP 2.0



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

Courtesy: HTTP/2 101 Chrome Dev Summit 2015

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

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

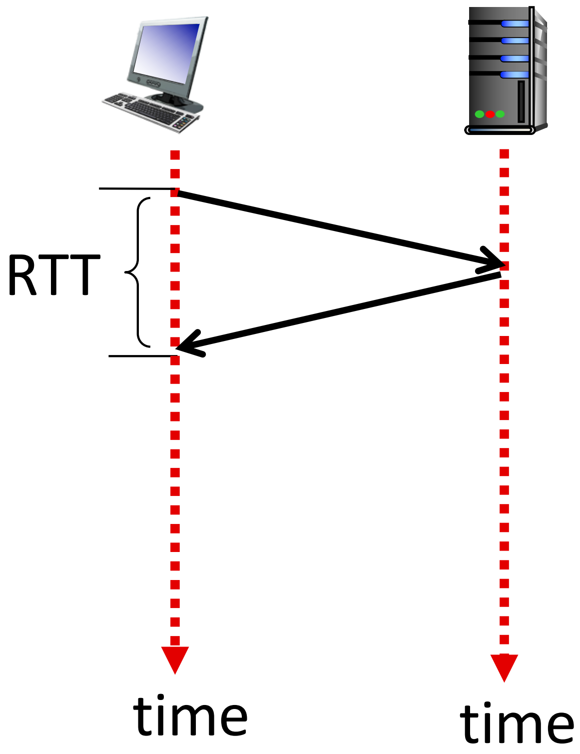
A. One RTT + (File transfer time per object)

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

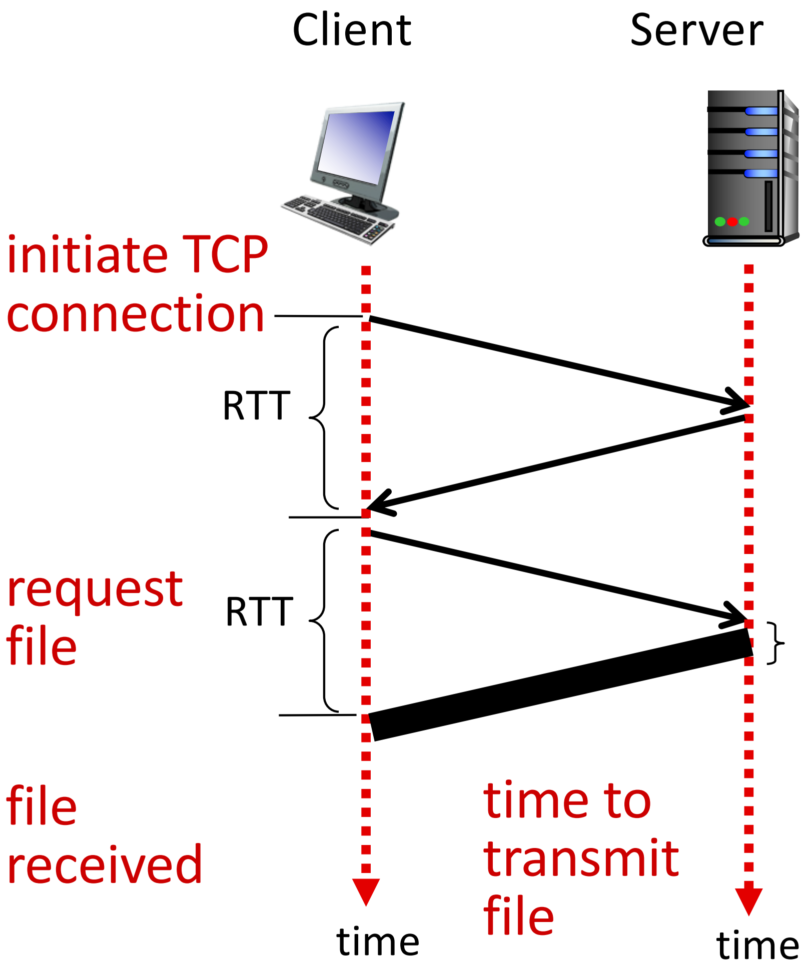
Round Trip Time (RTT): time for a small packet to travel from client to server and back

