CS110 Lecture 20: Introduction to Networking

CS110: Principles of Computer Systems

Winter 2021-2022

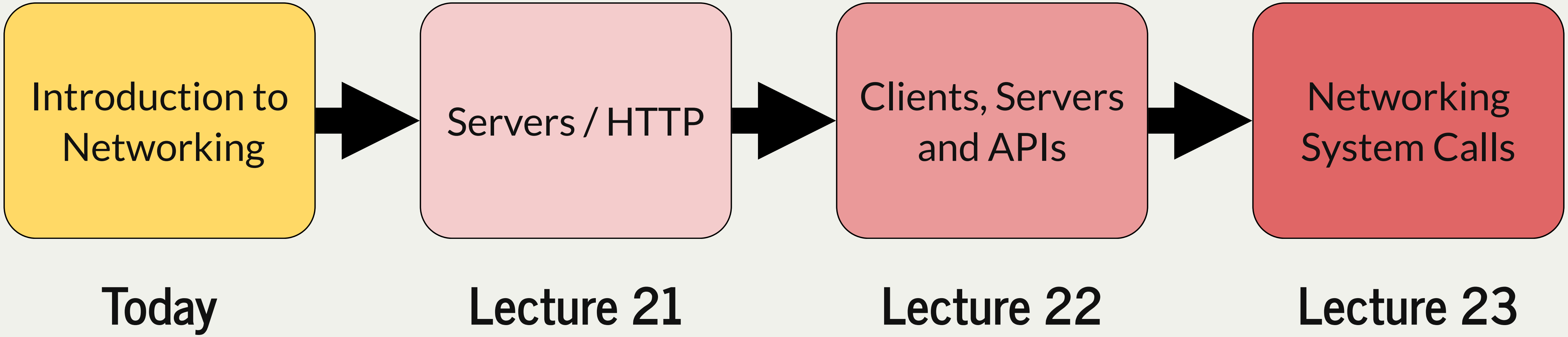
Stanford University

Instructors: Nick Troccoli and Jerry Cain

PDF of this presentation

<u>CS110 Topic 4:</u> 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

- Understand how networking enables two programs on separate machines to communicate
- Learn about the client-server model and how client and server programs interact
- Understand how to write our first client program

Plan For Today

- Networking Overview
- IP Addresses, DNS Lookup and Ports
- Sockets and Descriptors
- Our first client program

Plan For Today

- **Networking Overview**
- IP Addresses, DNS Lookup and Ports
- Sockets and Descriptors
- Our first client program

Networking Overview

- We have learned how to write programs that can communicate with other programs via mechanisms like signals and pipes.
- However, the communicating programs must both be running on the same machine.
- Networking allows us to write code to send and receive data to/from a program running on another machine.
- Many new questions, such as:
  - how does the data get there?
  - what functions do we use to send/receive data?

Networking Patterns

- Most networked programs rely on a pattern called the "client-server model"
- This refers to two program "roles": **clients** and **servers**
- **clients** send requests to **servers**, who respond to those requests
  - e.g. YouTube app (client) sends requests to the YouTube servers for what content to display
  - e.g. Web browser (client) sends requests to the server at a URL for what content to display
- A **server** continually listens for incoming requests and sends responses back ("running a phone call center")
- A **client** sends a request to a server and does something with the response ("making a call")
- We will learn how to write both client and server programs.
  - on assign6, your proxy will act as both a client *and* a server!

Sending/Receiving Data

- We can send **any** arbitrary bytes over the network
- The client and server usually agree on a data format to use for requests and responses
- Many *data protocols* like HTTP (internet), IMAP (email), others

But how does data actually *get* from one machine to another?

Plan For Today

- Networking Overview
- **IP Addresses, DNS Lookup and Ports**
- Sockets and Descriptors
- Our first client program

IP Addresses

- To send data to another program, we need to know the **IP Address** ("Internet Protocol Address") of its machine
- Every computer on a network has a unique **IP Address** - e.g. 171.67.215.200
- A traditional IPv4 ("version 4") address is 4 bytes long: 4 numbers from 0-255 separated by periods
- **Problem:** there aren't enough IPv4 addresses to go around anymore! Exhausted in the 2010s
- Now there is a new version, IPv6, supporting more values with 16-byte addresses
- An IPv6 address is 8 groups of 4 hexadecimal digits - e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334

DNS Lookup

- **Problem**: it's hard for us to remember IP addresses for different machines!
- **Solution**: assign human-readable names (e.g. "google.com") to different machines, and translate those names to IP addresses.
- The **Domain Name System** (DNS) is what translates names to IP addresses
  - A collection of *decentralized* and *hierarchical* servers that we can contact to perform translation
  - *decentralized:* many DNS servers handling lookup all over
  - *hierarchical*: translation performed in steps: e.g for looking up web.stanford.edu:
    - query an .edu root server for IP address of a stanford.edu name server
    - query stanford.edu name server for IP address of web.stanford.edu
- Form of **name resolution**, like inode numbers and filename lookup in filesystems!

