UNIVERSITY of WASHINGTON

LECTURE 18 · CSE 333 · Summer 2026

Networking and Sockets

Discussion: Favorite healthy food? Favorite unhealthy food?

Reminders

Assessments:

- Exercise 7 due Wednesday at 11:59pm
- Homework 3 due Thursday at 11:59pm

General:

- How was the guest lecture?
- Midterm grades released
  - Review solutions and come to office hours
  - Submit a regrade requests if you think we made a mistake!

Review

unique_ptr: exactly one owner

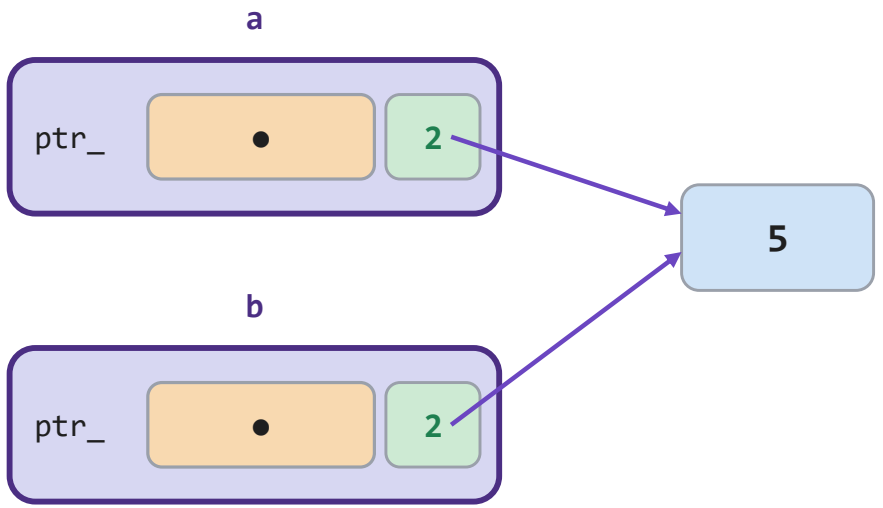
```cpp
#include <memory>
using namespace std;

std::unique_ptr<int> a(new int(5));
std::unique_ptr<int> b = a;   // COMPILE ERROR: no copy
std::cout << *a << std::endl;      // a is the sole owner
// preferred:
auto c = std::make_unique<int>(5);
```

Sole ownership

- **#include <memory>**, type **unique_ptr<T>**.
- **Copy and assign are** deleted - the toy's bug cannot compile.
- **Frees in its destructor,** zero overhead vs a raw pointer.
- **make_unique<T>(...)** is the preferred way to create one.

shared_ptr: count the 🍪wners

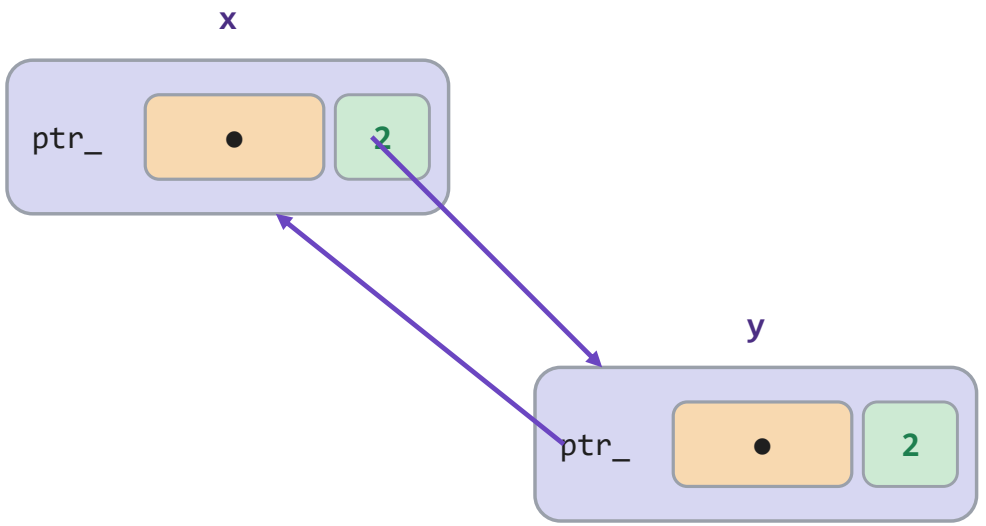
```
1  #include <memory>
2
3
4  std::shared_ptr<int> a(new int(5));  // count = 1
5  std::shared_ptr<int> b = a;          // count = 2 (copy ok)
6  std::cout << a.use_count();          // 2
```



