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).

Lecture 13: Networks and Clients

- Emulation of **wget**
  - **wget** is a command line utility that, given its URL, downloads a single document (HTML document, image, video, etc.) and saves a copy of it to the current working directory.
  - Without being concerned so much about error checking and robustness, we can write a simple program to emulate **wget**'s most basic functionality.
  - To get us started, here are the **main** and **parseURL** functions.
- **parseURL** dissects the supplied URL to surface the host and pathname components.

```
static const string kProtocolPrefix = "http://";
static const string kDefaultPath = "/";
static pair<string, string> parseURL(string url) {
    if (startsWith(url, kProtocolPrefix))
        url = url.substr(kProtocolPrefix.size());
    size_t found = url.find('/');
    if (found == string::npos)
        return make_pair(url, kDefaultPath);
    string host = url.substr(0, found);
    string path = url.substr(found);
    return make_pair(host, path);
}

int main(int argc, char *argv[]) {
    pullContent(parseURL(argv[1]));
    return 0;
}
```

Lecture 13: Networks and Clients

Emulation of **wget** (continued)

- **pullContent**, of course, needs to manage everything, including the networking.

```
static const unsigned short kDefaultHTTPPort = 80;
static void pullContent(const pair<string, string>& components) {
    int client = createClientSocket(components.first, kDefaultHTTPPort);
    if (client == kClientSocketError) {
        cerr << "Could not connect to host named \"" << components.first << "\"." << endl;
        return;
    }
    sockbuf sb(client);
    iosockstream ss(&sb);
    issueRequest(ss, components.first, components.second);
    skipHeader(ss);
    savePayload(ss, getFileName(components.second));
}
```

- We've already used this **createClientSocket** function for our **time-client**. This time, we're connecting to real but arbitrary web servers that speak HTTP.
- The implementations of **issueRequest**, **skipHeader**, and **savePayload** subdivide the client-server conversation into manageable chunks.
  - The implementations of these three functions have little to do with network connections, but they have much to do with the protocol that guides any and all HTTP conversations.

Lecture 13: Networks and Clients

Emulation of **wget** (continued)

Here's the implementation of **issueRequest**, which generates the smallest legitimate HTTP request possible and sends it over to the server.

```
static void issueRequest(iosockstream& ss, const string& host, const string& path) {
    ss << "GET " << path << " HTTP/1.0\r\n";
    ss << "Host: " << host << "\r\n";
    ss << "\r\n";
    ss.flush();
}
```

- It's standard HTTP-protocol practice that each line, including the blank line that marks the end of the request, end in CRLF (short for carriage-return-line-feed), which is **'\r'** following by **'\n'**.
- The **flush** is necessary to ensure all character data is pressed over the wire and consumable at the other end.
- After the **flush**, the client transitions from supply to ingest mode. Remember, the **iosockstream** is read/write, because the socket descriptor backing it is bidirectional.

Lecture 13: Networks and Clients

- **`skipHeader`** reads through and discards all of the HTTP response header lines until it encounters either a blank line or one that contains nothing other than a **`'\r'`**. The blank line is, indeed, supposed to be **`"\r\n"`**, but some servers—often hand-rolled ones—are sloppy, so we treat the **`'\r'`** as optional. Recall that getline chews up the **`'\n'`**, but it won't chew up the **`'\r'`**.

```
static void skipHeader(iosockstream& ss) {
    string line;
    do {
        getline(ss, line);
    } while (!line.empty() && line != "\r");
}
```

- In practice, a true HTTP client—in particular, something as HTTP-compliant as the **`wget`** we're imitating—would ingest all of the lines of the response header into a data structure and allow it to influence how it treats payload.
  - For instance, the payload might be compressed and should be expanded before saved to disk.
  - I'll assume that doesn't happen, since our request didn't ask for compressed data.

Lecture 13: Networks and Clients

- Everything beyond the response header and that blank line is considered payload—that's the timestamp, the JSON, the HTML, the image, or the cat video.
  - Every single byte that comes through should be saved to a local copy.

```
static string getFileName(const string& path) {
    if (path.empty() || path[path.size() - 1] == '/') return "index.html";
    size_t found = path.rfind('/');
    return path.substr(found + 1);
}

static void savePayload(iosockstream& ss, const string& filename) {
    ofstream output(filename, ios::binary); // don't assume it's text
    size_t totalBytes = 0;
    while (!ss.fail()) {
        char buffer[2014] = {'\0'};
        ss.read(buffer, sizeof(buffer));
        totalBytes += ss.gcount();
        output.write(buffer, ss.gcount());
    }
    cout << "Total number of bytes fetched: " << totalBytes << endl;
}
```

- HTTP dictates that everything beyond that blank line is payload, and that once the server publishes that payload, it closes its end of the connection. That server-side close is the client-side's **EOF**, and we write everything we read.