CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 20: Networks, Clients, and Protocols

- 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 20: Networks, Clients, and Protocols

- We have been talking about how to open connections between two machines. These two machines are typically a server and a client.
  - If the server writes to its socket endpoint, whatever's written is sent to the client. If the server reads from its socket endpoint, it's reading whatever the client sent.
  - In principle, the client and the server can send anything they want. However, the conversation goes more smoothly if the two sides speak the same language—or more formally, the conversation conforms to some well-defined grammar.
  - In networking terms, this conversation grammar is referred to as a **protocol**.
  - Formally, a protocol is a specification dictating how two computers should should converse. By respecting a protocol, both client and server know they'll understand each other, even if they are running different software written by different people in different programming languages, running in different environments on varying architectures.
  - Network protocols are generally codified in **R**equests **F**or **C**omments, or RFCs.
  - The specification for HTTP/1.1, for instance, is right here. It's long and reads like legalese, but if you implement a client program that respects the HTTP/1.1 protocol, the client application can interact with any server that speaks HTTP/1.1.

Lecture 20: Networks, Clients, and Protocols

- The HTTP protocol is pretty much universal in the networking world. Since so many programs speak and understand it, it's a common protocol for exchanging information and executing commands over network connections.
- Once a client establishes a connection to a server, the client sends a request, and then the server sends a response. The client and server can go back and forth several times, issuing several requests and responses over the same connection before closing it.
- To keep things simple for lecture, we'll only look at examples where the client sends a single request and ingests a single response from the server.

- An HTTP request (sent by a client to a server) looks something like the following:

```
GET /search?q=cats&tbm=isch HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:61.0) Gecko/20100101 Firefox/61.0
Accept-Language: en-US,en;q=0.5
```

Lecture 20: Networks, Clients, and Protocols

```
GET /search?q=cats&tbm=isch HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:61.0) Gecko/20100101 Firefox/61.0
Accept-Language: en-US,en;q=0.5
```

- The first line is called the start line or request line.
- The first part of the request line is a **verb**. HTTP supports *many* verbs as part of the language, but you should know the following:
  - **GET**: requests some information from the server
  - **POST**: upload information to the server with the expectation that the server stores it or something related to it
    - Logging into a website sends a **POST** request, because you're sending your username and password and creating a login session on that server.
    - Uploading an image to Google Drive typically sends a **POST** request as well. You're asking the server to store your image somewhere to retrieve later on.
  - **HEAD**: like a **GET** request, except the response should only include metadata without the (potentially very large) payload.
    - Why? Metadata included in an HTTP response often includes information about how recently the document of interest has changed. If some **Last-Modified** timestamp suggests your cached copy is the same as the one it would other send you, you can skip the full **GET** request.

Lecture 20: Networks, Clients, and Protocols

```
GET /search?q=cats&tbm=isch HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:61.0) Gecko/20100101 Firefox/61.0
Accept-Language: en-US,en;q=0.5
```

- The second part of the request line is the *request path*. This particular path—you can tell it's just the full URL with the **http://www.google.com** chopped of—tells Google that I want to **search**, that I am looking for cats (**q=cats**, where **q** is short for query), and that I want to search Google images (**tbm=isch**, where **isch** is short for image search, and tbm is reportedly short for to be matched, though Google hasn't documented that).
- Finally, the last part of the request line specifies what version of the HTTP protocol we want to speak. HTTP/2.0 is becoming widespread as of just this year, and it has many exciting features. In this class, we'll limit ourselves to HTTP/1.1. Let's not get crazy.

- Following the request line are zero or more lines of **request headers**.
  - Headers are key/value pairs, written as Key: Value, with each pair on a separate line. There are many standard headers, although a program can add any extra, non-standard headers if it likes. In the example above, my browser is confessing that it's Firefox (through the **User-Agent** header) and that, if possible, any content should be expressed in American English (through the **Accept-Language** header).

- Finally, a blank line is used to mark the end of the request headers.