one object, count = 2 owners

Uh-oh! Reference cycles

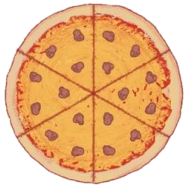
```
1  struct Node {
2      std::shared_ptr<Node> other;
3  };
4
5  auto x = std::make_shared<Node>();
6  auto y = std::make_shared<Node>();
7  x->other = y;   // y's count -> 2
8  y->other = x;   // x's count -> 2
```



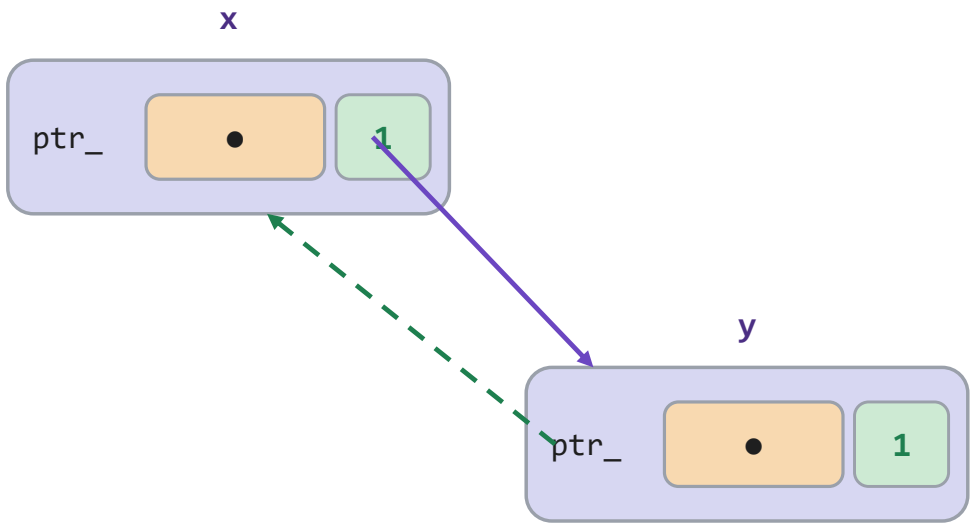
x and y leave scope: 2 -> 1, never 0

REVIEW

weak_ptr: observe without owning



```
1  struct Node {
2    std::shared_ptr<Node> next;   // owns forward
3    std::weak_ptr<Node>   prev;   // does NOT count
4  };
5
6  // use a weak_ptr by promoting it:
7  if (auto p = n->prev.lock()) {
8    // p is a shared_ptr; object still alive
9  }
```



back-edge is weak: does not count

REVIEW

Comparison

`unique_ptr<T>`	`shared_ptr<T>`	`weak_ptr<T>`
One owner	Many owners	No ownership
Copy: DELETED	Copy: count + 1	Does not count
Move to transfer	Free at count 0	lock() -> shared_ptr
Zero overhead	use_count()	expired()
make_unique<T>()	make_shared<T>()	Breaks cycles
Your DEFAULT	Only if truly shared	Use for back-edges

Random Aside

Networking

What is a network?

Nodes

devices that send, receive, or forward data

Links

wired or wireless paths between nearby nodes

Protocols

shared rules for formatting and exchanging data

The Internet is a network of networks joined by routers and common protocols.

The complete seven-layer model

#	OSI layer	Responsibility	Examples	Internet stack
7	**Application**	Meaning and message rules	`HTTP, DNS`	**Application**
6	**Presentation**	Representation, encryption	`UTF-8, TLS`	**Application**
5	**Session**	Conversation coordination	`RPC session`	**Application**
4	**Transport**	Process-to-process delivery	`TCP, UDP`	**Transport**
3	**Network**	Host-to-host routing	`IP`	**Internet**
2	**Data link**	Delivery across one link	`Ethernet`	**Link**
1	**Physical**	Bits as signals	`copper, radio`	**Physical**

Layer 1: Physical

Move individual bits across one physical medium.