HTTP response time:

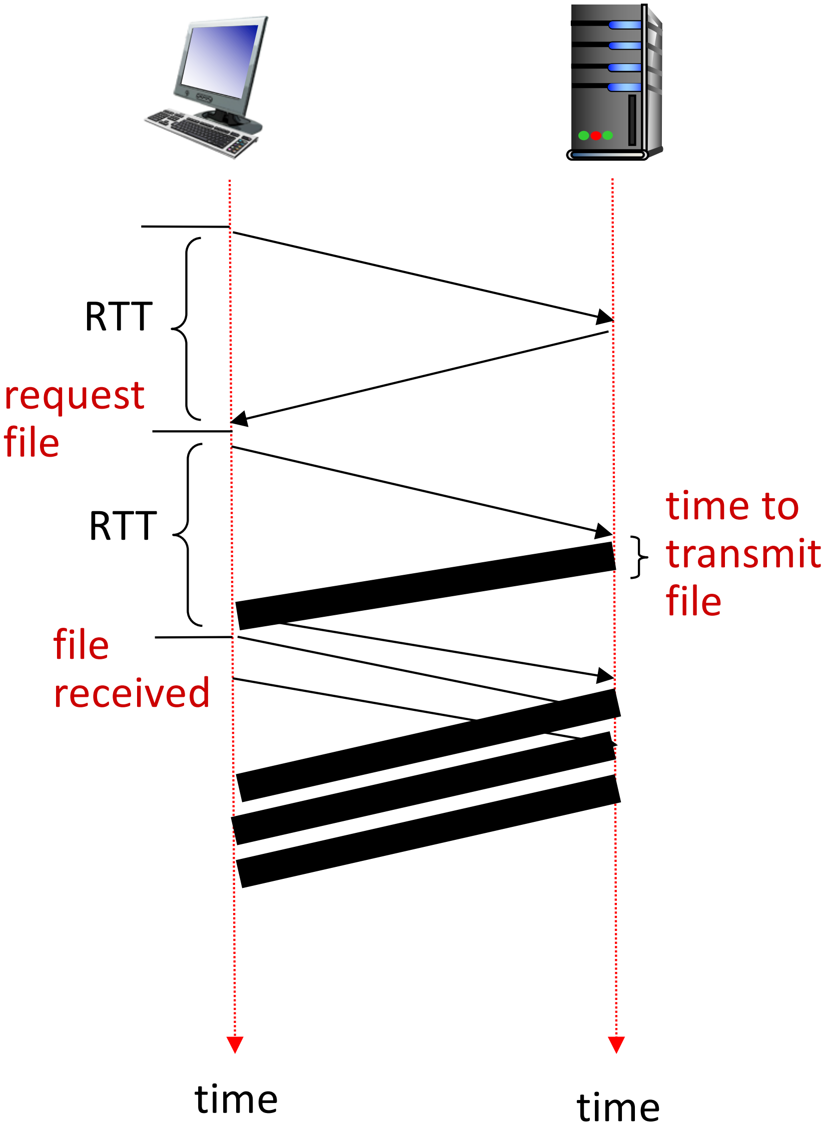
- 1-RTT to initiate TCP connection
- 1-RTT for HTTP request + first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2-RTT+ file transmission time

  <u>For each object</u>



Persistent Connection



Persistent HTTP

Non-persistent HTTP issues:

