Lecture 15: Networking, Building an API

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Instructors: Chris Gregg and Philip Levis

PDF of this presentation

Lecture 15: API Servers, Threads, Processes

- Let's 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 **`scrableword-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, in JSON format:


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

```
{
  "time": 0.242551,
  "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 the next lecture, 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: application/javascript; charset=UTF-8\r\n";
    ss << "Content-Length: " << payload.size() << "\r\n";
    ss << "\r\n";
    ss << payload << flush;
}
```

HTTP status ranges in a nutshell:

1xx: hold on

2xx: here you go

3xx: go away

4xx: you f up

5xx: I f up

-via @abt_programming

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,
                                                                    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.

Lecture 15: API Servers, Threads, Processes

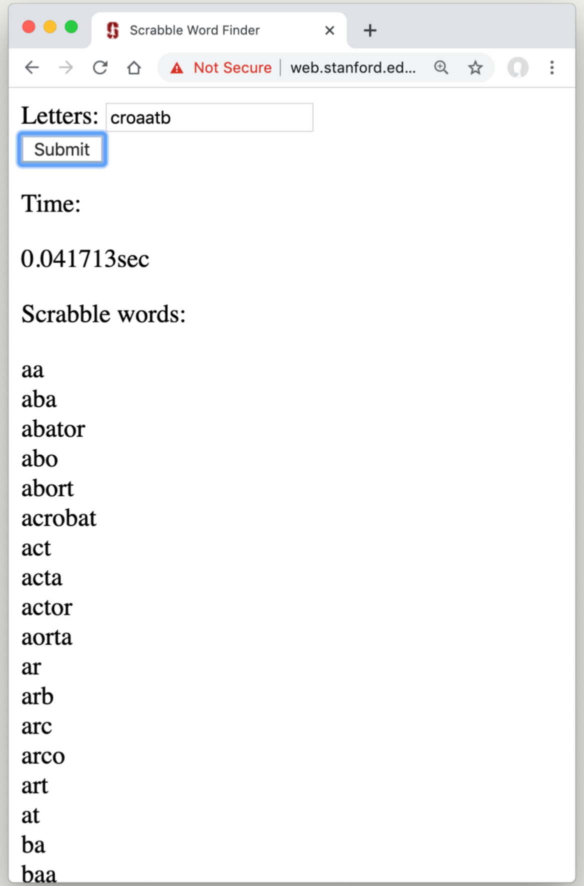
- It turns out we actually wrote a web-ready program. Let's look at some HTML and Javascript:

