

CS110 Lecture 21: Servers and HTTP

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

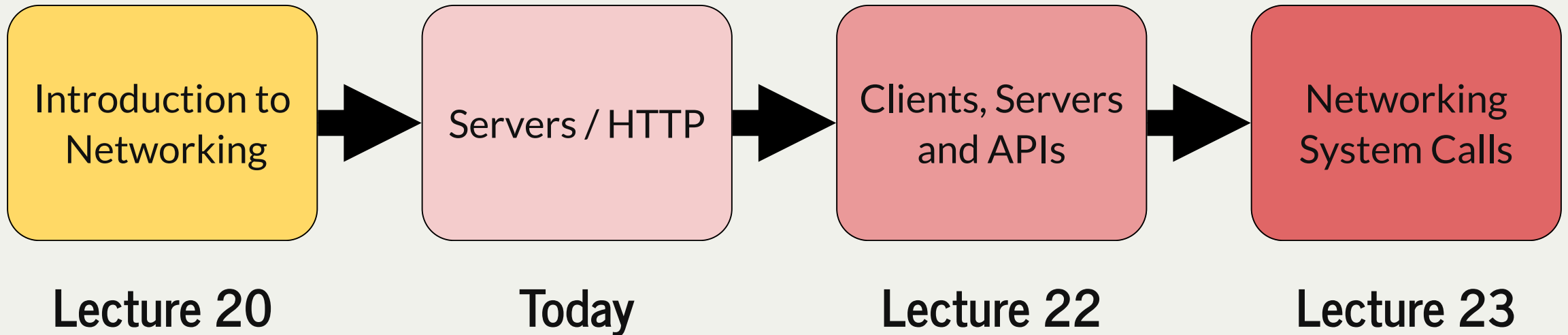
Instructors: Nick Troccoli and Jerry Cain



[PDF of this presentation](#)

CS110 Topic 4: How can we write programs that communicate over a network with other programs?

Learning About Networking



assign6: implement an HTTP Proxy that sits between a client device and a web server to monitor, block or modify web traffic.

Learning Goals

- Gain more practice with the client-server model
- Understand how to write our first server program
- Get exposure to the HTTP protocol for making requests and responses

Plan For Today

- **Recap:** Networking So Far
- **Recap:** Our first client program
- Our first server program
- Protocols and HTTP

Plan For Today

- Recap: Networking So Far
- Recap: Our first client program
- Our first server program
- Protocols and HTTP

