CS110 Lecture 22: HTTP and APIs

CS110: Principles of Computer Systems

Winter 2021-2022

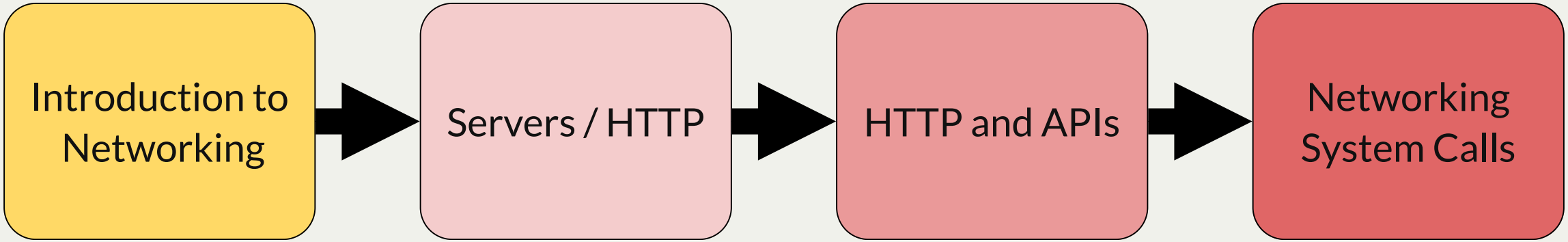
Stanford University

Instructors: Nick Troccoli and Jerry Cain

PDF of this presentation

CS110 Topic 4: How can we write programs that communicate over a network with other programs?

Learning About Networking



assign6: implement an HTTP Proxy that sits between a client device and a web server to monitor, block or modify web traffic.

Learning Goals

- Gain more practice with the client-server model
- Understand the HTTP protocol for making requests and responses
- Write a client program that makes HTTP requests
- Write a server program that sends back HTTP responses

Plan For Today

- **Recap:** Protocols and HTTP
- **HTTP Client Example:** wget
- **HTTP Server Example:** scrabble

Plan For Today

- **Recap: Protocols and HTTP**
- **HTTP Client Example:** wget
- **HTTP Server Example:** scrabble

Data Protocols

Our time server chose to send a raw single-line string response to a client. A client connecting must be aware of this to know how to handle / use the response data.

Key idea: a client and server must agree on the format of the data being sent back and forth so they know what to send and how to parse the response.

- A **protocol** is a specification dictating how two computers should should converse. By respecting a protocol, both the client and server know they'll understand each other.

HTTP ("HyperText Transfer Protocol") is the predominant protocol for Internet requests and responses (e.g. webpages, web resources, web APIs).

HTTP Request Format

```
GET / HTTP/1.0
Host: www.google.com
...
[BLANK LINE]
{request body?}
```

The first line is the request line. It specifies general information about the kind of request and the protocol version.
Following that is a list of headers, 1 per line, and sometimes a payload in the body.

HTTP Request Format

```
GET /posts?sort=recent&limit=10 HTTP/1.0
```

The path can have *query parameters*; these are key-value pairs that appear after the "?" that can specify additional information about the request.

HTTP Response Format

```
HTTP/1.0 200 OK
Content-Type: text/html
[BLANK LINE]
{response body}
```

The first line is the status line. It specifies general information about how the request was handled and the protocol version. Following that is a list of headers, 1 per line, and the payload in the body.

HTTP Response Format

HTTP response payloads contain the requested data. The payload format could be:

- HTML (a webpage) for a browser
- an image, file or other non-text data
- JSON - "Javascript Object Notation": common text format for data types like maps, lists, strings, etc. Used for sending data that can be parsed by another program.
- or others (XML, etc.)

Demo: HTTP Requests/Responses using your browser and **telnet**

Browser and Telnet

We can play around with HTTP requests and responses using browser tools and telnet.

- Browser developer tools show all HTTP requests and responses being sent for us
- telnet lets us "phone" a server and manually send/receive HTTP requests/responses

Both will be useful for testing assign6!

Plan For Today

- **Recap:** Protocols and HTTP
- **HTTP Client Example: wget**
- **HTTP Server Example:** scrabble

wget

wget is a command line utility that, given a URL, downloads a single document (HTML document, image, video, etc.) and saves a copy of it to the current working directory.

- Let's see a quick demo
- **wget** works by sending an HTTP GET request to the specified URL!
- We can implement our own version called **web-get** that relies on our knowledge of HTTP requests and responses to do the same thing.

web-get

web-get is a program that, given a URL, downloads a single document (HTML document, image, video, etc.) and saves a copy of it to the current working directory.