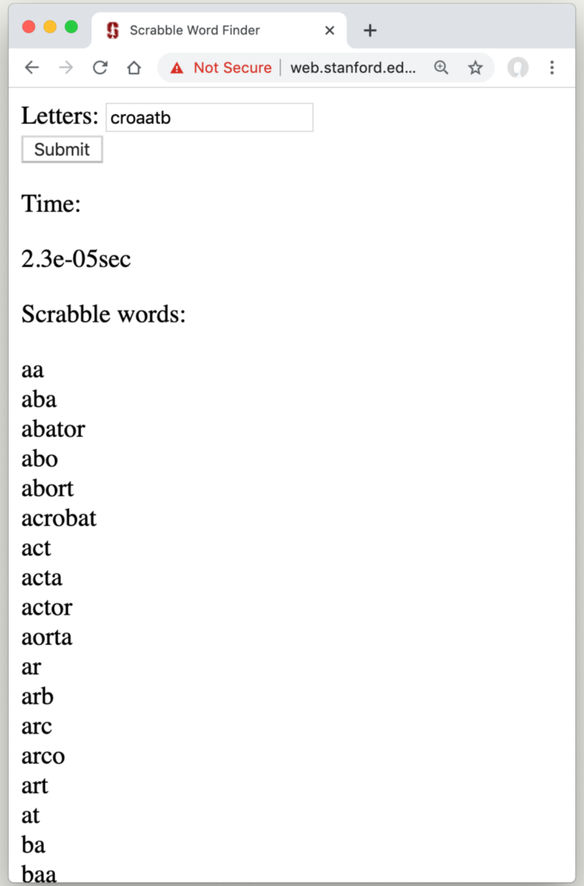
```
<!DOCTYPE html>
<html>
    <head>
        <title>Scrabble Word Finder</title>
    </head>
    <body>
        Letters: <input type="text" id="letters" name="letters"><br>
        <input type="submit" value="Submit" onclick="getWords()">
        <p>Time:</p><span id="words_time"></span>
        <p>Scrabble words:</p>
        <div id="scrabble_words"></div>

        <script>
            function getWords(){
                let letters = document.getElementById("letters").value;
                let scrabble_words = fetch("http://myth59.stanford.edu:13133/"+letters,{method:"GET"})
                    .then(data=>{return data.json()})
                    .then(res=>{
                        console.log(res);
                        document.getElementById("words_time").innerText = res['time']+"sec";
                        possibilitiesStr = "";
                        for (var i=0; i < res.possibilities.length; i++) {
                            possibilitiesStr += res.possibilities[i]+"<br>";
                        }
                        document.getElementById("scrabble_words").innerHTML = possibilitiesStr;
                    })
                    .catch(error=>console.log(error))
            }
        </script>
    </body>
</html>
```

Lecture 15: API Servers, Threads, Processes

- We can put the html file in a Stanford web location (e.g., http://web.stanford.edu/class/cs110/scrabble-word-finder.html)
- As long as we are running our scrabble server on myth59, port 13133, we can get words:





- We have a legitimate back-end server. It has a cache, just like built it to have.
- Are many servers written in C++?
- Surprisingly, yes -- it is fast, which often makes it better than Python, or Node, or PHP.

Lecture 15: Network System Calls, Library Functions

- Hostname Resolution: IPv4
- Linux C includes directives to convert host names (e.g. "**`www.facebook.com`**") to IPv4 address (e.g. "**`31.13.75.17`**") and vice versa. Functions called **`gethostbyname`** and **`gethostbyaddr`**, while technically deprecated, are still so prevalent that you should know how to use them.
- In fact, your B&O textbook only mentions these deprecated functions:

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

- Each function populates a statically allocated **`struct hostent`** describing some host machine on the Internet.
  - **`gethostbyname`** assumes its argument is a host name (e.g. "**`www.google.com`**").
  - **`gethostbyaddr`** assumes the first argument is a binary representation of an IP address (e.g. not the string "**`171.64.64.137`**", but the base address of a character array with ASCII values of 171, 64, 64, and 137 laid down side by side in **`network byte order`**. For IPv4, the second argument is usually 4 (or rather, **`sizeof(struct in_addr)`**) and the third is typically the **`AF_INET`** constant.

Lecture 15: Network System Calls, Library Functions

- Hostname Resolution: IPv4
  - The **`struct hostent`** record packages all of the information about a particular host:

```
struct in_addr {
    unsigned int s_addr // four bytes, stored in network byte order (big endian)
};
struct hostent {
    char *h_name;
    // official name of host
    char **h_aliases;
    // NULL-terminated list of aliases
    int h_addrtype;
    // host address type (typically AF_INET for IPv4)
    int h_length;
    // address length (typically 4, or sizeof(struct in_addr) for IPv4)
    char **h_addr_list; // NULL-terminated list of IP addresses
}; // h_addr_list is really a struct in_addr ** when hostent contains IPv4 addresses
```

- The **`struct in_addr`** is a one-field record modeling an IPv4 address.
  - The **`s_addr`** field packs each figure of a dotted quad (e.g. 171.64.64.136) into one of its four bytes. Each of these four numbers numbers can range from 0 up through 255.
- The **`struct hostent`** is used for all IP addresses, not just IPv4 addresses. For non-IPv4 addresses, **`h_addrtype`**, **`h_length`**, and **`h_addr_list`** carry different types of data than they do for IPv4

Lecture 15: Network System Calls, Library Functions

Users prefer the host naming scheme behind "**`www.facebook.com`**", but network communication ultimately works with IP addresses like "31.13.75.17".