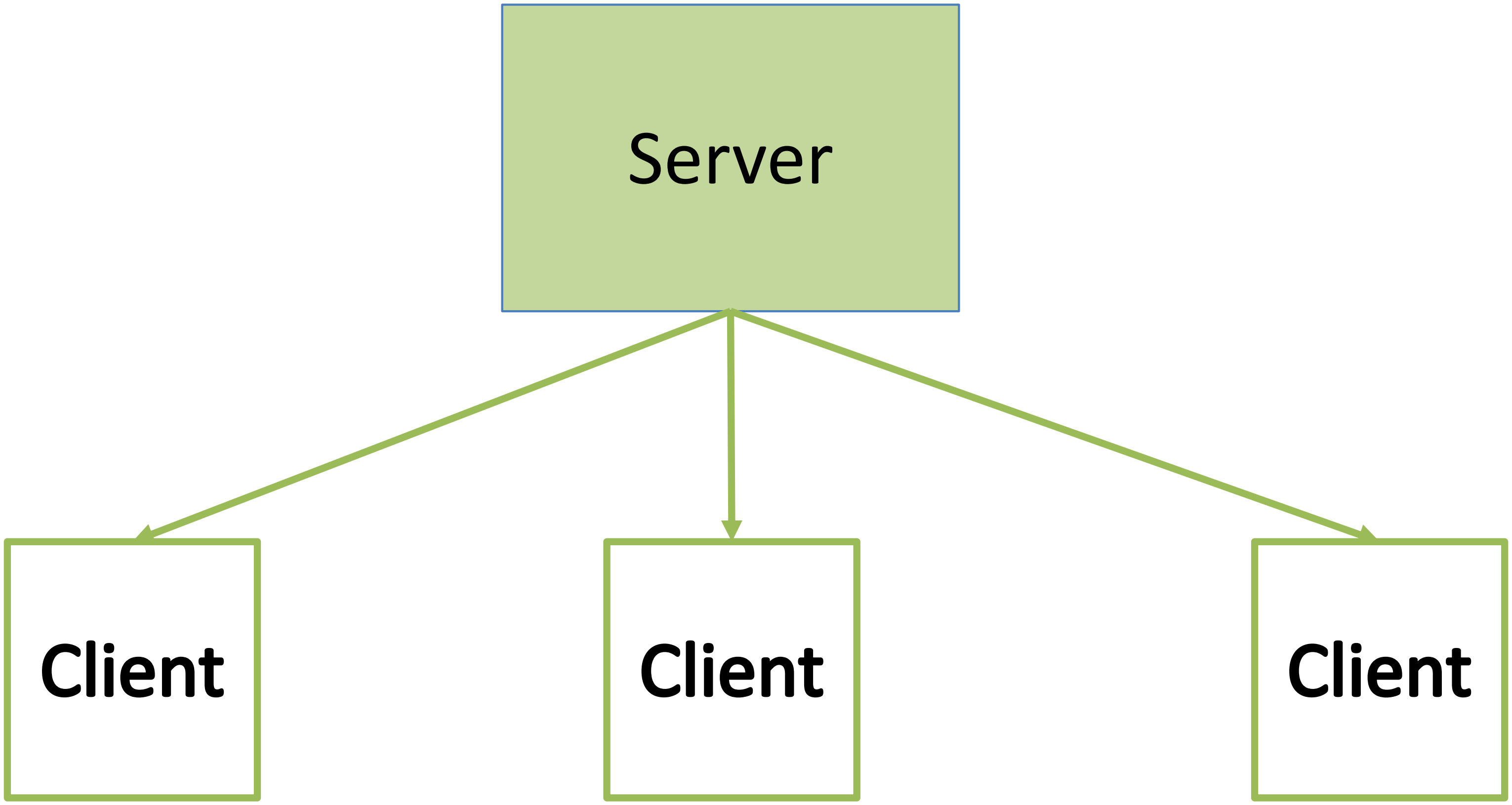
- requires 2 RTTs per object
- OS overhead for each TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