Copper

voltage changes

`0 1 1 0`

Fiber

light pulses

`dim bright bright dim`

Wi-Fi

radio waves

`modulated signal`

The layer provides

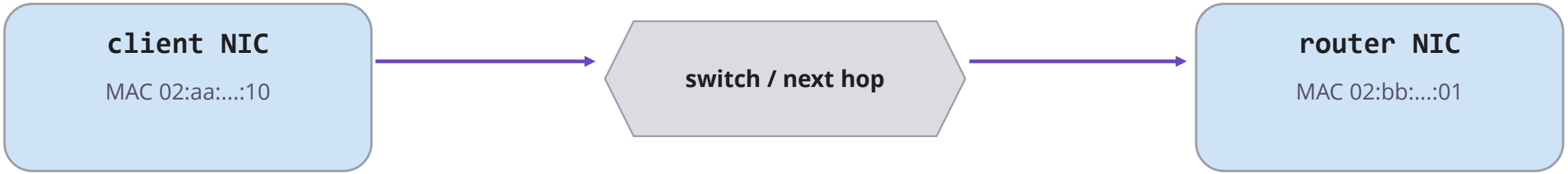
- A way to encode and detect bits on the medium

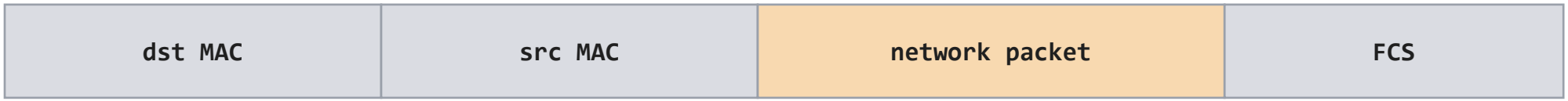
Practical checks

- Cable, link light, Wi-Fi signal, interface up/down

Layer 2: Data Link

Deliver one frame across one local link.





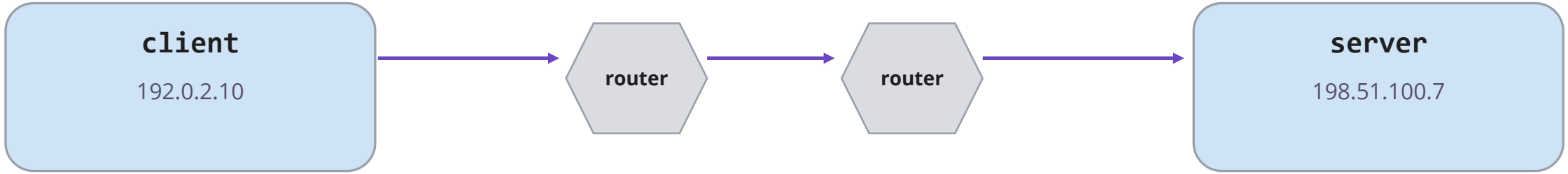
Ethernet and Wi-Fi

- Define frames and local interface addresses

Practical checks

- ip link, switches, ARP / neighbor table

Layer 3: Network



Each router inspects the destination IP and chooses a next hop.

Layer 4: Transport

simplified TCP segment

TCP provides

- Source and destination ports
- Ordered, reliable byte delivery
- Flow and congestion control

What the kernel can decide

- Port 3333 selects the listening uppercase service
- The endpoint pair keeps this client separate from other clients

Layer 5: Session

Organize a sustained conversation between applications.

1. **start** — authenticate / negotiate
2. **exchange** — many requests and responses
3. **recover** — resume or reconnect
4. **finish** — logout / orderly shutdown

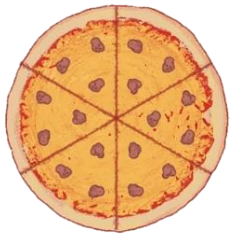
In real Internet software

❖ Often handled by application protocols and libraries

Examples

❖ Login state, RPC sessions, checkpoints, reconnect tokens

Layer 6: Presentation



Give shared meaning and representation to application data.

