LECTURE 19 · CSE 333 · Summer 2026

Networking and Sockets (cont.d)

Discussion: Do you have more trouble starting a project? Or finishing a project?

Reminders

Assessments:

- Homework 3 due Friday at 11:59pm
- Releasing Exercise 8 soon
- Releasing Homework 4 soon (tricky in a different way!)

General:

- Tim is doing a GPU lecture on Week 9 Friday
- Midterm statistics
  - Last time someone asked me to post them. But… why does it matter?

Reminders

Assessments:

- Homework 3 due Friday at 11:59pm
- Releasing Exercise 8 soon
- Releasing Homework 4 soon (tricky in a different way!)

General:

- Tim is doing a GPU lecture on Week 9 Friday
- Midterm statistics

Midterm 68.0 points

Minimum	Median	Maximum	Mean	Std Dev
5.88%	77.94%	100.0%	75.76%	18.77%

Review

Terminology

`host`	`port`	`connection`	`byte stream`
which machine	**which program on it**	**one conversation**	**what flows through it**
attu, or your laptop. Found by name or by IP address.	One machine runs many services. 22 is SSH, 3333 is ours.	Two endpoints, opened on purpose, kept apart from everyone else's.	Bytes, in order. Not messages. Just bytes.

Once you have these four, the code is a handful of calls similar to the file I/O you already wrote.

A socket is a file descriptor

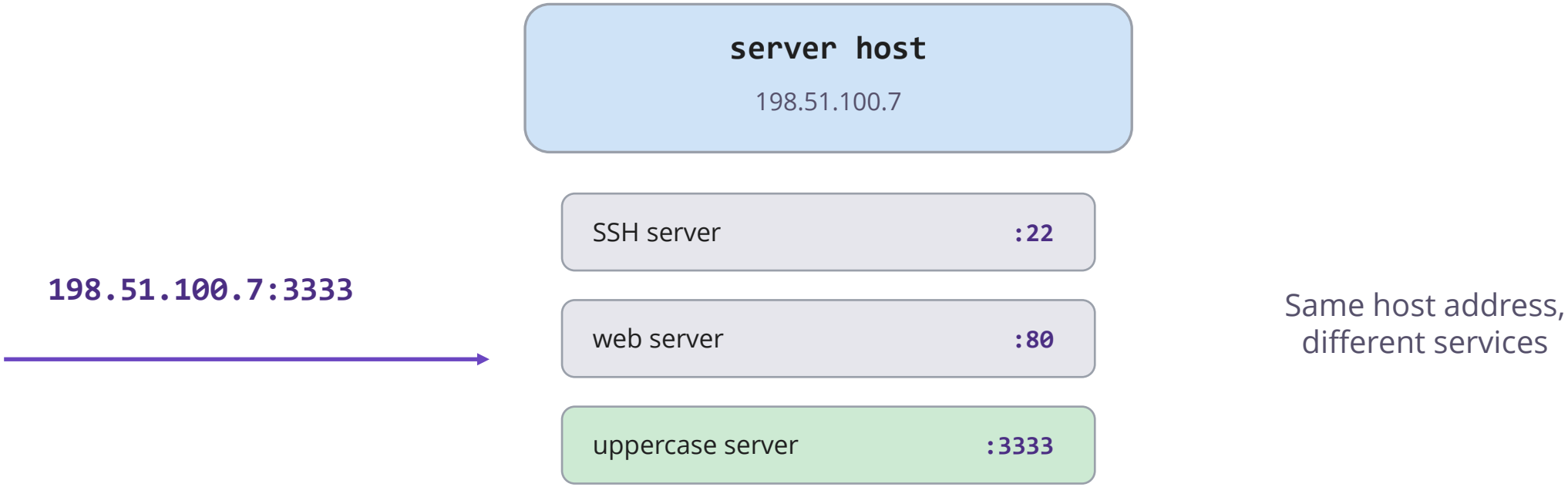
```c
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
if (socket_fd == -1) return EXIT_FAILURE;

char buffer[64];
write(socket_fd, "hello\n", 6);
read(socket_fd, buffer, sizeof(buffer));
close(socket_fd);
```