1. parse the specified URL into the host and path components
2. Send an HTTP GET request to the server for that resource
3. Read through the server's HTTP response and save its payload data to a file

web-get

Step 1: parse the specified URL into the host and path components

```
 1 int main(int argc, char *argv[]) {
 2   if (argc != 2) {
 3     cerr << "Usage: " << argv[0] << " <url>" << endl;
 4     return kWrongArgumentCount;
 5   }
 6
 7   // string pair of <host, path>
 8   pair<string, string> hostAndPath = parseURL(argv[1]);
 9   fetchContent(hostAndPath.first, hostAndPath.second);
10   return 0;
11 }
```

web-get

Step 1: parse the specified URL into the host and path components

```
 1 static pair<string, string> parseURL(string url) {
 2   // If the URL starts with the protocol e.g. http://, remove it
 3   if (startsWith(url, kProtocolPrefix)) {
 4     url = url.substr(kProtocolPrefix.size());
 5   }
 6
 7   // Search for the first /
 8   size_t found = url.find('/');
 9
10   // If there is none, the path should be /
11   if (found == string::npos) return make_pair(url, "/");
12
13   // Otherwise, the host is what is before the /, and the path is after the /
14   string host = url.substr(0, found);
15   string path = url.substr(found);
16   return make_pair(host, path);
17 }
```

web-get

Step 2: Send an HTTP GET request to the server for that resource

```
 1 int main(int argc, char *argv[]) {
 2   if (argc != 2) {
 3     cerr << "Usage: " << argv[0] << " <url>" << endl;
 4     return kWrongArgumentCount;
 5   }
 6
 7   // string pair of <host, path>
 8   pair<string, string> hostAndPath = parseURL(argv[1]);
 9   fetchContent(hostAndPath.first, hostAndPath.second);
10   return 0;
11 }
```

web-get

Step 2: Send an HTTP GET request to the server for that resource

```
1  static void fetchContent(const string& host, const string& path) {
2    // Create a connection to the server on the HTTP port
3    int socketDescriptor = createClientSocket(host, kDefaultHTTPPort);
4    if (socketDescriptor == kClientSocketError) {
5      cerr << "Count not connect to host named \"" << host << "\"." << endl;
6      return;
7    }
8
9    sockbuf socketBuffer(socketDescriptor);
10   iosockstream socketStream(&socketBuffer);
11
12   // Send our request (using HTTP/1.0 for simpler requests)
13   socketStream << "GET " << path << " HTTP/1.0\r\n";
14   socketStream << "Host: " << host << "\r\n";
15   socketStream << "\r\n" << flush;
16
17   readResponse(socketStream, getFileName(path));
18 }
```

Note: 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'. We must also flush!

web-get.cc

web-get

Step 3: Read through the server's HTTP response and save its payload data to a file

- The server's response will contain a status line, headers, and a payload
- We don't actually care about the status line or headers in this case - let's skip them. We must read them in (even if we don't need them) in order to get to the payload.
- Once we get to the payload, we can save that part to a file

web-get

Step 3: Read through the server's HTTP response and save its payload data to a file

```
1  static void readResponse(iosockstream& socketStream, const string& filename) {
2    // Skip the status line and headers (we don't need any information from them)
3    while (true) {
4      string line;
5      getline(socketStream, line);
6      if (line.empty() || line == "\r") break;
7    }
8
9    readAndSavePayload(socketStream, filename);
10 }
```

We keep reading lines until we encounter one that is empty or "\r" (getline consumes the \n). That means we have gotten to the payload. We include line.empty() in case the server forgot the "\r".

web-get

Step 3: Read through the server's HTTP response and save its payload data to a file

```
1  static void readAndSavePayload(iosockstream& socketStream, const string& filename) {
2    ofstream output(filename, ios::binary); // don't assume it's text
3    size_t totalBytes = 0;
4    while (!socketStream.fail()) {
5      char buffer[kBufferSizeBytes] = {'\0'};
6      socketStream.read(buffer, sizeof(buffer));
7      totalBytes += socketStream.gcount();
8      output.write(buffer, socketStream.gcount());
9    }
10   cout << "Total number of bytes fetched: " << totalBytes << endl;
11 }
```