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

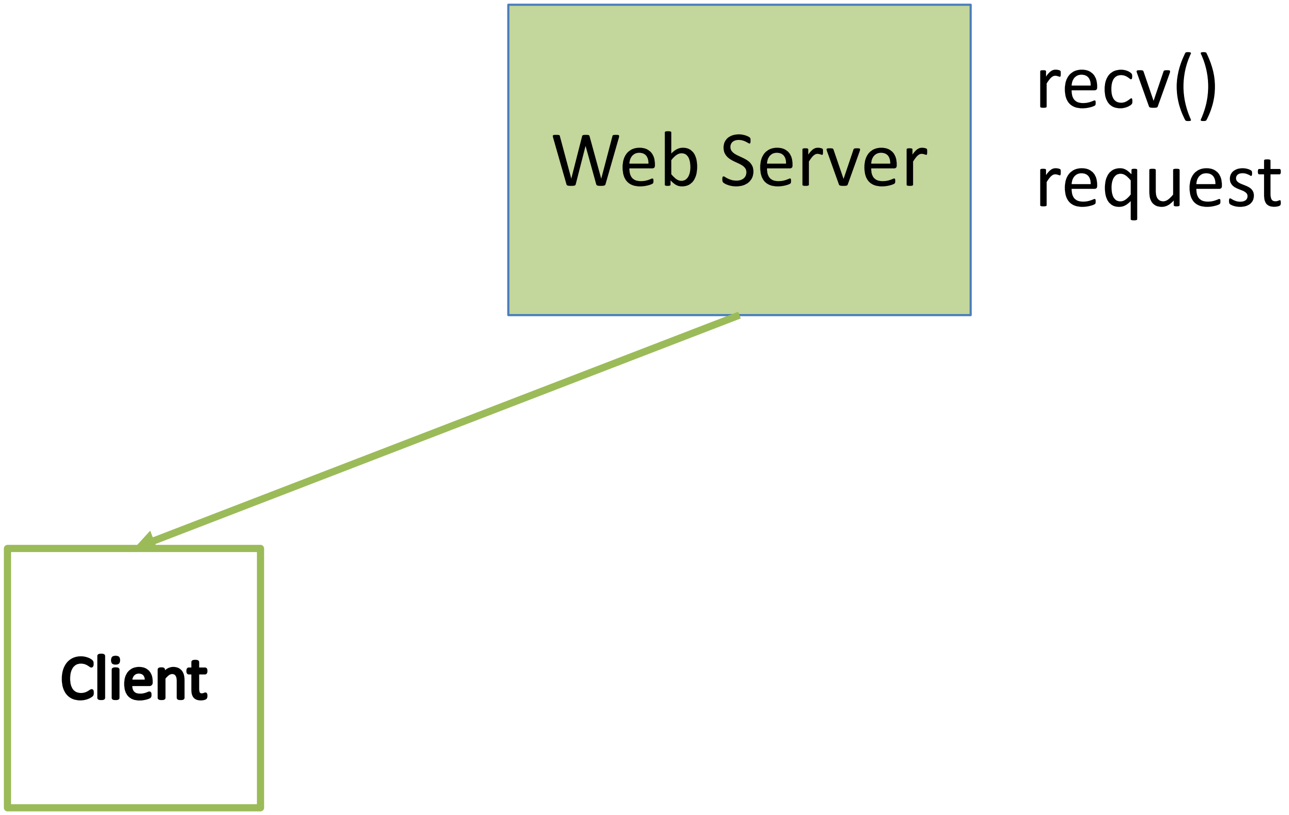
Concurrency

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



Without Concurrency

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