file descriptor table

0	pipe	stdin
1	pipe	stdout
2	pipe	stderr
3	TCP socket	remote process

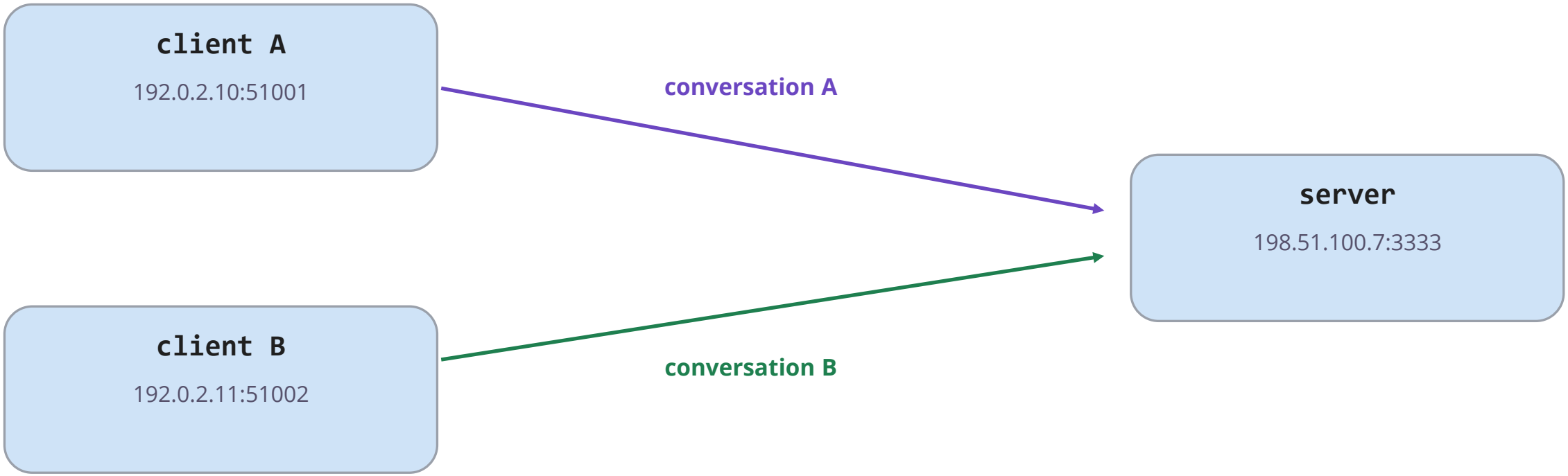
Network I/O uses familiar UNIX file operations.

A port selects one application on the host



Same host address, different services

A connection identifies one conversation



One write can arrive through several reads

```
1  write(socket_fd, "hello\n", 6);
2  // receiver might observe:
3  read(socket_fd, buf, 64);  // "he"
4  read(socket_fd, buf, 64);  // "llo\n"
```

- Correct answer: **C**
- Accumulate bytes until the application message is complete

Building the Server

Shape

Building our server

```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
addrinfo* results = nullptr;
getaddrinfo(nullptr, port, &hints, &results);
int listen_fd = socket(results->ai_family,
                       results->ai_socktype,
                       results->ai_protocol);
bind(listen_fd, results->ai_addr, results->ai_addrlen);
listen(listen_fd, SOMAXCONN);
while (true) {
  int client_fd = accept(listen_fd, nullptr, nullptr);
  HandleClient(client_fd);
  close(client_fd);
}
```

What does it do

- bind claims a port others can look up
- listen says we are willing to be reached
- the loop keeps us running and picking up
- HandleClient is the code you already have

Ask for a localhost address

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  hints.ai_flags = AI_PASSIVE;
5  addrinfo* results = nullptr;
6  getaddrinfo(nullptr, port, &hints, &results);
7  int listen_fd = socket(results->ai_family,
8                         results->ai_socktype,
9                         results->ai_protocol);
10 bind(listen_fd, results->ai_addr, results->ai_addrlen);
11 listen(listen_fd, SOMAXCONN);
12 while (true) {
13   int client_fd = accept(listen_fd, nullptr, nullptr);
14   HandleClient(client_fd);
15   close(client_fd);
16 }
```

We are describing our intent

- AF_UNSPEC: IPv4 or IPv6, either is fine
- SOCK_STREAM: a TCP byte stream
- AI_PASSIVE: an address we are allowed to claim

Look up the address and create the socket

```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
addrinfo* results = nullptr;
getaddrinfo(nullptr, port, &hints, &results);
int listen_fd = socket(results->ai_family,
                       results->ai_socktype,
                       results->ai_protocol);
bind(listen_fd, results->ai_addr, results->ai_addrlen);
listen(listen_fd, SOMAXCONN);
while (true) {
  int client_fd = accept(listen_fd, nullptr, nullptr);
  HandleClient(client_fd);
  close(client_fd);
}
```

We made an OS call

- nullptr for the host means one of ours
- socket() returns a file descriptor, or -1
- It has no port and no role yet

Claim the port with bind()

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  hints.ai_flags = AI_PASSIVE;
5  addrinfo* results = nullptr;
6  getaddrinfo(nullptr, port, &hints, &results);
7  int listen_fd = socket(results->ai_family,
8                         results->ai_socktype,
9                         results->ai_protocol);
10 bind(listen_fd, results->ai_addr, results->ai_addrlen);
11 listen(listen_fd, SOMAXCONN);
12 while (true) {
13   int client_fd = accept(listen_fd, nullptr, nullptr);
14   HandleClient(client_fd);
15   close(client_fd);
16 }
```

Bind to our file descriptor

- Attaches the descriptor to local port 3333
- 3333 is the number we publish to users
- Two programs cannot hold the same port
- Similar to open(...) but does some extra work...

