CS110 Lecture 23: APIs and Networking Functions

CS110: Principles of Computer Systems

Winter 2021-2022

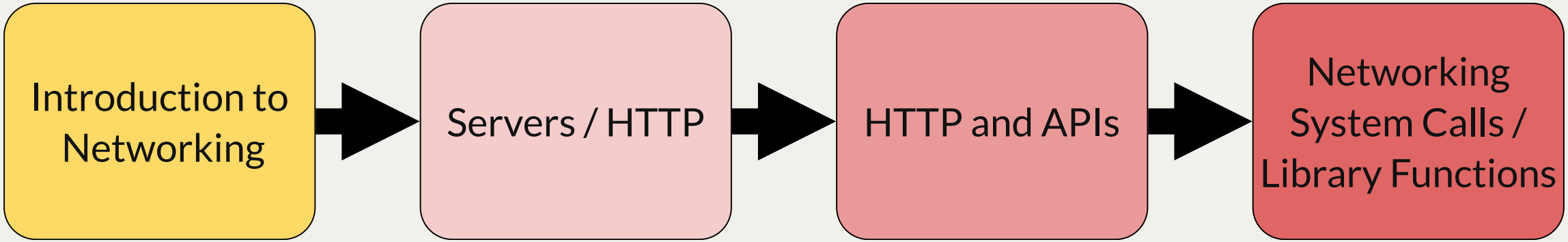
Stanford University

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
- Write a server program that sends HTTP responses and supports a web application
- Learn about the implementations of createClientSocket and createServerSocket

Plan For Today

- **Recap: wget** and Web APIs
- **HTTP Server Example:** scrabble
- Implementing **createClientSocket**

Plan For Today

- **Recap: wget** and Web APIs
- **HTTP Server Example:** scrabble
- Implementing **createClientSocket**

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 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

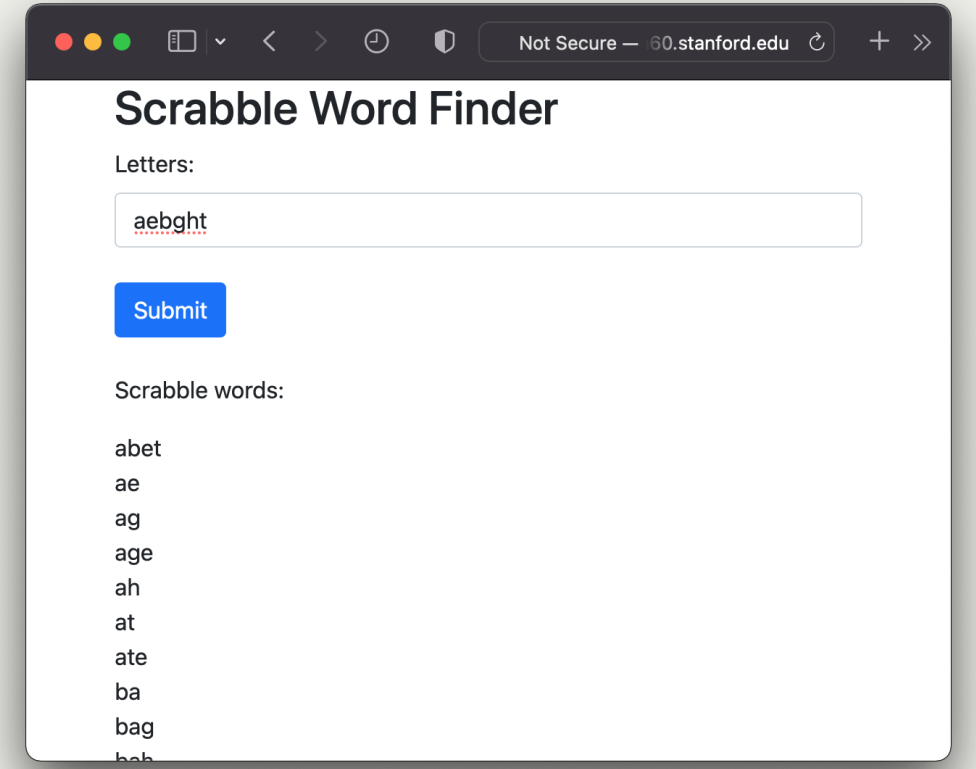
Step 3: Read through the server's HTTP response and save its payload data to a file

```
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".

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 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.
- How do you design an API? Similar question: how do you decide what functions your program will have? And which are public or private?
    - For Web APIs, like for function APIs, it can be useful to browse other publicly available APIs for design patterns.

Plan For Today

- **Recap: wget** and Web APIs
- **HTTP Server Example: scrabble**
- Implementing **createClientSocket**

Scrabble Word Finder

We are going to build a web application that lets users find valid scrabble words given certain letters.

- We will write a web server that can respond to 2 different types of requests:
  1. requesting "/" will send back HTML for the web application homepage
  2. requesting "/words?letters=[LETTERS]" will send back non-HTML text with a list of valid words using those letters
- The HTML sent back for request type #1 will contain code that sends a request of type #2 whenever the user clicks the "submit" button.
- Other developers could also use request type #2 (our *API)* in their programs or websites to get a list of words given specified letters.

`scrabble-word-finder-server.cc`

Scrabble Word Finder Server

scrabble-word-finder-server is a server that can respond to requests for an HTML page, and requests for a list of words given specified letters.

1. open a server socket and listen for incoming HTTP requests
2. when we receive a request, parse it to determine whether its path is "/" (meaning we should send back an HTML page) or "/words" (we should send back a list of words)
3. If it's for "/", read in the file "scrabble-word-finder.html" and send the HTML back in an HTTP response
4. If it's for "/words", compute a list of valid words using the letters in the query params and send it back in an HTTP response in JSON format

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 1: open a server socket and listen for incoming HTTP requests