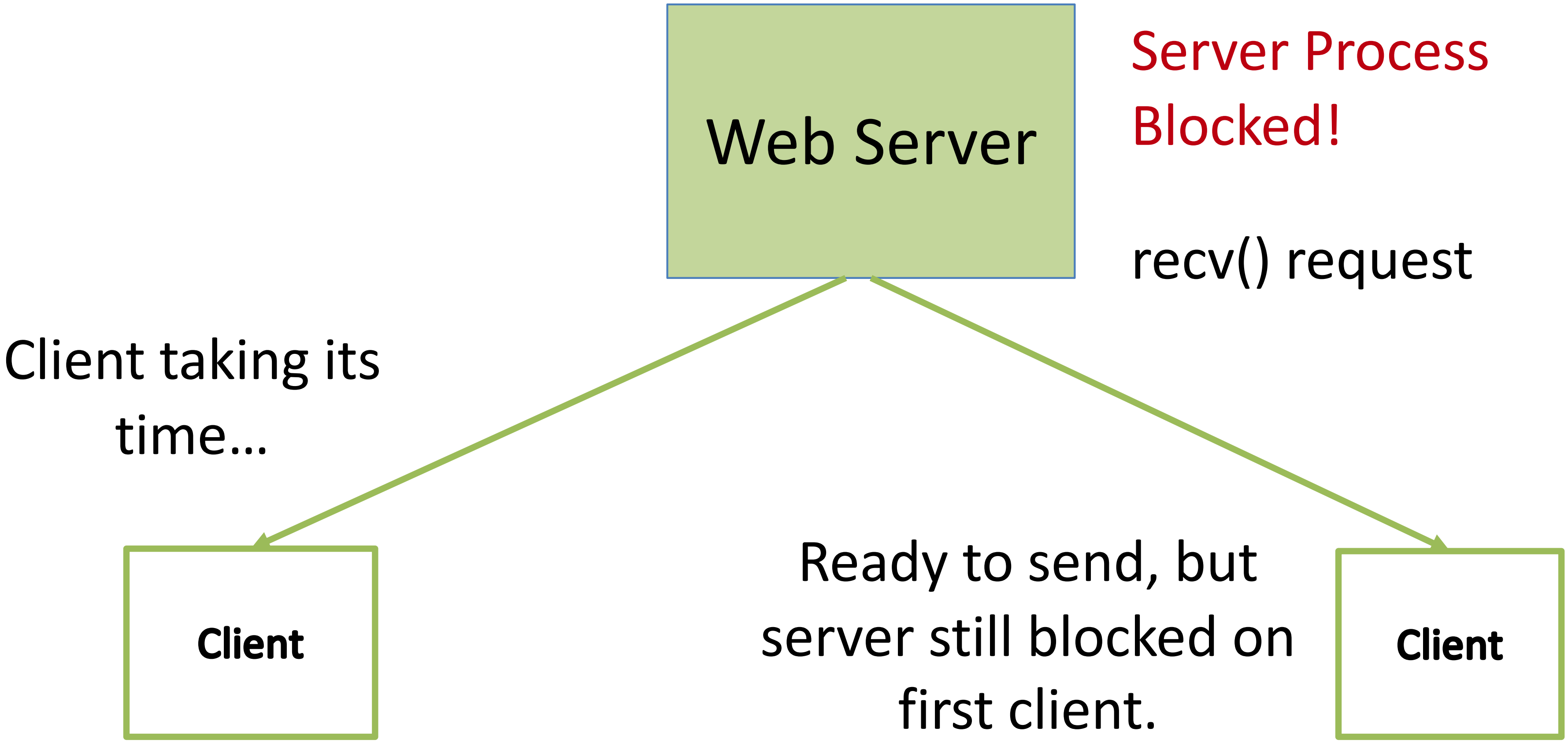
Web Server

recv()
request

Client

Without Concurrency

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



Web Server

Server Process Blocked!