Announce with listen()

```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
addrinfo* results = nullptr;
getaddrinfo(nullptr, port, &hints, &results);
int listen_fd = socket(results->ai_family,
                       results->ai_socktype,
                       results->ai_protocol);
bind(listen_fd, results->ai_addr, results->ai_addrlen);
listen(listen_fd, SOMAXCONN);
while (true) {
  int client_fd = accept(listen_fd, nullptr, nullptr);
  HandleClient(client_fd);
  close(client_fd);
}
```

This answers whether we are open

- Marks the descriptor as a listening socket
- The kernel starts queueing arrivals
- **SOMAXCONN** = max connections this server will take at once

Accept clients in a loop

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  hints.ai_flags = AI_PASSIVE;
5  addrinfo* results = nullptr;
6  getaddrinfo(nullptr, port, &hints, &results);
7  int listen_fd = socket(results->ai_family,
8                         results->ai_socktype,
9                         results->ai_protocol);
10 bind(listen_fd, results->ai_addr, results->ai_addrlen);
11 listen(listen_fd, SOMAXCONN);
12 while (true) {
13   int client_fd = accept(listen_fd, nullptr, nullptr);
14   HandleClient(client_fd);
15   close(client_fd);
16 }
```

Handle a single client

- ❖ accept blocks until somebody connects
- ❖ It returns a brand new descriptor
- ❖ That descriptor is the one HandleClient wanted
- ❖ close ends one conversation, not the server

accept() returns a second descriptor

```
1  int client_fd = accept(listen_fd, nullptr, nullptr);
2  if (client_fd == -1) {
3    if (errno == EINTR) continue;
4    return EXIT_FAILURE;
5  }
6  HandleClient(client_fd);
7  close(client_fd);
```

Two descriptors, two jobs

- listen_fd: accepts future connections
- client_fd: carries this client's bytes
- Closing client_fd leaves listen_fd open
- accept blocks by default when the queue is empty

Complete

Serve repeatedly, one client at a time

```
1  while (true) {
2    int client_fd = accept(listen_fd, nullptr, nullptr);
3    if (client_fd == -1) {
4      if (errno == EINTR) continue;
5      break;
6    }
7    HandleClient(client_fd);  // may block
8    close(client_fd);
9  }
10 close(listen_fd);
```

The deliberate limitation

- Only one thread of execution
- A slow client blocks every waiting client
- Correct first, concurrent later

Full server code

```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
addrinfo* results = nullptr;
getaddrinfo(nullptr, port, &hints, &results);
int listen_fd = socket(results->ai_family,
                       results->ai_socktype,
                       results->ai_protocol);
bind(listen_fd, results->ai_addr, results->ai_addrlen);
listen(listen_fd, SOMAXCONN);
while (true) {
  int client_fd = accept(listen_fd, nullptr, nullptr);
  HandleClient(client_fd);
  close(client_fd);
}
```

```
$ ./server &

$ nc localhost 3333

hello, networks

HELLO, NETWORKS
```

One copy but any number of users

- The uppercase logic never changed
- netcat is somebody else's client
- Your own client comes next

How many sockets

One client is connected and being served. How many sockets does the server have open?