Structure

`object → JSON / protobuf`

Encoding

`text → UTF-8 bytes`

Compression

`bytes → fewer bytes`

Encryption

`plaintext → TLS records`

Agreement matters

- Both endpoints must interpret the same bytes the same way

Layer 7: Application

Define the messages and behavior users actually need.

```
// uppercase protocol
request:  hello, networks\n
response: HELLO, NETWORKS\n
```

The protocol decides

- What requests and responses mean
- Where each message ends
- What errors and valid sequences look like

HTTP	web
DNS	names
SMTP	mail
SSH	remote shell

Sockets

A local uppercase program

```
1  int main(int argc, char** argv) {
2    if (argc != 2) {
3      cerr << "usage: ./uppercase <word>" << endl;
4      return EXIT_FAILURE;
5    }
6    string line = argv[1];
7    for (char& ch : line) ch = toupper(ch);
8    cout << line << endl;
9    return EXIT_SUCCESS;
10 }
```

```
$ ./uppercase hello

HELLO
```

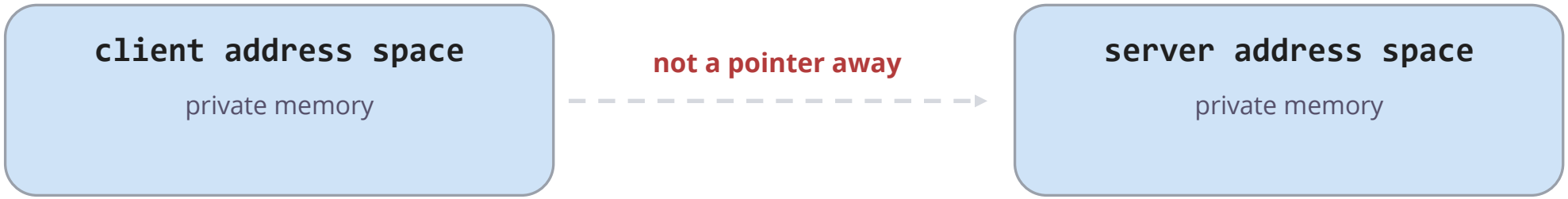
Nothing here is new

- Arguments in, standard output out
- One machine, one user, one run
- No network anywhere in this file

Another process wants to use it on the same machine

```
1  // client process
2  string message = "hello";
3  // server cannot read message
```

```
1  // server process
2  string reply;
3  // message is not in this address space
```



Another machine wants to use it

send them the source

then it is their program, not a service

read their input somehow

argv only reaches this machine

let them reach it over a network

one copy has to be used by many users

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.

You've already written most of the socket code!

Lecture 7: reading a file

```
1  int fd = open("poem.txt", O_RDONLY);
2  char buf[64];
3  ssize_t n = read(fd, buf, sizeof(buf));
4  write(STDOUT_FILENO, buf, n);
5  close(fd);
```

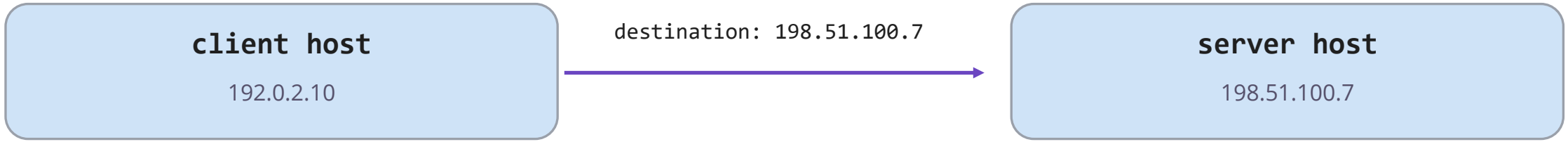
Today: reading from a socket

```
1  int fd = /* connect to a server */;
2  char buf[64];
3  ssize_t n = read(fd, buf, sizeof(buf));
4  write(STDOUT_FILENO, buf, n);
5  close(fd);
```

Only major difference is the input descriptor. Everything else is file I/O!

- read() can come up short. That is the readN loop from the POSIX I/O lecture.
- write() can come up short too. You wrote WriteAll for HW2. We reuse it byte for byte.
- 0 means the peer is done, -1 with EINTR means try again. Same rules, same errno.
- **Only line 1 is new.** The rest of today is about getting that descriptor.

