CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 19: Introduction to Networking

- Every computer on a network has a unique IP address that identifies it on the network.
    - A traditional IP address is stored as four bytes and generally presented as a dotted quad: four numbers between 0 and 255, inclusive, all separated by periods.
    - One example of an IP address is **192.168.1.1**.
    - **300.704.256.1** is not a valid IP address, since the numbers 300, 704, and 256 are greater than 255 and can't be easily stored a single bytes.
- Humans aren't very at remembering strings of numbers like IP addresses.
    - The Domain Name System (DNS) translates human-friendly hostnames (also called domain names) into IP addresses.
    - You can issue a DNS request for the IP address of **web.stanford.edu**, and you'll hear back that it has an IP address of **171.67.215.200**.
- Throwback: in filesystems, if you want the inumber for **/usr/class/cs110/WWW**, you first ask the root directory for **usr**'s inumber, then ask **/usr** for **class**'s inumber, then ask **/usr/class** for **cs110**'s inumber, then ask **/usr/class/cs110** for **WWW**'s inumber.
- A very similar process exists in DNS!
    - A set of root name servers exist to answer DNS lookups for each domain name suffix—e.g. **.com** has a set of root name servers, as do **.org**, **.edu**, **.gov**, and so forth.
    - When you want to look up **web.stanford.edu**, the DNS request queries the **.edu** root server for the IP address of the **stanford.edu** name server, which itself can be queried for the IP address of **web.stanford.edu**.

Lecture 19: Introduction to Networking

- You can play around with DNS lookups by running **dig**.
    - You don't have to be familiar with the specifics of DNS and you don't have to know what all the different kinds of records mean, but you should have a general familiarity with what it does and how it works.

```
poohbear@myth64:~$ dig -t NS +noall +answer edu # where are the edu nameservers?
edu.                    6691    IN      NS      k.edu-servers.net.
edu.                    6691    IN      NS      i.edu-servers.net.
edu.                    6691    IN      NS      c.edu-servers.net.
edu.                    6691    IN      NS      h.edu-servers.net.
edu.                    6691    IN      NS      d.edu-servers.net.
edu.                    6691    IN      NS      b.edu-servers.net.
edu.                    6691    IN      NS      a.edu-servers.net.
edu.                    6691    IN      NS      f.edu-servers.net.
edu.                    6691    IN      NS      m.edu-servers.net.
edu.                    6691    IN      NS      l.edu-servers.net.
edu.                    6691    IN      NS      e.edu-servers.net.
edu.                    6691    IN      NS      j.edu-servers.net.
edu.                    6691    IN      NS      g.edu-servers.net.
poohbear@myth64:~$ dig -t NS +noall +answer stanford.edu # the stanford.edu nameservers?
stanford.edu.           84428   IN      NS      ns6.dnsmadeeasy.com.
stanford.edu.           84428   IN      NS      argus.stanford.edu.
stanford.edu.           84428   IN      NS      avallone.stanford.edu.
stanford.edu.           84428   IN      NS      atalante.stanford.edu.
stanford.edu.           84428   IN      NS      ns7.dnsmadeeasy.com.
stanford.edu.           84428   IN      NS      ns5.dnsmadeeasy.com.
poohbear@myth64:~$ dig -t A +noall +answer web.stanford.edu  # Where is web.stanford.edu?
web.stanford.edu.       431     IN      A       171.67.215.200
```

Lecture 19: Introduction to Networking

- We want to run a server on a computer. People should be able to connect to this computer over a network and interact with your server.

- When you start a server, it binds to a specific port number between 0 and 65535.
    - You might want to run several servers on any one machine—e.g. a web server, an SSH server, a file server—and each server needs to bind to a different port number.
    - Port numbers for well-known services are generally fixed. SSH servers usually listen for connections on port 22, web servers usually listen for connections on port 80, and mail servers often listen on port 25.

- You can think of an IP address as the address of a Stanford dorm and think of port numbers as room numbers within the dorm.
    - When a client connects to your server, they "go to your dorm" and then "go to your room" once inside the dorm.
    - When you ssh to **myth54.stanford.edu**, you're interacting with the server running on 171.64.15.19 and bound to port 22.

- I like to think of a port number as a *virtual process ID* the server associates with the true pid of the server application.

Lecture 19: Introduction to Networking

- A lot goes into establishing and maintaining a network connection.
  - Virtually all of these details are invisible to us and managed by the OS.
  - For CS110 purposes, a network connection is little more than a bidirectional pipe bridging two processes.

- Just for fun, log into a myth machine and run the following:

```
poohbear~$ echo $HOSTNAME # note this, you need it for the telnet command below
myth64.stanford.edu
poohbear~$ nc -l 12345
```

  - If you get an error saying "Address already in use", just choose some other five-digit number less than 60000.
- In a different terminal window, log into another myth (or non-myth machine if you're on campus) and run the following:

```
Jerrys-MacBook-Pro:~ jerry$ nc myth64.stanford.edu 12345
```

- Anything you type in either window shows up in the other (after hitting) return.
- **nc** (short for **netcat**) is the simplest of network tools that shows hostnames and port numbers really play a role in information exchange between networked computers.