```
1  listen(listen_fd, SOMAXCONN);
2  int client_fd = accept(listen_fd, nullptr, nullptr);
3  HandleClient(client_fd);   // ? how many are open here
```

Ask yourself what the next client would arrive on.

Predict first: Write an answer down before we move on.

listen_fd and client_fd

`accept()`

`listen_fd`

the doorway

`no client yet`

One socket exists. It is bound and listening, and it carries no request bytes.

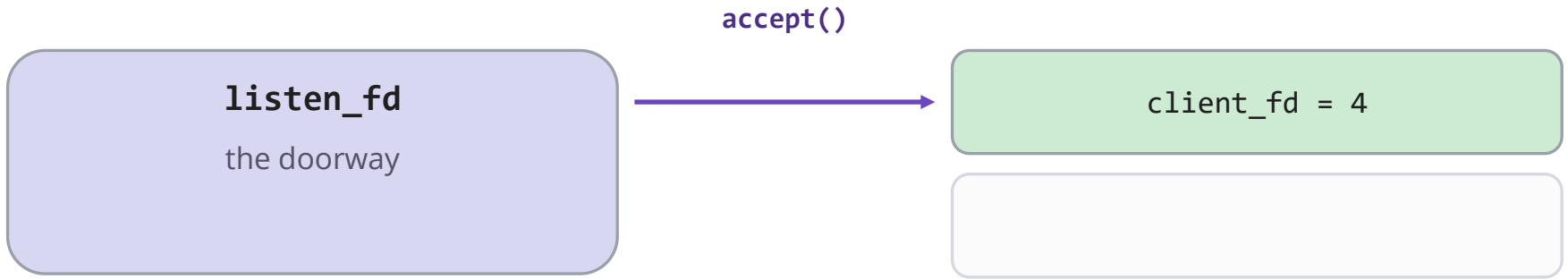
file descriptor table

`0-2`	std streams	`terminal`
`3`	socket	`listening :3333`

What listen_fd is for

- It was created, bound, and marked listening
- It is the address clients dial
- read() on it never returns request data

listen_fd and client_fd



accept() did not change listen_fd. It returned a second descriptor, and now both are open.

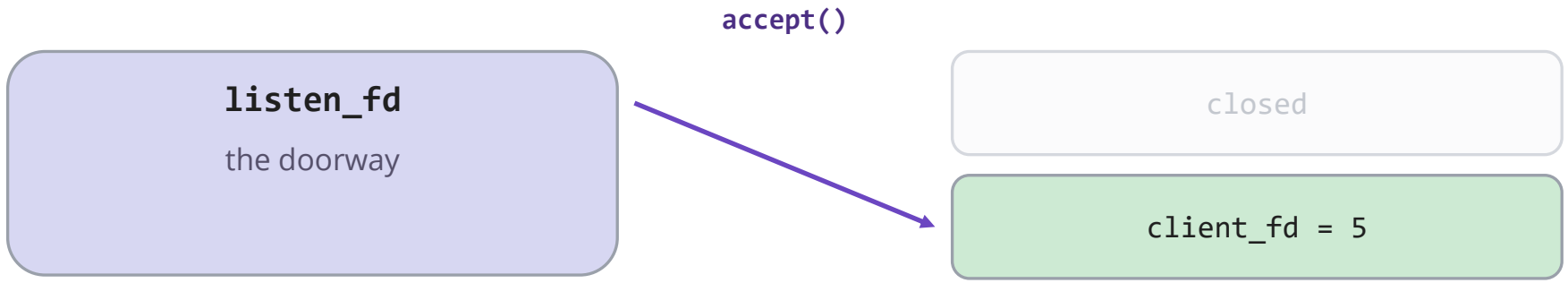
file descriptor table

0-2	std streams	terminal
3	socket	listening :3333
4	socket	client 10.0.0.7

What accept() actually did

- listen_fd is untouched and still listening
- client_fd is a new row in the same table
- Only client_fd carries this client's bytes
- HandleClient wanted this one

listen_fd and client_fd



close(client_fd) ends one conversation. listen_fd survives, and the next accept() returns a fresh descriptor.

file descriptor table

0-2	std streams	terminal
3	socket	listening :3333
4	closed	was 10.0.0.7
5	socket	client 10.0.0.9

Two lifetimes

- listen_fd lasts as long as the server
- Each client_fd lasts exactly one conversation
- close(listen_fd) is what shuts the service down
- Forgetting to close client_fd runs you out of descriptors

Building the Client

Building the client

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  addrinfo* results = nullptr;
5  getaddrinfo(host, port, &hints, &results);
6  int fd = socket(results->ai_family,
7                  results->ai_socktype,
8                  results->ai_protocol);
9  connect(fd, results->ai_addr, results->ai_addrlen);
10 freeaddrinfo(results);
11 WriteAll(fd, "hello\n", 6);
12 string reply = ReadLine(fd);
13 close(fd);
```

Mostly familiar

- Lines 1 to 10 produce a connected fd
- The last three are the uppercase exchange
- No loop: one conversation, then done

Get address of the server (instead of localhost)

```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
addrinfo* results = nullptr;
getaddrinfo(host, port, &hints, &results);
int fd = socket(results->ai_family,
                results->ai_socktype,
                results->ai_protocol);
connect(fd, results->ai_addr, results->ai_addrlen);
freeaddrinfo(results);
WriteAll(fd, "hello\n", 6);
string reply = ReadLine(fd);
close(fd);
```

Two small edits

- No hints.ai_flag = AI_PASSIVE
  - we are not claiming an address
- A real host name and port go in
- results is still a list of candidates

Create the socket and connect

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  addrinfo* results = nullptr;
5  getaddrinfo(host, port, &hints, &results);
6  int fd = socket(results->ai_family,
7                  results->ai_socktype,
8                  results->ai_protocol);
9  connect(fd, results->ai_addr, results->ai_addrlen);
10 freeaddrinfo(results);
11 WriteAll(fd, "hello\n", 6);
12 string reply = ReadLine(fd);
13 close(fd);
```

Connecting

- socket() is unchanged from the server
- connect names the remote endpoint
- This is the line that touches the network
- Then the candidate list can be released

Now it is just file I/O

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  addrinfo* results = nullptr;
5  getaddrinfo(host, port, &hints, &results);
6  int fd = socket(results->ai_family,
7                  results->ai_socktype,
8                  results->ai_protocol);
9  connect(fd, results->ai_addr, results->ai_addrlen);
10 freeaddrinfo(results);
11 WriteAll(fd, "hello\n", 6);
12 string reply = ReadLine(fd);
13 close(fd);
```

Similar

- ❖ WriteAll sends the request line
- ❖ ReadLine stops at the newline we agreed on
- ❖ close releases the fd exactly like a file

Finished client!