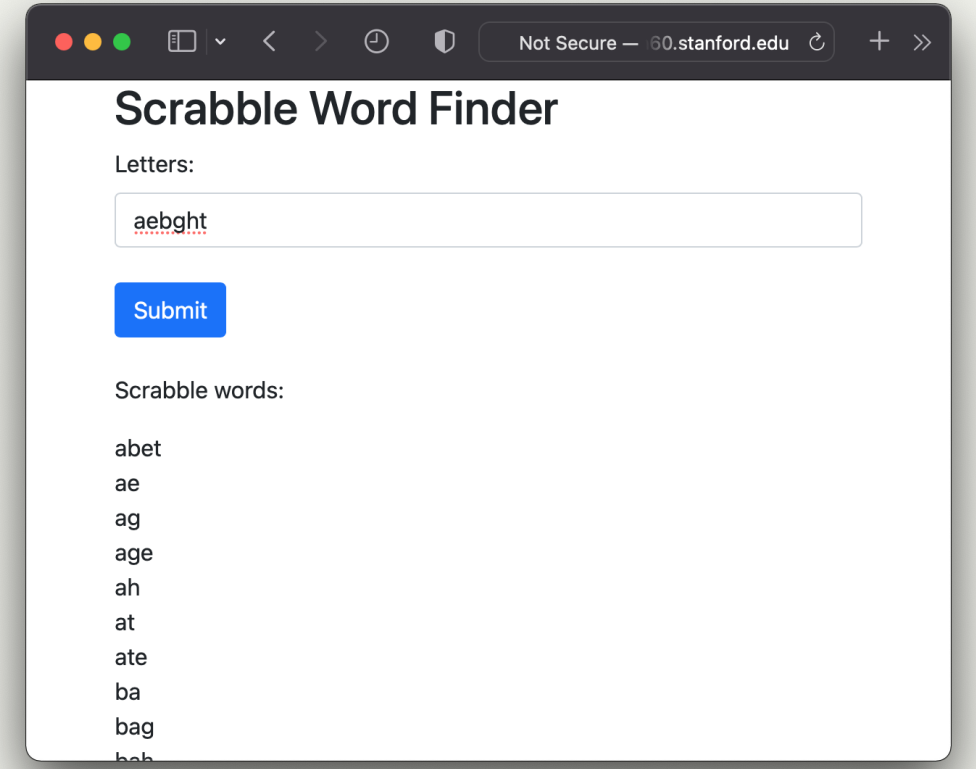
We won't focus too much on the intricacies of this function, but it reads the rest of the response as binary data, and saves it to a file in chunks. Once the server sends the payload, it closes its end of the connection which the client sees as "EOF".

Plan For Today

- **Recap:** Protocols and HTTP
- **HTTP Client Example:** wget
- **HTTP Server Example: scrabble**

HTTP Server: Scrabble Word Finder

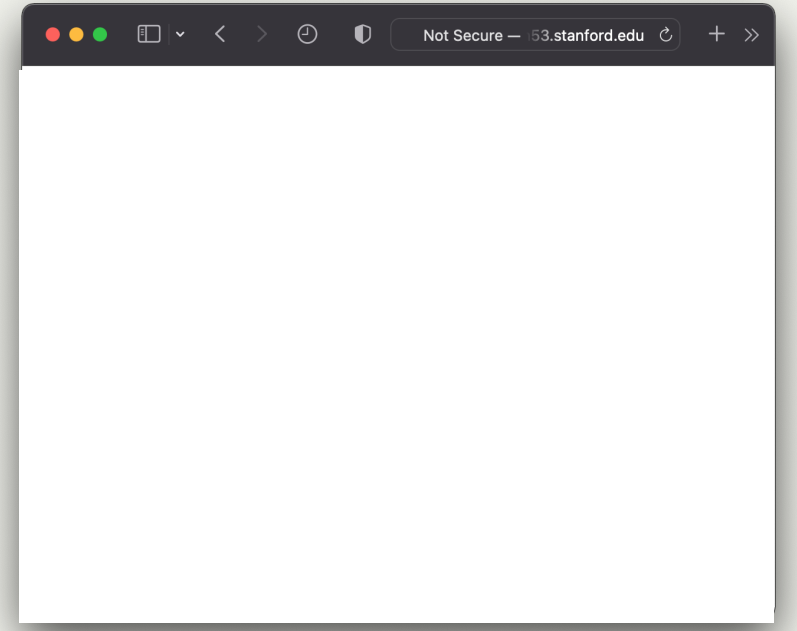
Let's write a web application for finding valid scrabble words given certain letters.



scrabble-word-finder-server.cc

Web Applications

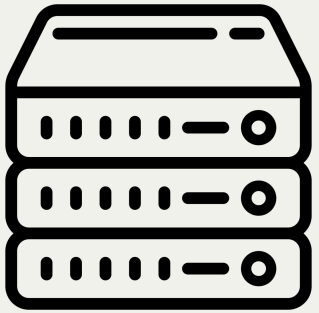
Client



BROWSER: scrabble-words.com, please!