recv() request

Client taking its time…

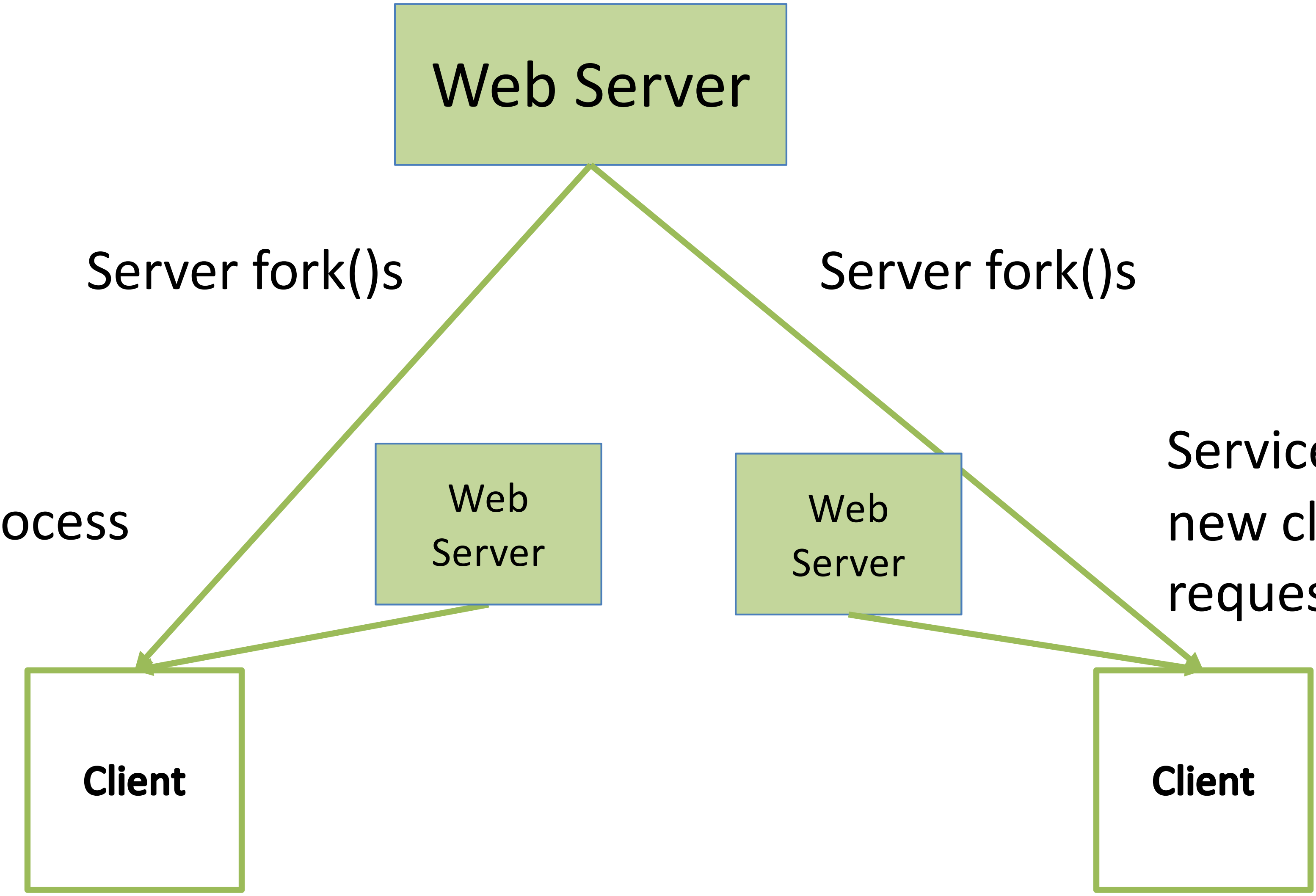
Client

Ready to send, but server still blocked on first client.