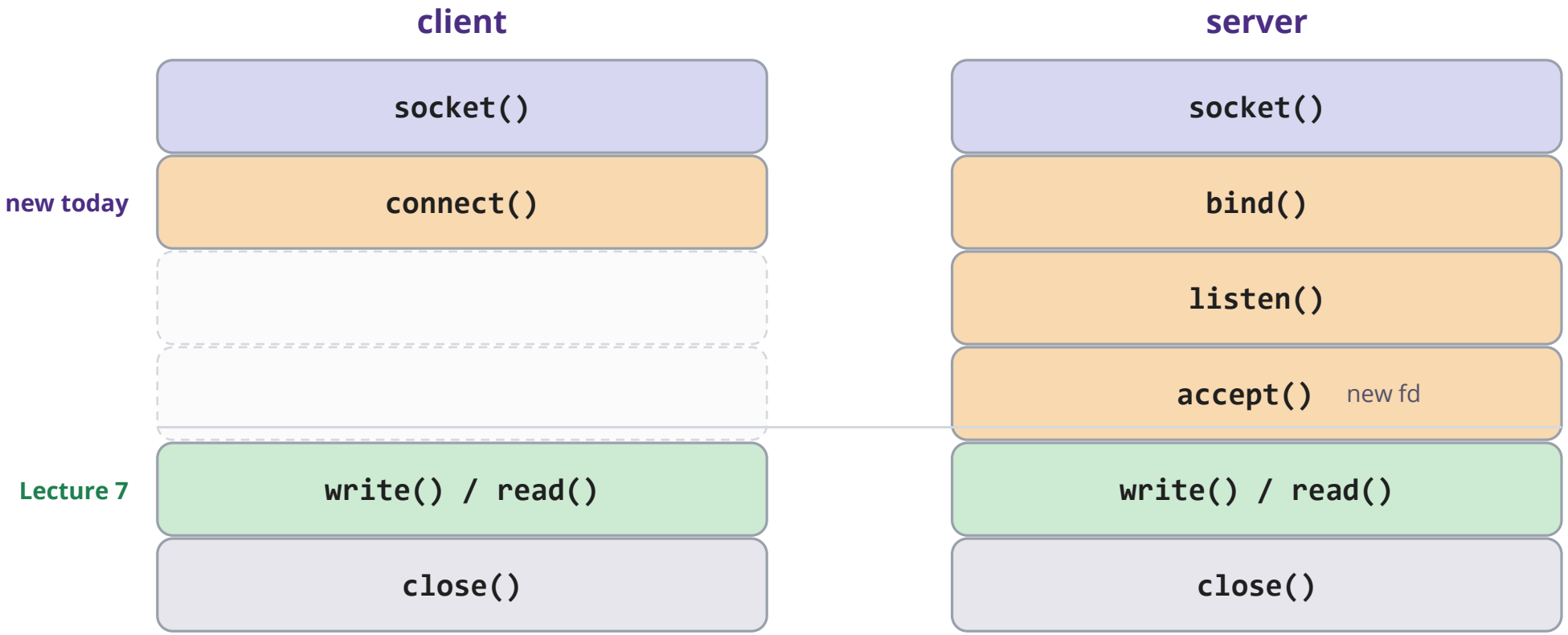
```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
addrinfo* results = nullptr;
getaddrinfo(host, port, &hints, &results);
int fd = socket(results->ai_family,
                results->ai_socktype,
                results->ai_protocol);
connect(fd, results->ai_addr, results->ai_addrlen);
freeaddrinfo(results);
WriteAll(fd, "hello\n", 6);
string reply = ReadLine(fd);
close(fd);
```

```
$ ./client attu 3333

hello, networks

HELLO, NETWORKS
```

Where we started

- argv became a descriptor we read
- stdout became a descriptor we write
- The uppercase line never changed

Client and server, side by side



Both sides call socket(). The client names who it wants; the server claims a port and waits. Below the line, both sides are doing Lecture 7 file I/O.

getaddrinfo()

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;      // v4 or v6, either
3  hints.ai_socktype = SOCK_STREAM;  // TCP
4  addrinfo* results = nullptr;
5  int rc = getaddrinfo(host, port, &hints, &results);
6  if (rc != 0) {
7    cerr << gai_strerror(rc) << endl;
8    return EXIT_FAILURE;
9  }
10 for (addrinfo* a = results; a != nullptr; a = a->ai_next) {
11   fd = socket(a->ai_family, a->ai_socktype, a->ai_protocol);
12   if (fd == -1) continue;
13   if (connect(fd, a->ai_addr, a->ai_addrlen) == 0) break;
14   close(fd);
15 }
16 freeaddrinfo(results);
```

Four jobs

- Resolves the name through DNS
- Picks IPv4 or IPv6 for you
- Converts the port to network order
- Builds the structs from the last slide
- **It returns a list, not an answer.** Try candidates until one connects.

DNS and HTTP

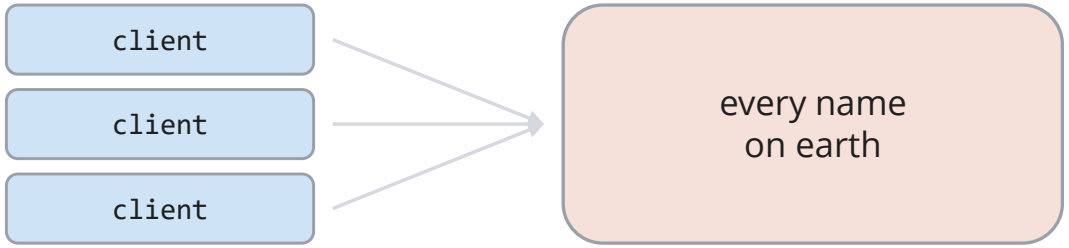
Two answers that do not scale

getaddrinfo() turns a name into an address. The mapping has to live somewhere. Where?

keep a file on every machine