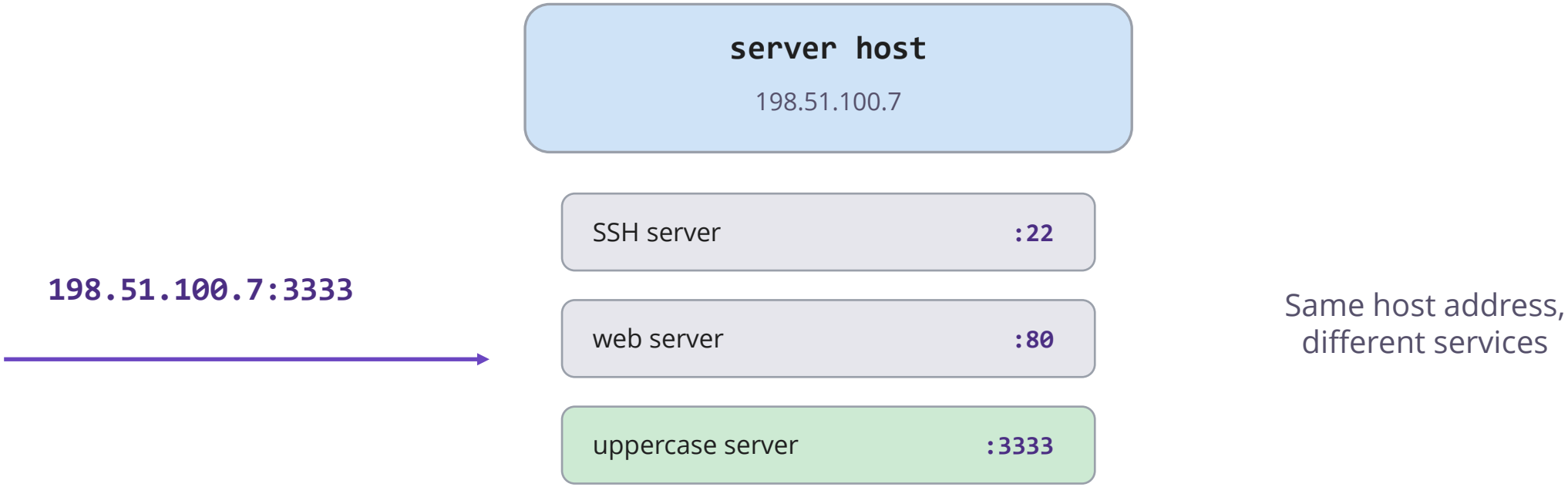
An IP address identifies a host interface



For now

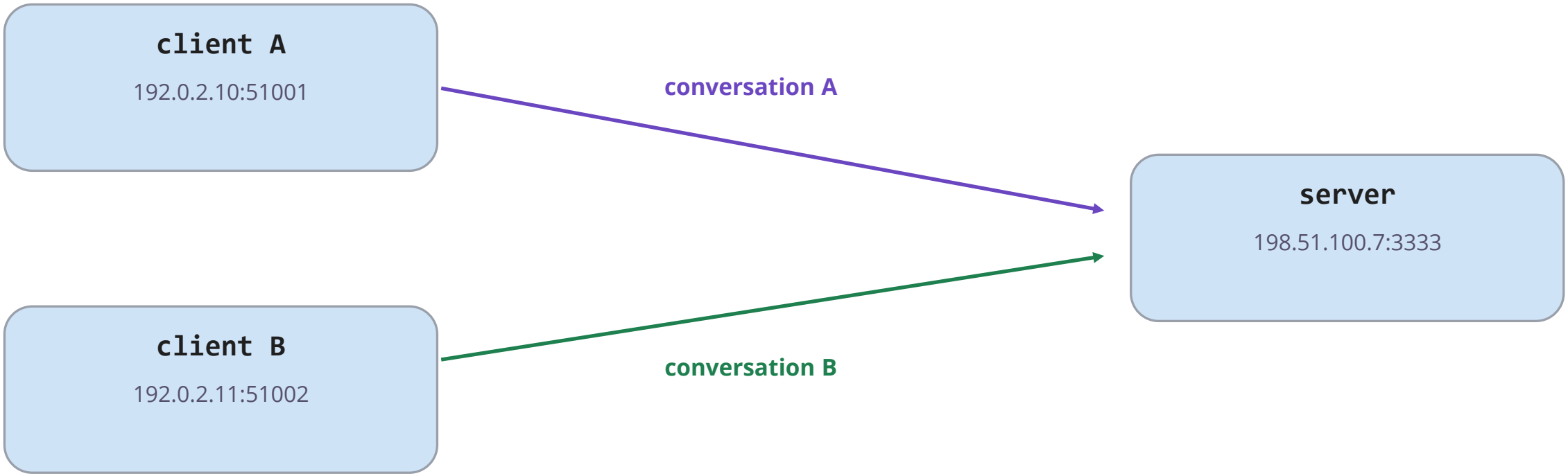
- Treat the address as the machine's network location
- DNS will translate a name such as attu.cs.washington.edu