Lecture 20: Networks, Clients, and Protocols

- If the client needs to include a payload with its request (e.g. to upload a new profile picture for your LinkedIn profile), the client can inline the payload after the blank line.
- This typically only happens with **POST** requests, although **GET** requests are technically permitted to upload a payload as well.
- One example? A request to log into a website might look something like this:

```
POST /login HTTP/1.1
Host: myinsecurebank.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:61.0) Gecko/20100101 Firefox/61.0
Accept-Language: en-US,en;q=0.5
Content-Length: 29

username=poohbear&password=pw
```

- Another? A request to upload an image to might look like this:

```
POST /uploadpp HTTP/1.1
Host: www.clubcardinal.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:61.0) Gecko/20100101 Firefox/61.0
Accept-Language: en-US,en;q=0.5
Content-Length: 3829102

{file: {name: "jerry-and-doris.png", data: "LyoqCiAqIGludC5jYwogKiAtLS0tLS0KICogU2hvcnQgcHJvZ3Jh
c2Vjb25kcyBpbiBvbmUtc2Vjb25kIGJ1cnN0cywgYW5kIHRoZW4gc2VuZHMgaXRzZWxmIGEKICogU0lHSU5ULgogKi8KI2lu
Z2V0cGlkCiNpbmNsdWRlIDxzeXMvd2FpdC5oPiAgICAgLy8gZm9yIHJhaXNlLCBTSUdJTlQKdXNpbmcgbmFtZXNwYWNlIHN0
IDE7CnN0YXRpYyBjb25zdCBpbnQga1JhaXNlRmFpbGVkID0gMjsKaW50IG1haW4oaW50IGFyZ2MsIGNoYXIgKmFyZ3ZbXSkg
IDw8IGFyZ3ZbMF0gPDwgIiA8bj4iIDw8IGVuZGw7CiAgICBByZXR1cm4ga1dyb25nQXJndW1lbnRDb3VudDsKICB9CgogIHNp
ID0gMDsgaSA8IHNlY3M7IGkrKykgc2xlZXAoMSk7CgogIGlmICghyYWlzZShTSUdJTlQpICE9IDApIHsKICAgIGNlcnIgPDwg
// lots and lots of base64-encoded bytes
MDsKfQo="}}
```

Lecture 20: Networks, Clients, and Protocols

- An HTTP response is structured as below:

```
HTTP/1.1 200 OK
Date: Fri, 05 Nov 2021 00:39:59 GMT
Server: Apache
Accept-Ranges: bytes
Transfer-Encoding: chunked
Content-Type: text/html

3f6
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1,shrink-to-fit=no" />
    <title>CS110: Principles of Computer Systems, Autumn 2021</title>
    <base href="https://web.stanford.edu/class/archive/cs/cs110/cs110.1222/">
    <link rel="stylesheet" href="assets/vendors/bootstrap-4.0.0-beta.2.min.css" />
    <link rel="stylesheet" href="assets/styles.css" type="text/css" />
    <script src="assets/cs110-code.js" async="async"></script>
```

- The start line specifies the HTTP version that the server can speak, as well as an HTTP status code. I'm sure you've seen status code **404 Not Found** and maybe even **403 Forbidden**. and **500 Internal Server Error**.
- Response headers follow—e.g. **Content-Type** indicates that we've been sent an HTML page. Other common headers include **Set-Cookie**, used to send browser cookies, and **Cache-Control**, which tells the browser whether it can cache the page or not.

- A blank line indicates the end of the headers, and then the server sends the payload.

Lecture 20: Networks, Clients, and Protocols

- 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 20: Networks, Clients, and Protocols

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);
  skipToPayload(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**, **skipToPayload**, 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 20: Networks, Clients, and Protocols

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 20: Networks, Clients, and Protocols

- **`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 skipToPayload(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 its execution.
  - For instance, the payload might be compressed and should be expanded before being saved to disk.
  - I'll assume that doesn't happen, since our request didn't ask for compressed data.

Lecture 20: Networks, Clients, and Protocols

- 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 everything we read gets saved to disk.