```
1  int main(int argc, char *argv[]) {
2    unsigned short port = atoi(argv[1]);
3    int serverSocket = createServerSocket(port);
4
5    cout << "Server listening on port " << port << "." << endl;
6    ThreadPool pool(kNumThreads);
7    while (true) {
8      int clientDescriptor = accept(serverSocket, NULL, NULL);
9
10     pool.schedule([clientDescriptor]() {
11       sockbuf socketBuffer(clientDescriptor); // destructor closes socket
12       iosockstream socketStream(&socketBuffer);
13       handleRequest(socketStream);
14     });
15   }
16
17   return 0;
18 }
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 2: when we receive a request, parse it to see whether its path is "/" or "/words?...".

```
1  static void handleRequest(iosockstream& socketStream) {
2    string method;       // e.g. GET
3    string path;         // e.g. /letters
4    string protocol;     // e.g. HTTP/1.0
5
6    socketStream >> method >> path >> protocol;
7
8    // Extract just the query params, e.g. "key=value" in "/url?key=value"
9    size_t queryParamsStart = path.find("?");
10   string queryParams = "";
11   if (queryParamsStart != string::npos) {
12     queryParams = path.substr(queryParamsStart + 1);
13     path = path.substr(0, queryParamsStart);
14   }
15
16   ...
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 2: when we receive a request, parse it to see whether its path is "/" or "/words?...".

```
1   ...
2
3   // read in the rest of the lines/headers, though we don't need it for anything
4   string newline;
5   getline(socketStream, newline);
6
7   while (true) {
8     string line;
9     getline(socketStream, line);
10    if (line.empty() || line == "\r") break;
11  }
12
13  if (path == "/") {
14    ...
15  } else if (...)
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 3: If it's for "/", read in the file "scrabble-word-finder.html" and send the HTML back.

```
1   // The payload and its format differ depending on what was requested
2   string payload;
3   string contentType;
4
5   if (path == "/") {
6     // send back HTML file
7     ifstream fileStream("scrabble-word-finder.html");
8     std::stringstream fileStringStream;
9     fileStringStream << fileStream.rdbuf();
10    payload = fileStringStream.str();
11    contentType = "text/html; charset=UTF-8";
12  } else if (...) {
13    ...
14  }
15
16  sendResponse(socketStream, payload, contentType);
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 3: If it's for "/", read in the file "scrabble-word-finder.html" and send the HTML back.

```
1 static void sendResponse(iosockstream& socketStream, const string& payload, const string& contentType) {
2   socketStream << "HTTP/1.1 200 OK\r\n";
3   socketStream << "Content-Type: " << contentType << "\r\n";
4   socketStream << "Content-Length: " << payload.size() << "\r\n";
5   socketStream << "\r\n";
6   socketStream << payload << flush;
7 }
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 4: If it's for "/words?letters=XXXX", compute a list of valid words with those letters and send it back in JSON format.

```
// The payload and its format differ depending on what was requested
string payload;
string contentType;

if (path == "/") {
  ...
} else if (path == "/words" && queryParams.find("letters=") != string::npos) {
  // compute valid words with these letters and send them back as JSON
  string letters = queryParams.substr(queryParams.find("letters=") + string("letters=").length());
  sort(letters.begin(), letters.end());
  vector<string> formableWords;
  findFormableWords(letters, formableWords);
  payload = constructJSONPayload(formableWords);
  contentType = "text/javascript; charset=UTF-8";
} else {
  ...
}
sendResponse(socketStream, payload, contentType);
```

scrabble-word-finder-server.cc

```
{
  "possibilities": ["word1", "word2"]
}
```

Scrabble Word Finder Server

We need a way to get a list of valid words given a set of characters.