Lecture 19: Introduction to Networking

- A genuinely interesting point of view: networking is little more than remote function call and return.
- We've seen many new function-call-and-return models this quarter:
  - Traditional function call and return as covered in CS107.
  - System call call and return as covered during Week 2 of the CS110.
  - The **subprocess** model, where functionality of interest is packaged in executable form instead of library form.
  - Networks and remote procedure invocation, where functionality of interest resides on a server accessible to any client that knows its IP address (and possesses any necessary credentials needed to interact with it).
- Each of these models comes with varying levels of modularity, security, and stability.
  - In general, the network/ RPC model is the most modular, and at least as secure as the system call. Stability, of course, varies.
- Consider https://www.google.com/search?q=cats&tbm=isch.
  - What's the function name?
  - What are the parameters?
  - And what's the return value?

Lecture 19: Introduction to Networking

- Let's create our first server (entire program here):

```
int main(int argc, char *argv[]) {
  int server = createServerSocket(12345);
  while (true) {
    int client = accept(server, NULL, NULL); // the two NULLs could instead be used to
                                             // surface the IP address of the client
    publishTime(client);
  }
  return 0;
}
```

- **accept** (found in **sys/socket.h**) returns a read-write (!!) descriptor. This descriptor is one end of a bidirectional pipe bridging two processes—on different machines!

Lecture 19: Introduction to Networking

- The **`publishTime`** function is straightforward:

```
static void publishTime(int client) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timestr[128]; // more than big enough
  /* size_t len = */ strftime(timestr, sizeof(timestr), "%c\n", ptm);
  size_t numBytesWritten = 0, numBytesToWrite = strlen(timestr);
  while (numBytesWritten < numBytesToWrite) {
    numBytesWritten += write(client,
                             timestr + numBytesWritten,
                             numBytesToWrite - numBytesWritten);
  }
  close(client);
}
```

- The first five lines here produce the full time string that should be published.
  - Let these five lines represent more generally the server-side computation needed for the service to produce output.
  - Here,the payload is the current time, but it could have been a static HTML page, a Google search result, an image, or a movie on Netflix.
- The remaining lines publish the time string to the client socket using the raw, low-level I/O we've seen before.

Lecture 19: Introduction to Networking

- Note that the **`while`** loop for writing bytes is a bit more important now that we are networking: we are more likely to need to write multiple times.
    - A socket descriptor is attached to a network driver with a finite amount of space
    - That means **`write`**'s return value could very well be less than what was supplied by the third argument. For example, you may try to publish the contents of a 62GB movie and be told that only 4MB went through.
- Ideally, we'd rely on either C streams (e.g. the **`FILE *`**) or C++ streams (e.g. the **`iostream`** class hierarchy) to layer over data buffers and manage the **`while`** loop around exposed **`write`** calls for us.
- Fortunately, there's a stable, easy-to-use third-party library—one called **`socket++`**—that provides precisely this.
    - **`socket++`** provides iostream subclasses that respond to **`operator<<`**, **`operator>>`**, **`getline`**, **`endl`**, and so forth, just as **`cin`**, **`cout`**, and file streams do.
    - We are going to operate as if this third-party library was just part of standard C++.
- The next slide shows a prettier version of **`publishTime`**.

Lecture 19: Introduction to Networking

- Here's the new implementation of **`publishTime`**:

```
static void publishTime(int client) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timestr[128]; // more than big enough
  /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", ptm);
  sockbuf sb(client);
  iosockstream ss(&sb);
  ss << timestr << endl;
} // sockbuf destructor closes client
```

- We rely on the same C library functions to generate the time string.
- This time, however, we insert that string into an **`iosockstream`** that itself layers over the client socket.
- Note that the intermediary **`sockbuf`** class takes ownership of the socket and closes it when its destructor is called.

Lecture 19: Introduction to Networking

- You've already seen two examples—the **`myth-buster`** and Assignment 5's **`aggregate`**—where multithreading can significantly improve the performance of networked applications.
- Our time server can benefit from multithreading as well. The work a server needs to do in order to meet the client's request might be time consuming—so time consuming, in fact, that the server is slow to iterate and accept new client connections.
- As soon as **`accept`** returns a socket descriptor, spawn a child thread—or reuse an existing one within a **`ThreadPool`**—to get any intense, time consuming computation off of the main thread. The child thread can make use of a second processor or a second core, and the main thread can quickly move on to its next **`accept`** call.
- Here's a new version of our time server, which uses a **`ThreadPool`** (you're implementing one for **assign5**) to get the computation off the main thread asap.

```
int main(int argc, char *argv[]) {
    int server = createServerSocket(12345);
    ThreadPool pool(4);
    while (true) {
        int client = accept(server, NULL, NULL); // the two NULLs could instead be used
                                                 // to surface the IP address of the client
        pool.schedule([client] { publishTime(client); });
    }
    return 0;
}
```

Lecture 13: Networks and Clients

- Implementing your first client! (code here)
    - The protocol—that's the set of rules both client and server must follow if they're to speak with one another—is very simple.
        - The client connects to a specific server and port number. The server responds to the connection by publishing the current time into its own end of the connection and then hanging up. The client ingests the single line of text and then itself hangs up.

```
int main(int argc, char *argv[]) {
    int clientSocket = createClientSocket("myth64.stanford.edu", 12345);
    assert(clientSocket >= 0);
    sockbuf sb(clientSocket);
    iosockstream ss(&sb);
    string timeline;
    getline(ss, timeline);
    cout << timeline << endl;
    return 0;
}
```

        - We'll soon discuss the implementation of **`createClientSocket`**. For now, view it as a built-in that sets up a bidirectional pipe between a client and a server running on the specified host (e.g. **`myth64`**) and bound to the specified port number (e.g. 12345).