Client

If only we could handle these connections separately…

Multiple processes



Web Server

Server fork()s

Server fork()s

Web Server

Web Server

Child process recv()s

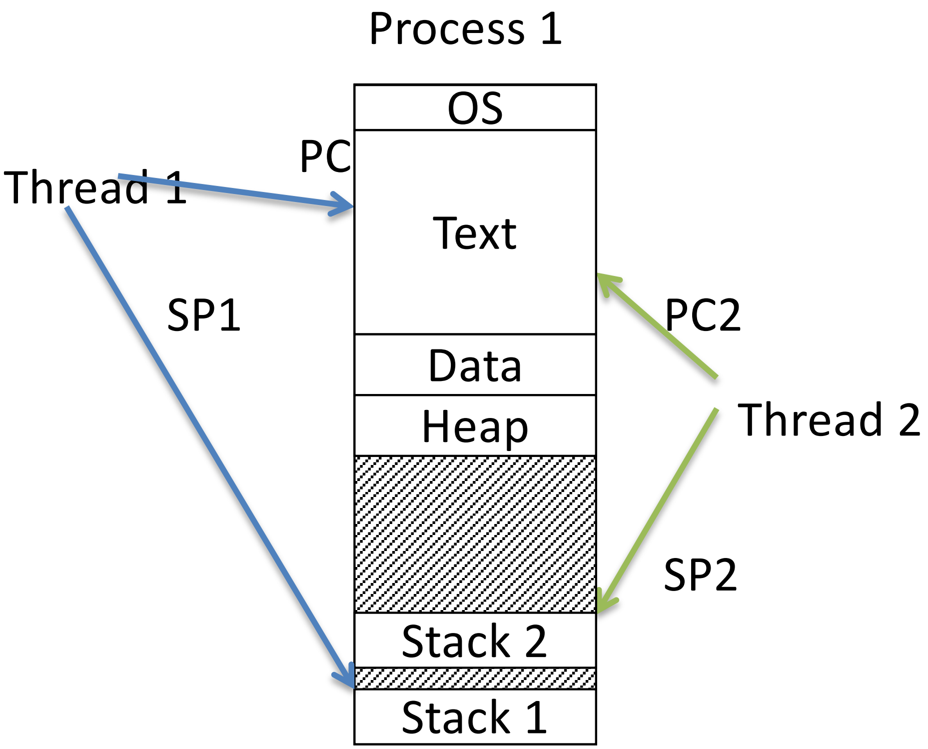
Services the new client request

Client

Client

Concurrent Web-servers with multiple threads/processes

- Threads (shared memory)



Process 1

OS

Text

Data

Heap

Stack 2

Stack 1

Thread 1

PC

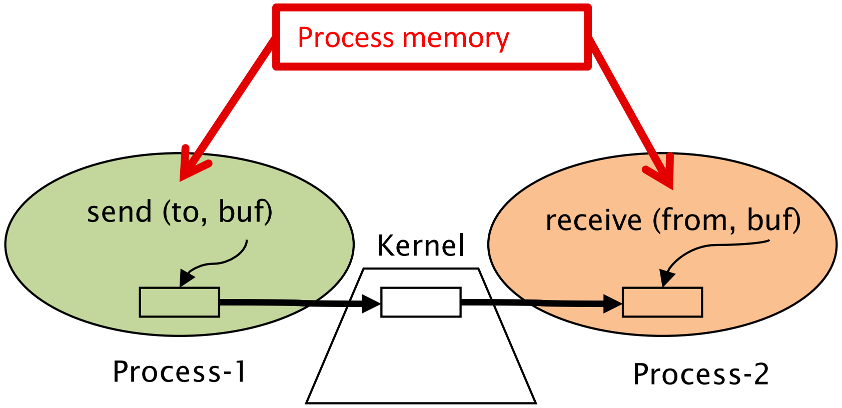
SP1

PC2

Thread 2

SP2

- Message Passing (locally)



Process memory

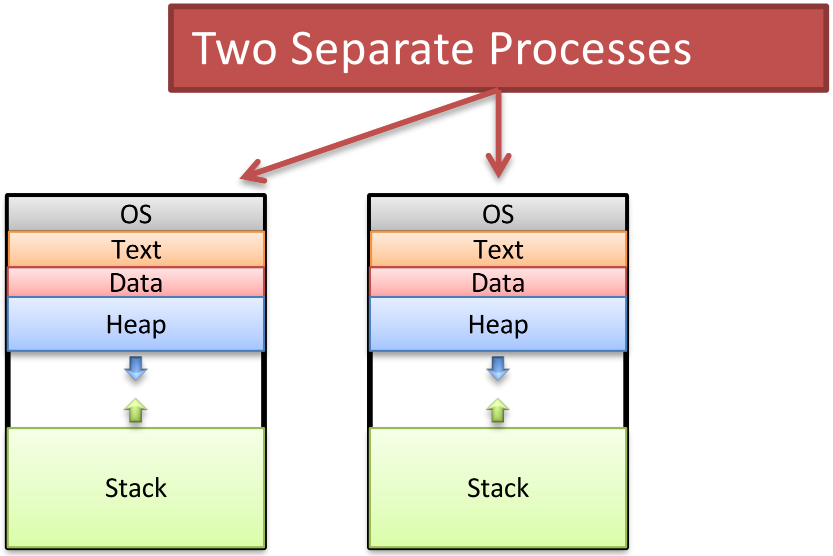
send (to, buf)