A port selects one application on the host



Same host address, different services

A connection identifies one conversation



TCP preserves bytes and order

TCP's promise

- The receiver sees the bytes in the same order
- TCP retransmits lost data behind the socket API
- TCP does not preserve the sender's write() boundaries

How many read() calls receive one write()?

```
1  // sender
2  write(socket_fd, "hello\n", 6);
```

```
1  // receiver
2  ssize_t n = read(socket_fd, buf, 64);  // ?
```

A. always 1

B. 1 to 6

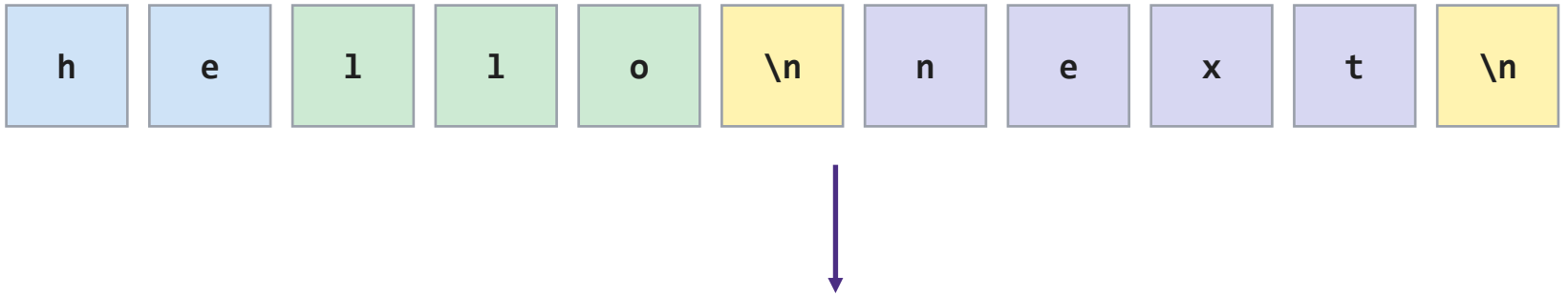
C. any positive count

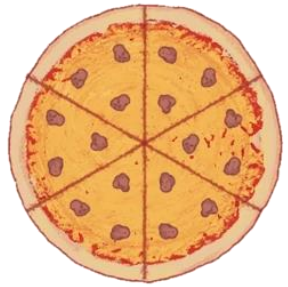
One write can arrive through several reads

```
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

The newline marks one complete message





`process "hello\n" now`

Our protocol

- Request: one line ending in newline
- Response: uppercase line ending in newline

Stream sockets vs datagram sockets

```
1  int fd = socket(AF_INET, SOCK_STREAM, 0);
2  // connection-oriented TCP byte stream
```

```
1  int fd = socket(AF_INET, SOCK_DGRAM, 0);
2  // connectionless UDP datagrams
```

Stream socket (TCP)

- One connected peer at each endpoint
- Reliable, ordered byte stream
- Used by SSH, HTTP connections, databases

Datagram socket (UDP)

- Each send is a separate datagram
- No built-in delivery or ordering guarantee
- Used by DNS, games, real-time media

Building the Server

Input is handled differently

what we wrote

```
1  string line = argv[1];
2  for (char& ch : line) ch = toupper(ch);
3  cout << line << endl;
```

what we want

```
1  string line = ReadLine(client_fd);
2  for (char& ch : line) ch = toupper(ch);
3  WriteAll(client_fd, line.data(), line.size());
```

- The actual work is identical. Uppercasing a string does not care where the string came from.
- Only the ends move. argv becomes a descriptor we read; stdout becomes a descriptor we write.
- Both new lines are Lecture 7 calls. ReadLine and WriteAll are the loops you wrote for HW2.
- **So the whole job is one question:** where does client_fd come from?

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

Which descriptor carries the client's bytes?