- Not surprisingly, **`gethostbyname`** and **`gethostbyaddr`** are used to manage translations between the two.
- Here's the core of larger program (full program here) that continuously polls the users for hostnames and responds by publishing the set of one or more IP addresses each hostname is bound to:

```
static void publishIPAddressInfo(const string& host) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) { // NULL return value means resolution attempt failed
        cout << host << " could not be resolved to an address. Did you mistype it?" << endl;
        return;
    }

    cout << "Official name is \"" << he->h_name << "\"" << endl;
    cout << "IP Addresses: " << endl;
    struct in_addr **addressList = (struct in_addr **) he->h_addr_list;
    while (*addressList != NULL) {
        char str[INET_ADDRSTRLEN];
        cout << "+ " << inet_ntop(AF_INET, *addressList, str, INET_ADDRSTRLEN) << endl;
        addressList++;
    }
}
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv4

- **`h_addr_list`** is typed to be a **`char *`** array, implying it's an array of C strings, perhaps dotted quad IP addresses. However, that's not correct. For IPv4 records, **`h_addr_list`** is an array of **`struct in_addr *s`**.
- The **`inet_ntop`** function places a traditional C string presentation of an IP address into the provided character buffer, and returns the the base address of that buffer.
- The while loop crawls over the **`h_addr_list`** array until it lands on a **`NULL`**.

```
static void publishIPAddressInfo(const string& host) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) { // NULL return value means resolution attempt failed
        cout << host << " could not be resolved to an address. Did you mistype it?" << endl;
        return;
    }

    cout << "Official name is \"" << he->h_name << "\"" << endl;
    cout << "IP Addresses: " << endl;
    struct in_addr **addressList = (struct in_addr **) he->h_addr_list;
    while (*addressList != NULL) {
        char str[INET_ADDRSTRLEN];
        cout << "+ " << inet_ntop(AF_INET, *addressList, str, INET_ADDRSTRLEN) << endl;
        addressList++;
    }
}
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv4

- A sample run of our hostname resolver is presented on the right.
- In general, you see that most of the hostnames we recognize are in fact the officially recorded hostnames.
- **`www.yale.edu`**, **`www.facebook.com`**, and **`www.wikipedia.org`** are exceptions. It looks like Yale relies on a content delivery network called Cloudflare, and **`www.yale.edu`** is catalogued as an alias.
- Google's IP address is different by geographical location, which is why it exposes only one IP address.

```
myth61$ ./resolve-hostname
Welcome to the IP address resolver!
Enter a host name: www.google.com
Official name is "www.google.com"
IP Addresses:
+ 216.58.192.4
Enter a host name: www.coinbase.com
Official name is "www.coinbase.com"
IP Addresses:
+ 104.16.9.251
+ 104.16.8.251
Enter a host name: www.yale.edu
Official name is "www.yale.edu.cdn.cloudflare.net"
IP Addresses:
+ 104.16.140.133
+ 104.16.141.133
Enter a host name: www.facebook.com
Official name is "star-mini.c10r.facebook.com"
IP Addresses:
+ 31.13.70.36
Enter a host name: www.wikipedia.org
Official name is "dyna.wikimedia.org"
IP Addresses:
+ 198.35.26.96
Enter a host name:
All done!
myth61$
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- Because IPv4 addresses are 32 bits, there are 2^32, or roughly 4 billion different IP addresses. That may sound like a lot, but it was recognized decades ago that we'd soon run out of IPv4 addresses.
- In contrast, there are 340,282,366,920,938,463,463,374,607,431,768,211,456 IPv6 addresses. That's because IPv6 addresses are 128 bits.
- Here are a few IPv6 addresses:
  - Google's 2607:f8b0:4005:80a::2004
  - MIT's 2600:1406:1a:396::255e and 2600:1406:1a:38d::255e
  - Berkeley's 2600:1f14:436:7801:15f8:d879:9a03:eec0 and 2600:1f14:436:7800:4598:b474:29c4:6bc0
  - The White House's 2600:1406:1a:39e::fc4 and 2600:1406:1a:39b::fc4

A more generic version of **`gethostbyname`**—inventively named **`gethostbyname2`**—can be used to extract IPv6 address information about a hostname.

