

Lecture 15: Networking, Clients

Principles of Computer Systems
Spring 2019
Stanford University
Computer Science Department
Lecturer: Chris Gregg



[PDF of this presentation](#)

Lecture 15: Networking, 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(client >= 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 15: Networking, 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 the `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 15: Networking, 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 15: Networking, 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 15: Networking, 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 15: Networking, 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.

Lecture 15: API Servers, Threads, Processes

- I want to implement an API server that's architecturally in line with the way Google, Twitter, Facebook, and LinkedIn architect their own API servers.
- This example is inspired by a website called [Lexical Word Finder](#).
 - Our implementation assumes we have a standard Unix executable called **scrabbleword-finder**. The source code for this executable—completely unaware it'll be used in a larger networked application—can be found [right here](#).
 - **scrabble-word-finder** is implemented using only CS106B techniques—standard file I/O and procedural recursion with simple pruning.
 - Here are two abbreviated sample runs:

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

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


Lecture 15: API Servers, Threads, Processes

- I want to implement an API service using HTTP to replicate what **scrabble-wordfinder** 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
    'lei',
    'lex',
    'lexica',
    'lexical',
    'li',
    'lice',
    'lie',
    'lilac',
    'xi'
  ]
}
```

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

Lecture 15: API Servers, Threads, Processes

- 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) throw (SubprocessException);
```

Lecture 15: API Servers, Threads, Processes

- 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;
}
```

Lecture 15: API Servers, Threads, Processes

- The second and third arguments to **accept** are used to surface the IP address of the client.
- 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 dedicated 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 slide includes 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 15: API Servers, Threads, Processes

- 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);
        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 15: API Servers, Threads, Processes

- 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 15: API Servers, Threads, Processes

- 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, ostream& 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.