```
$ cat /etc/hosts
127.0.0.1       localhost
128.208.1.138   attu.cs.washington.edu
```

keep one server for everyone



This is real, and it works

- Your machine checks it before anything else
- But every machine needs a copy
- And names change faster than you can ship it

Worse than it looks

- No machine can serve every lookup on the internet
- One outage takes down every name
- Whoever runs it decides who gets to exist

How would you split it?

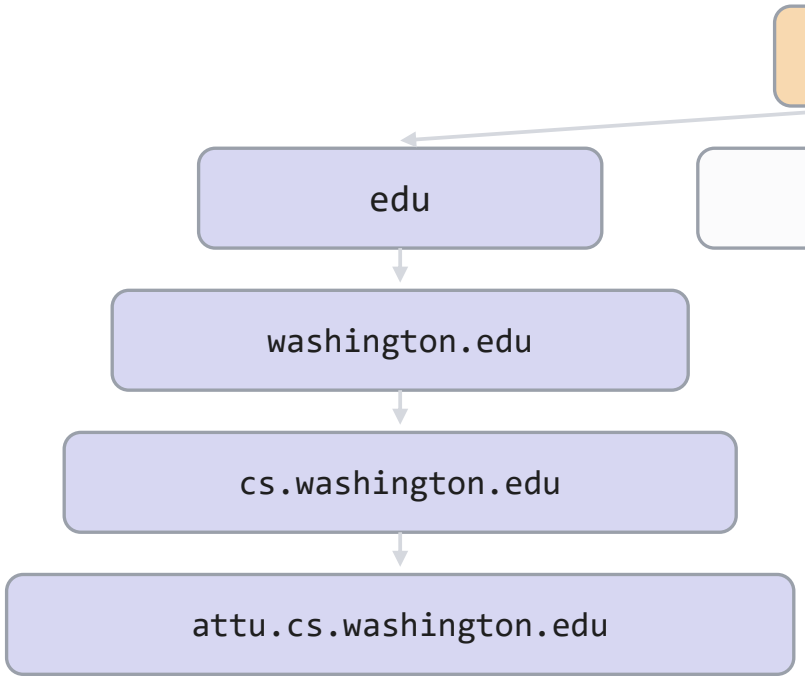
Too big for one file and too important for a single server.
How would you divide up the names?