Digging For Treasure

- Your computer performs DNS lookups frequently on your behalf - e.g. when you want to visit a website in your browser.
- For fun, we can view DNS servers using the dig command:
  - **where are the edu nameservers?** "dig -t NS +noall +answer edu"
  - **the stanford.edu nameservers?** "dig -t NS +noall +answer stanford.edu"
  - **where is web.stanford.edu?** "dig -t A +noall +answer web.stanford.edu"

IP Addresses and Ports

- **IP addresses** let us identify the machine we want to communicate with
- **DNS** lets us look up the IP address for a given name
- **Another problem:** what if we want to run multiple networked programs per machine?
  - Limiting if we can *only* e.g. ssh *or* have a web server on one machine
- **Solution:** every networked program running is assigned a unique *port number*
  - mail analogy: IP address = Stanford dorm, port number = dorm room number
- When 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
- port numbers are like "virtual process IDs"
- You can see some of the ports a computer is listening to with "netstat -plnt"
- How do we remember port numbers? What if they can change each time we run?

IP Addresses and Ports

- **Key Idea:** establish standard port numbers for some common types of programs
  - HTTP (internet traffic): port 80
  - SSH: port 22
  - DNS: port 53
  - https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
  - For your own programs, generally try to stay away from established port numbers, but otherwise, ports are up for grabs to any program that wants one.
- Your web browser takes an entered URL, uses DNS to look up the IP address, and sends a request to that IP address, port 80 for the webpage you requested.
- A **server** program will run on a machine and be assigned a port number
- A **client** program wishing to connect to that server must send a request to that port number at that IP address.

So how can we write code that communicates with another program?

Plan For Today

- Networking Overview
- IP Addresses, DNS Lookup and Ports
- **Sockets and Descriptors**
- Our first client program

Sockets and Descriptors

- Linux uses the same **descriptor** abstraction for network connections as it does for files!
- 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
- A **socket** is the endpoint of a single connection over a port. It is represented as a descriptor we can read from/write to.
- You can read to / write from that descriptor to communicate
- You close the descriptor when you're done
- **Like a pipe, but with only *one* descriptor, not two:** network communication is bidirectional, but usually the client and server speak one a time, not simultaneously.
- *"socket descriptor" is to "port number" as "file descriptor" is to "filename"*

Key Idea: networking is remote function call and return.

Plan For Today

- Networking Overview
- IP Addresses, DNS Lookup and Ports
- Sockets and Descriptors
- **Our first client program**

Our First Client Program

- Let's write our first program that sends a request to a server!
- **Example:** 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.

New 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.

```
int main(int argc, char *argv[]) {
  // Open a connection to the server
  int socketDescriptor = createClientSocket("myth64.stanford.edu", 12345);

  // Read in the data from the server (assumed to be at most 1024 byte string)
  char buf[1024];
  size_t bytes_read = 0;
  while (true) {
    size_t read_this_time = read(socketDescriptor, buf + bytes_read, sizeof(buf) - bytes_read);
    if (read_this_time == 0) break;
    bytes_read += read_this_time;
  }
  buf[bytes_read] = '\0';
  close(socketDescriptor);

  // print the data from the server
  cout << buf << flush;
  return 0;
}
```

time-client-descriptor.cc

Client Sockets

Client sockets work similarly to regular file descriptors - we open one, read from/write to it, and close it.

```
int main(int argc, char *argv[]) {
  // Open a connection to the server
  int socketDescriptor = createClientSocket("myth64.stanford.edu", 12345);

  // Read in the data from the server (assumed to be at most 1024 byte string)
  char buf[1024];
  size_t bytes_read = 0;
  while (true) {
    size_t read_this_time = read(socketDescriptor, buf + bytes_read, sizeof(buf) - bytes_read);
    if (read_this_time == 0) break;
    bytes_read += read_this_time;
  }
  buf[bytes_read] = '\0';
  close(socketDescriptor);

  // print the data from the server
  cout << buf << flush;
  return 0;
}
```

time-client-descriptor.cc

Using Socket Descriptors

Using **read/write** is cumbersome with socket descriptors. The socket++ library provides a type **iosockstream** that let us wrap a socket descriptor in a *stream* (so that we can read/write like we do with **cout**):

```
static string readLineFromSocket(int socketDescriptor) {
  sockbuf socketBuffer(socketDescriptor);
  iosockstream socketStream(&socketBuffer);
  string timeline;
  getline(socketStream, timeline);
  return timeline;
} // sockbuf destructor closes client
```

Our First Client Program

Here is a version of the same client program using **sockbuf** and **iosockstream** instead of **read:**

```
int main(int argc, char *argv[]) {
  // Open a connection to the server
  int socketDescriptor = createClientSocket("myth64.stanford.edu", 12345);

  // Read in the data from the server (sockbuf descructor closes descriptor)
  sockbuf socketBuffer(socketDescriptor);
  iosockstream socketStream(&socketBuffer);
  string timeline;
  getline(socketStream, timeline);

  // Print the data from the server
  cout << timeline << endl;

  return 0;
}
```

time-client.cc

Recap

- Networking Overview
- IP Addresses, DNS Lookup and Ports
- Sockets and Descriptors
- Our first client program

Next time: more about servers, data formats and protocols