```
struct hostent *gethostbyname2(const char *name, int af);
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- There are only two valid address types that can be passed as the second argument to **`gethostbyname2`**: **`AF_INET`** and **`AF_INET6`**.
  - A call to **`gethostbyname2(host, AF_INET)`** is equivalent to a call to **`gethostbyname(host)`**
  - A call to **`gethostbyname2(host, AF_INET6)`** still returns a **`struct hostent *`**, but the struct hostent is populated with different values and types:
    - the **`h_addrtype`** field is set to **`AF_INET6`**,
    - the **`h_length`** field houses a 16 (or rather, sizeof(struct in6_addr)), and
    - the **`h_addr_list`** field is really an array of **`struct in6_addr`** pointers, where each **`struct in6_addr`** looks like this:

```
struct in6_addr {
    u_int8_t s6_addr[16]; // 16 bytes (128 bits), stored in network byte order
};
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- Here is the **`IPv6`** version of the **`publishIPAddressInfo`** we wrote earlier (we call it **`publishIPv6AddressInfo`**).

```
static void publishIPv6AddressInfo(const string& host) {
    struct hostent *he = gethostbyname2(host.c_str(), AF_INET6);
    if (he == NULL) { // NULL return value means resolution attempt failed
        cout << host << " could not be resolved to an address. Did you mistype it?" << endl;
        return;
    }

    cout << "Official name is \"" << he->h_name << "\"" << endl;
    cout << "IPv6 Addresses: " << endl;
    struct in6_addr **addressList = (struct in6_addr **) he->h_addr_list;
    while (*addressList != NULL) {
        char str[INET6_ADDRSTRLEN];
        cout << "+ " << inet_ntop(AF_INET6, *addressList, str, INET6_ADDRSTRLEN) << endl;
        addressList++;
    }
}
```

- Notice the call to **`gethostbyname2`**, and notice the explicit use of **`AF_INET6`**, **`struct in6_addr`**, and **`INET6_ADDRSTRLEN`**.
- Full program is right here.

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- A sample run of our IPv6 hostname resolver is presented below.
  - Note that many hosts aren't IPv6-compliant yet, so they don't admit IPv6 addresses.

```
myth61$ ./resolve-hostname6
Welcome to the IPv6 address resolver!
Enter a host name: www.facebook.com
Official name is "star-mini.c10r.facebook.com"
IPv6 Addresses:
+ 2a03:2880:f131:83:face:b00c:0:25de
Enter a host name: www.microsoft.com
Official name is "e13678.dspb.akamaiedge.net"
IPv6 Addresses:
+ 2600:1406:1a:386::356e
+ 2600:1406:1a:397::356e
Enter a host name: www.google.com
Official name is "www.google.com"
IPv6 Addresses:
+ 2607:f8b0:4005:801::2004
Enter a host name: www.berkeley.edu
Official name is "www-production-1113102805.us-west-2.elb.amazonaws.com"
IPv6 Addresses:
+ 2600:1f14:436:7800:4598:b474:29c4:6bc0
+ 2600:1f14:436:7801:15f8:d879:9a03:eec0
Enter a host name: www.stanford.edu
www.stanford.edu could not be resolved to an address. Did you mistype it?
Enter a host name:
All done!
myth61$
```