```
1  www.cs.washington.edu
2  // ? who should be responsible for which part
```

You already use a naming system that splits like this. Kind of like file path but we read it backwards(ish)

Predict first: Write an answer down before we move on.

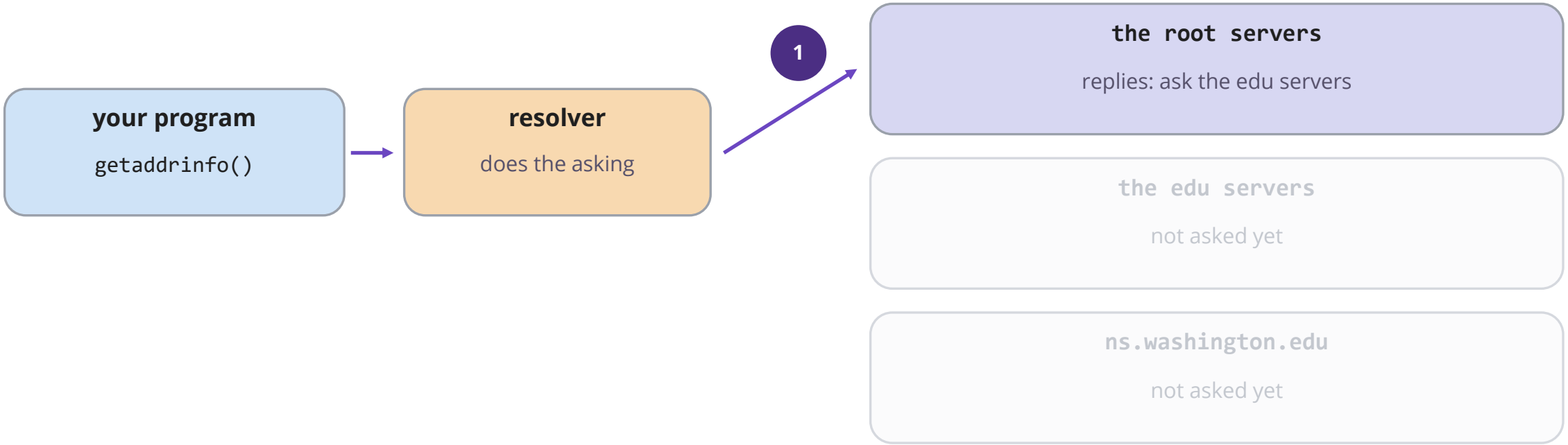
The name tree



Read the name right to left

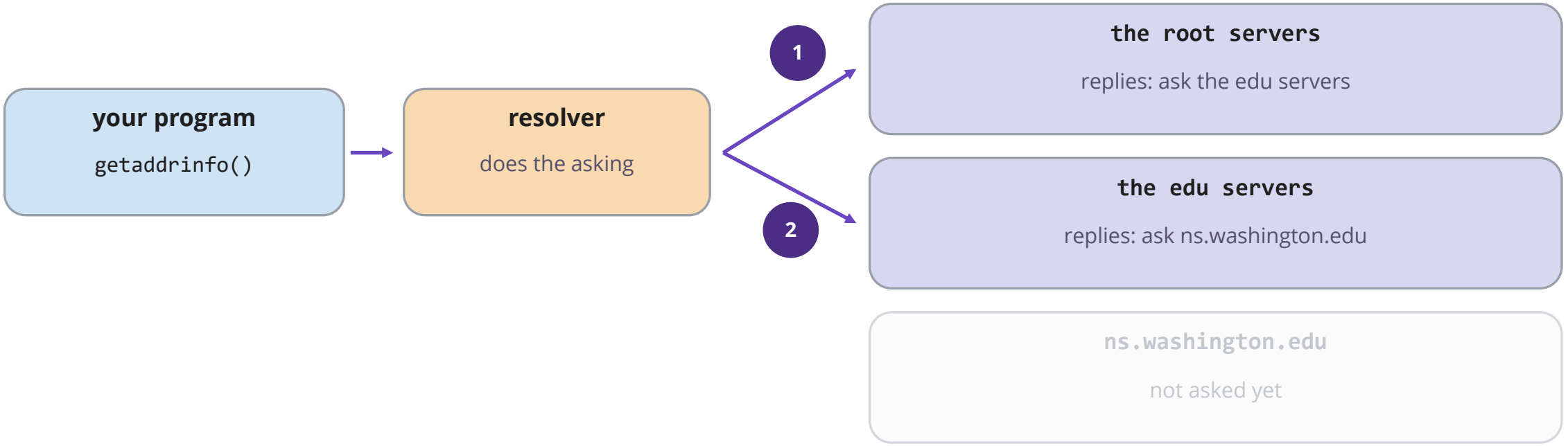
- Each level hands out the level below it
- UW was given washington.edu and answers for everything under it
- No server knows more than its own children
- Adding a machine here needs nobody else's permission

Resolving a name



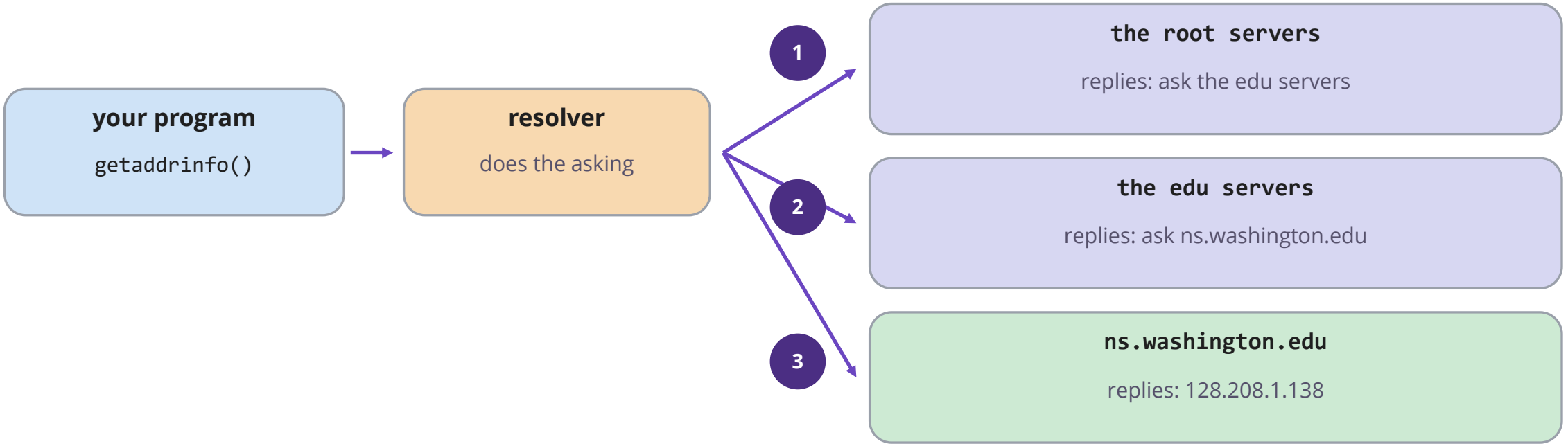
The resolver starts at the root, because the root is the only address it is born knowing. The root does not know attu, so it answers with a referral rather than an address.

Resolving a name



Same again, one level down. The edu servers do not know attu either, but they do know who was given washington.edu.

Resolving a name



UW was delegated this part of the tree, so ithe real answer. The resolver hands 128.208.1.138 back, and getaddrinfo() returns.

Caching and TTL

