Lecture 13: Introduction to Networking

- Networking is simply communicating between two computers connected on a network. You can actually set up a network connection on a single computer, as well.
- A network requires one computer to act as the *server*, waiting patiently for an incoming connection from another computer, the *client*.
- Server-side applications set up a *socket* that listens to a particular port. The server socket is an integer identifier associated with a local IP address, and a the port number is a 16-bit integer with up to 65535 allowable ports.
    - You can think of a port number as a *virtual process ID* the host associates with the true pid of the server application.
    - You can see some of the ports your machine is monitoring using **`netstat`**:

Chris Gregg contributed to these slides.

```
myth66:/usr/class/cs110/lecture-examples/networking$ netstat -plnt
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:10050           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:587           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp6       0      0 :::10050                :::*                    LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
myth66:/usr/class/cs110/lecture-examples/networking$
```

Lecture 13: Introduction to Networking

```
myth66:/usr/class/cs110/lecture-examples/networking$ netstat -plnt
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:10050           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:587           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp6       0      0 :::10050                :::*                    LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
tcp6       0      0 ::1:631                 :::*                    LISTEN      -
myth66:/usr/class/cs110/lecture-examples/networking$
```

- Some common ports are listed above. You can see a full list here and here.
  - Ports 25 and 587 are the SMTP (Simple Mail Transfer Protocol), for sending and receiving email.
  - Port 53 is the DNS (Domain Name Service) port, for associating names with IP addresses.
  - Port 22 is the port for SSH (Secure Shell)
  - Port 631 is for IPP (Internet Printing Protocol)
- For your own programs, generally try to stay away from port numbers listed in the links above, but otherwise, ports are up for grabs to any program that wants one.

Lecture 13: 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 descriptor that can be written to and read from. Whatever's written is sent to the client, and whatever the client sends back is readable here.
  - This descriptor is one end of a bidirectional pipe bridging two processes—on different machines!

Lecture 13: 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 13: 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 like **`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 13: 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 13: 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'll be implementing one for Assignment 5) to get the computation off the main thread.

```
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: Introduction to Networking

- The implementation of **`publishTime`** needs to change just a little if it's to be thread safe.
- The change is simple but important: we need to call a different version of **`gmtime`**.
- **`gmtime`** returns a pointer to a single, statically allocated global that's used by all calls.
- If two threads make competing calls to it, then both threads race to pull time information from the shared, statically allocated record.
- Of course, one solution would be to use a **`mutex`** to ensure that a thread can call **`gmtime`** without competition and subsequently extract the data from the global into local copy.
- Another solution—one that doesn't require locking and one I think is better—makes use of a second version of the same function called **`gmtime_r`**. This second, reentrant version just requires that space for a dedicated return value be passed in.

Lecture 13: Introduction to Networking

- A function is **reentrant** if a call to it can be interrupted in the middle of its execution and called a second time before the first call has completed.
- While not all reentrant functions are thread-safe, `gmtime_r` itself is, since it doesn't depend on any shared resources.
- A thread-safe version of `publishTime` is presented below.

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