Lecture 20: Networks, Clients, and Protocols

- An application programming interface (or API) is a set of library functions one can use in order to build a larger piece of software.
  - You're familiar with *some* APIs: #**include** files, system calls, and ad hoc protocols for driving and communicating with child processes using pipes and signals.
  - Very often these libraries reside **on other machines**, and we interface with them over the Internet.
- I want to implement an API server that's architecturally in line with the way Google, Twitter, Facebook, and LinkedIn architect their own API services.
- This example is inspired by a website called Lexical Word Finder.
  - Our implementation assumes we have a standard Unix executable called **`scrabble-word-finder`**. The source code for this executable—completely unaware it'll be used in a larger networked application—can be found right here.
  - Here are two abbreviated sample runs:

```
poohbear@myth61:$ ./scrabble-word-finder lexical
ace
// many lines omitted for brevity
lexical
li
lice
lie
lilac
xi
poohbear@myth61:$
```

```
poohbear@myth61:$ ./scrabble-word-finder network
en
// many lines omitted for brevity
work
worn
wort
wot
wren
wrote
poohbear@myth61:$
```

Lecture 20: Networks, Clients, and Protocols

- I want to implement an API service using HTTP to replicate what **`scrabble-word-finder`** is capable of.
  - We'll expect the API call to come in the form of a URL, and we'll expect that URL to include the rack of letters.
  - Assuming our API server is running on **`myth54:13133`**, we expect **`http://myth54:13133/lexical`** and **`http://myth54:13133/network`** to generate the following payloads:

```
{
  time: 0.223399,
  cached: false,
  possibilities: [
    'ace',
    // several words omitted
    'lexical',
    'li',
    'lice',
    'lie',
    'lilac',
    'xi'
  ]
}
```

```
{
  time: 0.223399,
  cached: false,
  possibilities: [
    'en',
    // several words omitted
    'work',
    'worn',
    'wort',
    'wot',
    'wren',
    'wrote'
  ]
}
```

Lecture 20: Networks, Clients, and Protocols

- One might think to cannibalize the code within **`scrabble-word-finder.cc`** to build the core of **`scrabble-word-finder-server.cc`**.
- Reimplementing from scratch is wasteful, time-consuming, and unnecessary. **`scrabble-word-finder`** already outputs the primary content we need for our payload. We're packaging the payload as JSON instead of plain text, but we can still tap **`scrabble-word-finder`** to generate the collection of formable words.
- Can we implement a server that leverages existing functionality? Of course we can!
- We can just leverage our **`subprocess_t`** type and **`subprocess`** function from Assignment 3.

```
struct subprocess_t {
    pid_t pid;
    int supplyfd;
    int ingestfd;
};

subprocess_t subprocess(char *argv[], bool supplyChildInput, bool ingestChildOutput);
```

Lecture 20: Networks, Clients, and Protocols

- Here is the core of the **`main`** function implementing our server:

```
int main(int argc, char *argv[]) {
    unsigned short port = extractPort(argv[1]);
    int server = createServerSocket(port);
    cout << "Server listening on port " << port << "." << endl;
    ThreadPool pool(16);
    map<string, vector<string>> cache;
    mutex cacheLock;
    while (true) {
        struct sockaddr_in address;
        // used to surface IP address of client
        socklen_t size = sizeof(address); // also used to surface client IP address
        bzero(&address, size);
        int client = accept(server, (struct sockaddr *) &address, &size);
        char str[INET_ADDRSTRLEN];
        cout << "Received a connection request from "
             << inet_ntop(AF_INET, &address.sin_addr, str, INET_ADDRSTRLEN) << "." << endl;
        pool.schedule([client, &cache, &cacheLock] {
            publishScrabbleWords(client, cache, cacheLock);
        });
    }
    return 0; // server never gets here, but not all compilers can tell
}
```