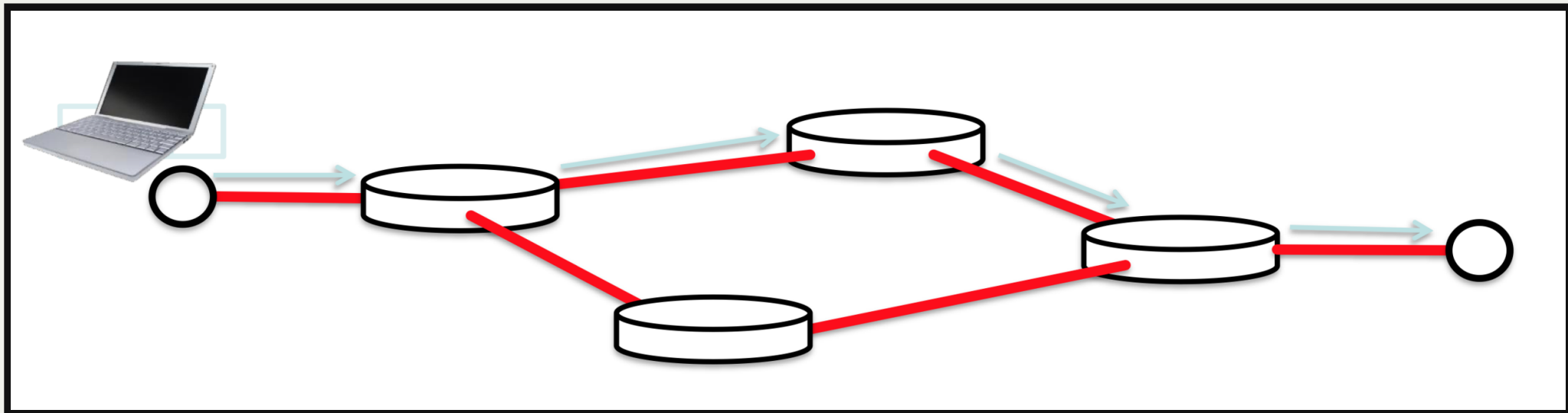
Networking So Far

- Networking allows us to write code to send and receive data to/from a program running on another machine.
- Most networked programs rely on a pattern called the "client-server model". **clients** send requests to **servers**, who listen for and respond to those requests
- We can send **any** arbitrary bytes over the network, but the client and server usually agree on a data format to use for requests and responses
- If you wish to connect to a program on another machine, you must specify both the **IP Address** of the machine and the **port number** assigned to that program
- **DNS** lets us look up the IP address for a given name
- You can open a connection to a program on another machine and you'll get back a **socket descriptor** number referring to your descriptor table. You can read/write to it and close it.
- A **socket** is the endpoint of a single connection over a port. *"socket descriptor" is to "port number" as "file descriptor" is to "filename"*

Other Networking Questions (and CS144!)

There's much more to networking than we have time to cover. We are focusing on the core ideas at the application level. Take CS144 if you're interested in learning more!

- how is data packaged up to be sent over the network? (packets)
- How does my data make it to the destination in one piece? (packet loss, TCP)
- How do packets get routed across the network from one machine to another?



(diagram from cs144 slides)

Other Networking Questions (and CS142!)

We're writing short client/server programs and focusing on the core aspects of how networked programs function. Take CS142 if you're interested in learning more about writing servers and web-based programs!

Plan For Today

- Recap: Networking So Far
- Recap: Our first client program
- Our first server program
- Protocols and HTTP

Our First Client Program

- Let's write our first program that sends a request to a server!
- Let's say I am running a server on **myth64.stanford.edu**, port **12345** that can tell you the current time.
- Whenever a client connects to it, the server sends back the time as text. The client doesn't need to send any data.
- Let's write a client program that connects and prints out what the server says.

New CS110 helper function to connect to a server:

```
// Opens a connection to a server (returns kClientSocketError on error)
int createClientSocket(const string& host, unsigned short port);
```

(Later on, we will learn how to implement createClientSocket!)

Our First Client Program

I am running a server on `myth64.stanford.edu`, port `12345` that can tell you the current time. Whenever a client connects to it, the server sends back the time as text. This client program connects to that server and prints the response using `sockbuf/iosockstream`.

```
1 int main(int argc, char *argv[]) {
2     // Open a connection to the server
3     int socketDescriptor = createClientSocket("myth64.stanford.edu", 12345);
4
5     // Read in the data from the server (sockbuf descriptor closes descriptor)
6     sockbuf socketBuffer(socketDescriptor);
7     iosockstream socketStream(&socketBuffer);
8     string timeline;
9     getline(socketStream, timeline);
10
11    // Print the data from the server
12    cout << timeline << endl;
13
14    return 0;
15 }
```

```
1 myth$ ./time-client
2 Fri Feb 25 08:15:22 2022
3 myth$
```

Key idea: there is no code in the client that is itself calculating the current time.

All that logic is in the server that the client connects to! Essentially "remote function call and return".

Our First Client Program

sockbuf/iosockstream let us avoid calling read/write directly, which is more cumbersome and is C-level instead of C++-level:

```
1 int main(int argc, char *argv[]) {
2     // Open a connection to the server
3     int socketDescriptor = createClientSocket("myth64.stanford.edu", 12345);
4
5     // Read in the data from the server (assumed to be at most 1024 byte string)
6     char buf[1024];
7     size_t bytes_read = 0;
8     while (true) {
9         size_t read_this_time = read(socketDescriptor, buf + bytes_read, sizeof(buf) - bytes_read);
10        if (read_this_time == 0) break;
11        bytes_read += read_this_time;
12    }
13    buf[bytes_read] = '\0';
14    close(socketDescriptor);
15
16    // print the data from the server
17    cout << buf << flush;
18    return 0;
19 }
```



time-client-descriptor.cc

Plan For Today

- Recap: Networking So Far
- Recap: Our first client program
- Our first server program
- Protocols and HTTP

Our First Server Program

- Let's write our first program that can respond to incoming requests!
- **Example:** I want to run a server on **myth64.stanford.edu**, port **12345** that can tell you the current time
- Whenever a client connects to it, the server sends back the time as text. The client doesn't need to send any data.

