Lecture 19: Introduction to Networking

- Implementing your first server!
  - Colloquially, a server-side application waits by the phone, waiting for someone to call.
  - Formally, a server-side application creates a server socket that listens to a particular port.
    - The server socket is an integer identifier associated with a local IP address (think phone number) and port (think phone extension).
    - You can also think of the port number as a virtual process ID that the host associates with the true pid of the server application.
  - Here is our first example of a server:

```c
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); // details presented on next slide
    }
    return 0;
}
```

- `accept` 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 19: Introduction to Networking

- Implementing your first server!
  - The implementation of `publishTime` is computationally straightforward.

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

- Implementing your second server!
  - The **while** loop around the exposed **write** calls in the prior example were actually necessary this time.
    - The socket descriptor is bound to a network driver that may have a limited amount of space.
    - That means **write**'s return value could very well be less than what was supplied by the third argument.
  - 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 exactly this.
    - **socket++** provides iostream subclasses that respond to **operator<<**, **operator>>**, **getline**, **endl**, and so forth, just like **cin**, **cout**, and file streams do.
    - We're going to operate as if this third-party library is just part of standard C++.
  - The next slide includes a prettier, version of **publishTime**.
Lecture 19: Introduction to Networking

- Implementing your second server!
  - Here's the new implementation of **publishTime**:

```c
static void publishTime(int client) {
    time_t rawtime;
    time(&rawtime);
    struct tm *ptm = gmtime(&rawtime);
    char timestr[128];
    /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", ptm);
    sockbuf sb(client);
    iosockstream ss(&sb);
    ss << timestr << endl;
} // the sockbuf closes the client socket when it's destroyed
```

- 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'll be implementing one for Assignment 6) to get the computation off the main thread.

```c
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 19: 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.
    - 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.
  - The thread-safe version of `publishTime` is presented on the next slide.
Lecture 19: Introduction to Networking

- Here's the updated version of `publishTime`:

```c
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;
}
```