Lecture 20: Networks, Clients, and Protocols

- The second and third arguments to **`accept`** are used to surface the client's IP address.
- Ignore the details around how I use **`address`**, **`size`**, and the **`inet_ntop`** function until next week, when we'll talk more about them. Right now, it's a neat-to-see!
- Each request is handled by a worker thread within a **`ThreadPool`** of size 16.
- The thread routine called **`publishScrabbleWords`** will rely on our **`subprocess`** function to marshal plain text output of scrabble-word-finder into JSON and publish that JSON as the payload of the HTTP response.
- The next several slides include the full implementation of **`publishScrabbleWords`** and some of its helper functions.
- Most of the complexity comes around the fact that I've *elected* to maintain a cache of previously processed letter racks.

Lecture 20: Networks, Clients, and Protocols

- Here is **`publishScrabbleWords`**:

```
static void publishScrabbleWords(int client, map<string, vector<string>>& cache,
                                 mutex& cacheLock) {
    sockbuf sb(client);
    iosockstream ss(&sb);
    string letters = getLetters(ss);
    sort(letters.begin(), letters.end());
    skipHeaders(ss);
    struct timeval start;
    gettimeofday(&start, NULL); // start the clock
    cacheLock.lock();
    auto found = cache.find(letters);
    cacheLock.unlock(); // release lock immediately, iterator won't be invalidated by competing find calls
    bool cached = found != cache.end();
    vector<string> formableWords;
    if (cached) {
        formableWords = found->second;
    } else {
        const char *command[] = {"./scrabble-word-finder", letters.c_str(), NULL};
        subprocess_t sp = subprocess(const_cast<char **>(command), false, true);
        pullFormableWords(formableWords, sp.ingestfd); // function exits
        waitpid(sp.pid, NULL, 0);
        lock_guard<mutex> lg(cacheLock);
        cache[letters] = formableWords;
    }
    struct timeval end, duration;
    gettimeofday(&end, NULL); // stop the clock, server-computation of formableWords is complete
    timersub(&end, &start, &duration);
    double time = duration.tv_sec + duration.tv_usec/1000000.0;
    ostringstream payload;
    constructPayload(formableWords, cached, time, payload);
    sendResponse(ss, payload.str());
}
```

Lecture 20: Networks, Clients, and Protocols

- Here's the **`pullFormableWords`** and **`sendResponse`** helper functions.

```
static void pullFormableWords(vector<string>& formableWords, int ingestfd) {
    stdio_filebuf<char> inbuf(ingestfd, ios::in);
    istream is(&inbuf);
    while (true) {
        string word;
        getline(is, word);
        if (is.fail()) break;
        formableWords.push_back(word);
    }
}

static void sendResponse(iosockstream& ss, const string& payload) {
    ss << "HTTP/1.1 200 OK\r\n";
    ss << "Content-Type: text/javascript; charset=UTF-8\r\n";
    ss << "Content-Length: " << payload.size() << "\r\n";
    ss << "\r\n";
    ss << payload << flush;
}
```

Lecture 20: Networks, Clients, and Protocols

- Finally, here are the **`getLetters`** and the **`constructPayload`** helper functions. I omit the implementation of **`skipHeaders`**—you saw it with **`web-get`**—and **`constructJSONArray`**, which you're welcome to view right here.

```
static string getLetters(iosockstream& ss) {
    string method, path, protocol;
    ss >> method >> path >> protocol;
    string rest;
    getline(ss, rest);
    size_t pos = path.rfind("/");
    return pos == string::npos ? path : path.substr(pos + 1);
}

static void constructPayload(const vector<string>& formableWords, bool cached,
                             double time, ostringstream& payload) {
    payload << "{" << endl;
    payload << " time: " << time << "," << endl;
    payload << " cached: " << boolalpha << cached << "," << endl;
    payload << " possibilities: " << constructJSONArray(formableWords, 2) << endl;
    payload << "}" << endl;
}
```

- Our **`scrabble-word-finder-server`** provided a single API call that resembles the types of API calls afforded by Google, Twitter, or Facebook to access search, tweet, or friend-graph data.