New CS110 helper function to create a socket descriptor to listen for incoming connections:

```
// Creates a socket to listen for incoming requests (returns kServerSocketFailure on error)
int createServerSocket(unsigned short port, int backlog = kDefaultBacklog);
```

(Later on, we will learn how to implement createServerSocket!)

Server Sockets

- Server sockets work slightly differently than regular file descriptors because we are *continually listening for incoming connections*.
- To actually wait for an incoming connection, we must call the **accept** function, which returns a descriptor we can use to communicate with that client. We call this in a loop to handle many incoming connections.

```
// Waits for an incoming connection and returns a descriptor for that connection
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- The first parameter is the server socket. The rest are for getting information about the client that is connecting, but we will pass in NULL for now.
- Analogy: **server socket** is main operator fielding all calls, **accept** transfers you to an agent to actually take your call.

Our First Server Program

This code runs a server on port **12345** that can tell you the current time. Whenever a client connects to it, the server sends back the time as text.

```
1 int main(int argc, char *argv[]) {
2     // Create a server socket we can use to listen for incoming requests
3     int serverSocket = createServerSocket(12345);
4
5     while (true) {
6         // Wait for an incoming client connection and establish a descriptor for it
7         int clientDescriptor = accept(serverSocket, NULL, NULL);
8
9         // Make a string of the current date and time and send it to the client
10        string dateTime = getCurrentDateTime();
11        sockbuf socketBuffer(clientDescriptor); // destructor closes socket
12        iosockstream socketStream(&socketBuffer);
13        socketStream << dateTime << endl;
14    }
15
16    close(serverSocket);
17
18    return 0;
19 }
```



`time-server-sequential.cc`

Our First Server Program

The function that actually creates the time string is not important - it could be any kind of data. The key takeaway is how the server listens for and responds to client connections.

```
1 // This function returns a string representation of the current date and time.
2 static string getCurrentDateTime() {
3     time_t rawtime;
4     time(&rawtime);
5     struct tm tm;
6     gmtime_r(&rawtime, &tm);
7     char timestr[128]; // more than big enough
8     /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", &tm);
9     return timestr;
10 }
```

Our First Server Program

Problem: servers may have many incoming connections at once. We need to be able to handle connections concurrently!

- **Demo:** let's see what happens when many requests come in at once.
- **Solution:** we can use a *thread pool* to handle connections concurrently; every time we receive a connection via **accept**, we will add a task to a thread pool to respond to the connection request.

Our First (Concurrent) Server Program

This server adds tasks to a thread pool to concurrently respond to client connections.

```
1 int main(int argc, char *argv[]) {
2     // Create a server socket we can use to listen for incoming requests
3     int serverSocket = createServerSocket(12345);
4
5     ThreadPool pool(kNumThreads);
6     while (true) {
7         // Wait for an incoming client connection and establish a descriptor for it
8         int clientDescriptor = accept(serverSocket, NULL, NULL);
9
10        // Add a task to make a string of the current date and time and send it to the client
11        pool.schedule([clientDescriptor]() {
12            string dateTime = getCurrentDateTime();
13            sockbuf socketBuffer(clientDescriptor); // destructor closes socket
14            iosockstream socketStream(&socketBuffer);
15            socketStream << dateTime << endl;
16        });
17    }
18
19    close(serverSocket);
20    return 0;
21 }
```



`time-server-concurrent.cc`

Takeaways: Clients and Servers

- **A client program** can open a connection to a server, send request information, and receive a response. We use `createClientSocket` to get a descriptor to read/write with.
- **A server program** can listen on a port for incoming client requests, and respond to them. We use `createServerSocket` to get a descriptor to listen with, and `accept()` to accept incoming client connections and get descriptors to read/write with.
- Servers greatly benefit from multithreading to parallelize handling incoming requests!

Plan For Today

- Recap: Networking So Far
- Recap: Our first client program
- Our first server program
- Protocols and HTTP

Data Protocols

Our time server chose to send a raw single-line string response to a client. A client connecting must be aware of this to know how to handle / use the response data.

Key idea: a client and server must agree on the format of the data being sent back and forth so they know what to send and how to parse the response.

- A **protocol** is a specification dictating how two computers should should converse. By respecting a protocol, both the client and server know they'll understand each other.

HTTP ("HyperText Transfer Protocol") is the predominant protocol for Internet requests and responses (e.g. webpages, web resources, web APIs).

HTTP

What happens when you type a URL into your web browser?

- Your browser looks up the IP address of the site you entered
- Your browser sends an HTTP request to that address on port 80 to get the webpage
- The response payload data is usually text (such as HTML - HyperText Markup Language) or other content that the browser can display

Note: a more secure version of HTTP, called HTTPS, is predominant today and encrypts requests/responses. Most conversations happen over HTTPS, but we're focusing just on HTTP.

HTTP Request Format