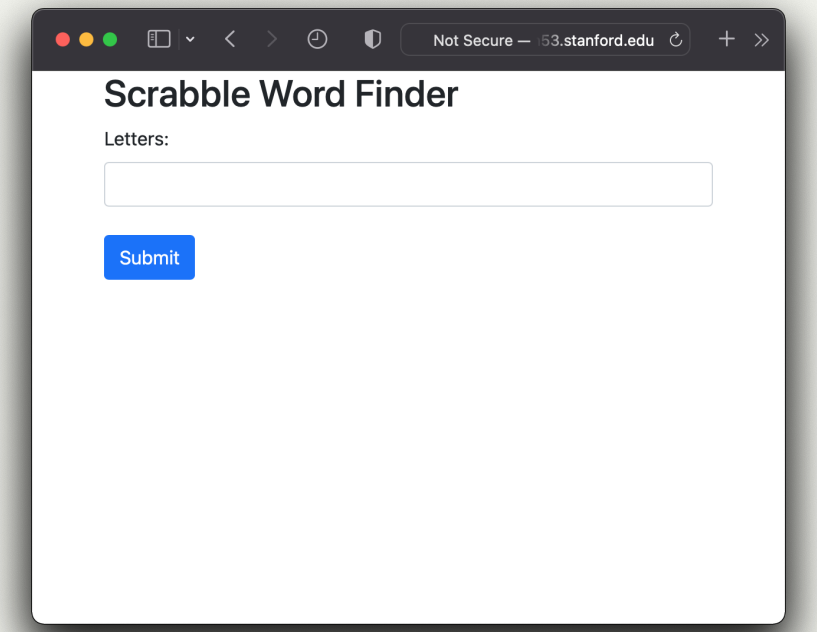
sure, here's HTML for that page.

Server



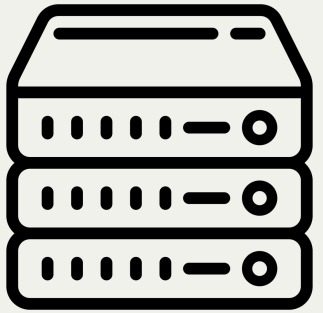
Web Applications

Client

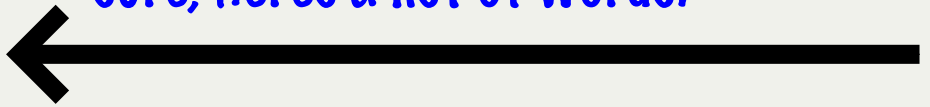


Server

WEBPAGE: scrabble words for "aebght", please!

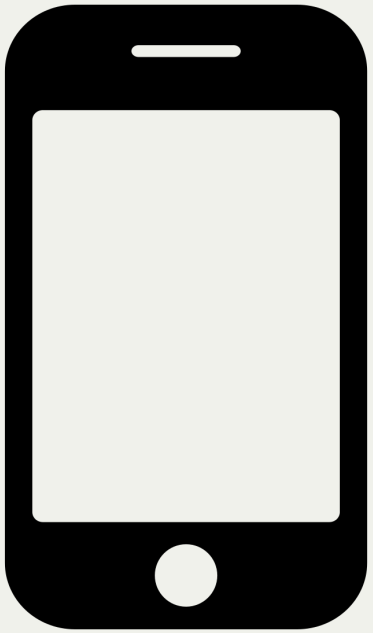


sure, here's a list of words.



Mobile Applications

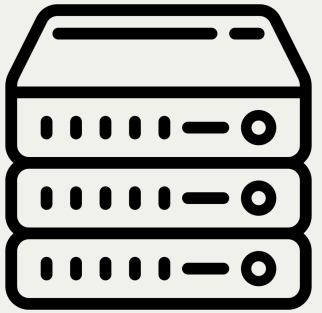
Client



APP: scrabble words for "aebght", please!

sure, here's a list of words.

Server



Web Applications and APIs

- A web server can handle different types of requests. Some can send back HTML for a browser, others can be for non-HTML data for programs or webpages to parse.
- A web application is like a "dynamic webpage" - the page can make more requests to the server while the user interacts with it.
- A web **API** (Application Programming Interface) is the list of request types that a given server can handle
  - More generally: an API is a set of functions one can use in order to build a larger piece of software.
  - APIs can be functions you import (like #include <stdio.h>) or types of requests servers can respond to (like the NASA API, or other web APIs like this or this).
  - Any kind of program can send/receive HTTP requests - webpages, apps, etc. When building a product, you may have the same server API used by your webpage and app.

Recap

- **Recap:** Protocols and HTTP
- **HTTP Client Example:** wget
- **HTTP Server Example:** scrabble

Next time: implementing scrabble server, and learning about how **createClientSocket** and **createServerSocket** are implemented.