Kernel

receive (from, buf)

Process-1

Process-2



Two Separate Processes

OS

Text

Data

Heap

Stack

OS

Text

Data

Heap

Stack

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

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

Spawned Thread

- Shares descriptor table
- Shares memory
  - Uses parent's address space
- Scheduled independently

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

Which benefit is most critical?

A. Modular code/separation of concerns.

B. Multiple CPU/core parallelism.

C. I/O overlapping.

D. Some other benefit.

Both processes and threads:

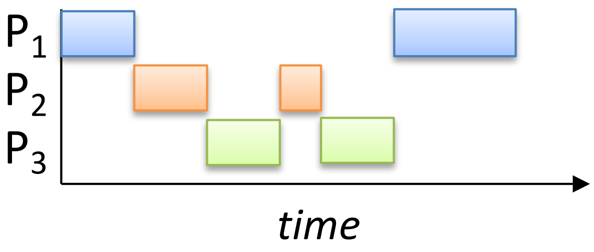
Several benefits

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

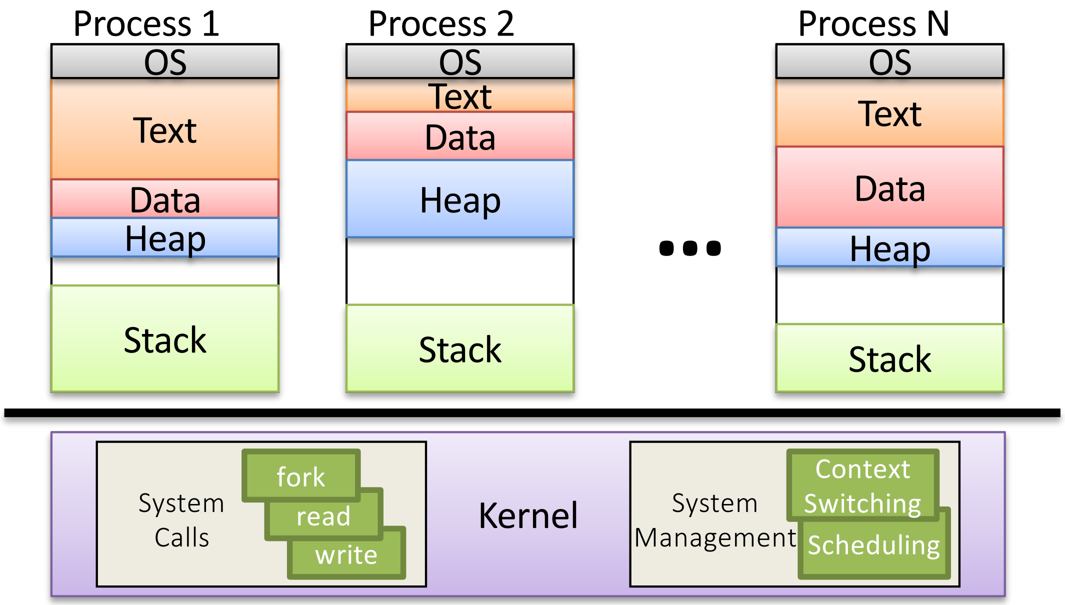
Both processes and threads

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

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!

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?

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
  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
```

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: 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: advantages

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