```
GET / HTTP/1.0  
Host: www.google.com  
...  
[ BLANK LINE ]
```



The first line is the request line. It specifies general information about the kind of request and the protocol version. 3 components:

- request type ("verb" or "method")
- request path
- request protocol version

HTTP Request Format

GET / HTTP / 1.0



"verb" or "method": what kind of request are we making?

- I wish to fetch a resource (GET)
- I wish to upload some new data (POST / PUT)
- I wish to get a preview of information about a resource (HEAD)

HTTP Request Format

GET / HTTP / 1.0



"path": what server resource am I referring to?

- just the component after the host name

HTTP Request Format

GET / HTTP / 1.0



"HTTP protocol version": what version of HTTP
am I speaking?

HTTP Request Format

```
GET / HTTP/1.0
Host: www.google.com
...
[ BLANK LINE ]
```



The second and onwards lines are each a **header** included to provide more information. They are key-value pairs.

Examples:

- Host ("what host am I sending this to?")
- Content-Type ("what type of content am I uploading?")
- User-Agent ("what kind of user program sent this request?")
- Cookie ("what cookie does the user have for speaking to this server?")

HTTP Request Format

```
GET / HTTP/1.0  
Host: www.google.com  
...  
[ BLANK LINE ]
```



The request ends with a blank line.

HTTP Request Format

```
GET / HTTP/1.0  
Host: www.google.com  
...  
[BLANK LINE]  
{request body}
```

Some requests (like POST) that are uploading data have a request **body** after this blank line.

HTTP Response Format

```
HTTP/1.0 200 OK  
Content-Type: text/html  
[BLANK LINE]  
{response body}
```

The first line is the status line. It specifies general information about how the request was handled and the protocol version. 2 components:

- response protocol version
- response status code

HTTP Response Format

HTTP / 1.0 200 OK



"HTTP protocol version": what version of HTTP
am I speaking?

HTTP Response Format

HTTP / 1.0 200 OK



How did things go? Examples:

- A-ok! (20X)
- What you're looking for is somewhere else (30X)
- you did something wrong (40X)
- we did something wrong (50X)

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

HTTP Response Format

HTTP / 1.0 200 OK




Some humorous status codes:

- 418 (I'm a teapot) - "Any attempt to brew coffee with a teapot should result in the error code "418 I'm a teapot". The resulting entity body **MAY** be short and stout."
- 451 (unavailable for legal reasons)

HTTP Response Format

```
HTTP/1.0 200 OK
Content-Type: text/html
[BLANK LINE]
{response body}
```




The second and onwards lines are each a **header** included to provide more information. They are key-value pairs.

Examples:

- Content-Type ("what type of content am I including?")
- Content-Length ("how much content am I including?")
- Set-Cookie ("here's a cookie you should remember for future requests")

HTTP Response Format

```
HTTP/1.0 200 OK
Content-Type: text/html
[BLANK LINE]
{response body}
```



Following a blank line, there is the response body ("payload") containing data the server sent back. E.g. HTML to display.

Demo: HTTP Requests/Responses using
your browser and **telnet**

Recap

- **Recap:** Networking So Far
- **Recap:** Our first client program
- Our first server program
- Protocols and HTTP

Next time: More HTTP and servers