```
1  int fd1 = socket(...);
2  listen(fd1, SOMAXCONN);
3  int fd2 = accept(fd2, nullptr, nullptr);
4  ReadLine( ______ );   // ?
```

A. lfd1

B. fd2

C. either one works

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

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 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!

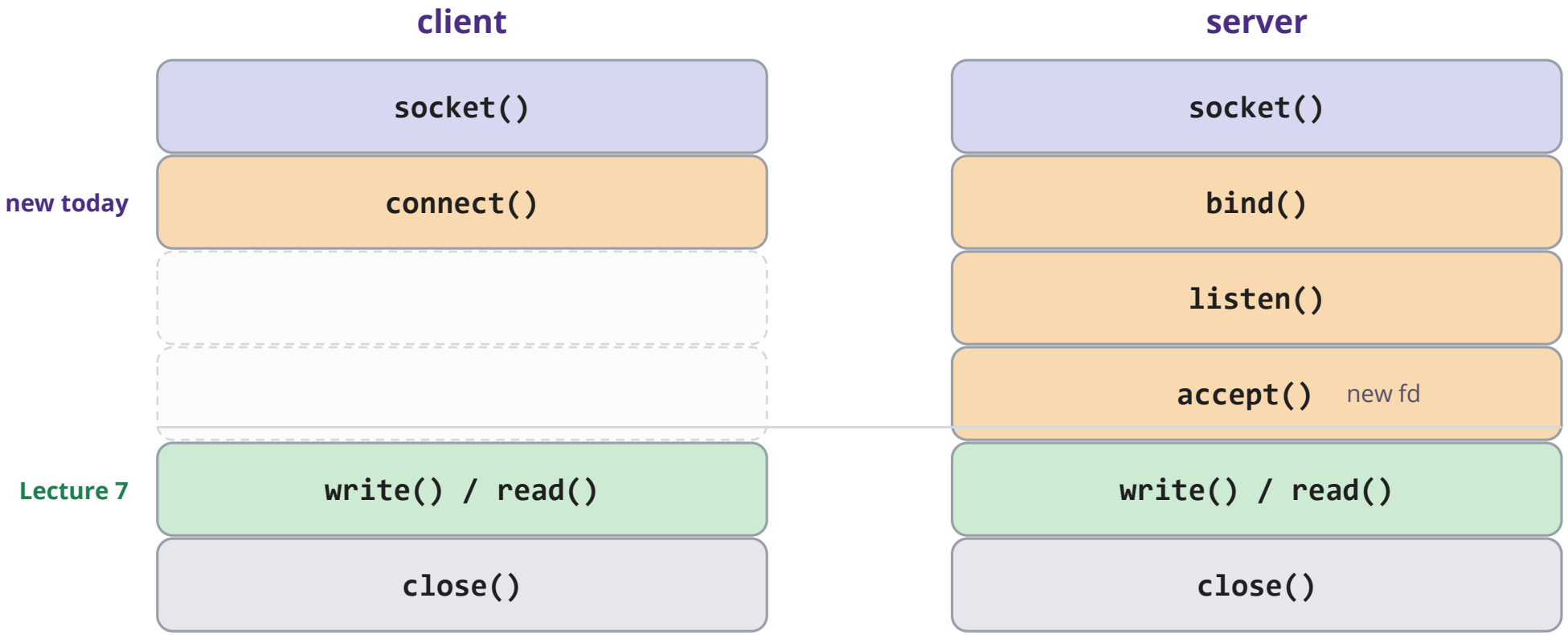
```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
addrinfo* results = nullptr;
getaddrinfo(host, port, &hints, &results);
int fd = socket(results->ai_family,
                results->ai_socktype,
                results->ai_protocol);
connect(fd, results->ai_addr, 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.

Reminders

Assessments:

- Exercise 7 due Wednesday at 11:59pm
- Homework 3 due Thursday at 11:59pm

General:

- How was the guest lecture?
- Midterm grades released
  - Review solutions and come to office hours
  - Submit a regrade requests if you think we made a mistake!

Questions?