```
$ dig +noall +answer attu.cs.washington.edu
attu.cs.washington.edu. 3600 IN A 128.208.1.138

$ dig +noall +answer attu.cs.washington.edu
attu.cs.washington.edu. 3512 IN A 128.208.1.138
```

3600 is the time to live (in seconds)
The second lookup came from a cache and the counter is what is left

Why the root survives

- Every answer carries a TTL
- Resolvers cache it and reuse it until it expires
- Most lookups never leave your building
- The root is asked about edu, not about attu
- **The cost:** a changed address takes a TTL to be believed everywhere.

HTTP: Request

```
$ nc www.example.com 80
GET /index.html HTTP/1.1
Host: www.example.com
Connection: close
```

The last line is empty on purpose. That blank line is the whole framing rule.

Three parts

- **Request line:** method, path, version
- **Headers:** name, colon, value, one per line
- **A blank line:** the headers are over
- Lines end with carriage return and newline

HTTP: Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

<html><body>hello</body></html>
```

Same sort of shape as the request, but now with a status line on top and a body underneath.

How you know when to stop

- Read until the blank line to get the headers
- Then read exactly **Content-Length** more bytes
- Without it you cannot tell a slow body from a finished one
- 200, 404, 500: the number is for the program, the text is for you

HTTP in HW4

your server does	using
accept a connection	the socket calls from the review
read until a blank line	the readN loop, because read() comes up short
parse the request line	split on spaces, then look the query up in your index
write the response	WriteAll, because write() comes up short too

Two bugs you'll have to consider:

- A client that connects and sends nothing, so read() returns 0
- A Content-Length that does not match the body you wrote

Concurrency

A second client

Client A is connected and idle. Client B now runs the client program.

```
1  // server, one thread of execution
2  int client_fd = accept(listen_fd, nullptr, nullptr);
3  HandleClient(client_fd);   // ? blocked on A's newline
```

A. the connection is refused

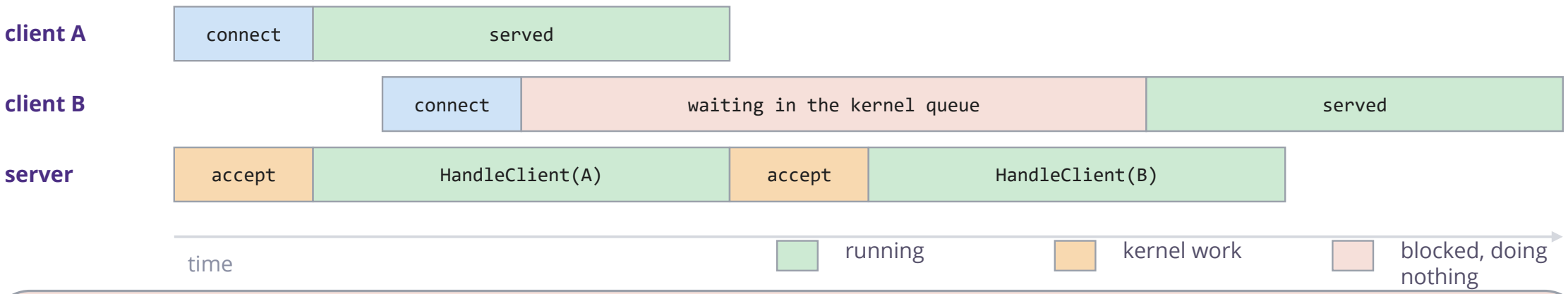
B. the connection succeeds and the bytes arrive immediately

C. the connection is queued, but nothing happens until we handle the other clients

D. the server crashes

Predict first: Separate 'the connect worked' from 'the server answered'.

One client at a time



listen() gives us a queue, but not a second "worker"

- B's connect returns right away, so B thinks it is connected
- But nothing is read until the server finishes with A
- One slow or idle client stalls every other client

Reminders

Assessments:

- Homework 3 due Friday at 11:59pm
- Releasing Exercise 8 soon
- Releasing Homework 4 soon (tricky in a different way!)

General:

- Tim is doing a GPU lecture on Week 9 Friday
- Midterm statistics

Questions?