- We could write the code in the server itself to do this - but there's an alternative.
- Sometimes we may have a provided *executable* program that does what we need.
- Here let's say we have a command called **scrabble-word-finder** that takes letters and prints out words with those letters, one word per line
- **Question:** how can we leverage this program's functionality and not re-implement it?
- **Idea**: let's use *subprocess()* from multiprocessing to run it in a child process and capture its output!
- Example of abstraction - client doesn't know how server works, server doesn't know how word finder code works.
- Way to "wrap an executable with a server to make it available to clients"

subprocess

We have implemented a custom function called **subprocess:**

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

subprocess spawns a child process to run the specified command, and can optionally set up pipes we can use to write to the child's STDIN and/or read from its STDOUT.

It returns a struct containing:

- the PID of the child process
- a file descriptor we can use to write to the child's STDIN (if requested)
- a file descriptor we can use to read from the child's STDOUT (if requested)

Scrabble Word Finder Server

Step 4: Otherwise, compute a list of valid words with those letters and send it back in JSON format.

```
 1 static void findFormableWords(const string& letters, vector<string>& formableWords) {
 2   // Make an argument array for the command subprocess should run
 3   const char *command[] = {"./scrabble-word-finder", letters.c_str(), NULL};
 4   subprocess_t sp = subprocess(const_cast<char **>(command), false, true);
 5
 6   // Make a stream around the file descriptor so we can read lines with getline
 7   stdio_filebuf<char> inbuf(sp.ingestfd, ios::in);
 8   istream instream(&inbuf);
 9   while (true) {
10     // Read the next line and add it to the list of formable words
11     string word;
12     getline(instream, word);
13     if (instream.fail()) break;
14     formableWords.push_back(word);
15   }
16   // Make sure to only return from this function once the process has finished
17   waitpid(sp.pid, NULL, 0);
18 }
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 4: Otherwise, compute a list of valid words with those letters and send it back in JSON format.

```
1   // The payload and its format differ depending on what was requested
2   string payload;
3   string contentType;
4
5   if (path == "/") {
6     ...
7   } else if (path == "/words" && queryParams.find("letters=") != string::npos) {
8     // compute valid words with these letters and send them back as JSON
9     string letters = queryParams.substr(queryParams.find("letters=") + string("letters=").length());
10    sort(letters.begin(), letters.end());
11    vector<string> formableWords;
12    findFormableWords(letters, formableWords);
13    payload = constructJSONPayload(formableWords);
14    contentType = "text/javascript; charset=UTF-8";
15  } else {
16    ...
17  }
18  sendResponse(socketStream, payload, contentType);
```

scrabble-word-finder-server.cc

Scrabble Word Finder Server

Step 4: Otherwise, compute a list of valid words with those letters and send it back in JSON format.

```
1  static string constructJSONPayload(const vector<string>& possibilities) {
2     /* An ostringstream is like cout, but it doesn't print to the screen;
3     * when you're done adding to the stream, you can convert it to a
4     * string.
5     */
6    ostringstream payload;
7    payload << "{" << endl;
8    payload << "  \"possibilities\": [";
9
10   // Append each word, followed by a comma for all but the last word
11   for (size_t i = 0 ; i < possibilities.size(); i++) {
12     payload << "\"" << possibilities[i] << "\"";
13     if (i < possibilities.size() - 1) payload << ", ";
14   }
15
16   payload << "]" << endl << "}" << endl;
17   return payload.str();
18 }
```

scrabble-word-finder-server.cc

```
{
   "possibilities": ["word1", "word2"]
}
```

Scrabble Word Finder Server

Step 4: Otherwise, compute a list of valid words with those letters and send it back in JSON format.

```
1 static void sendResponse(iosockstream& socketStream, const string& payload, const string& contentType) {
2   socketStream << "HTTP/1.1 200 OK\r\n";
3   socketStream << "Content-Type: " << contentType << "\r\n";
4   socketStream << "Content-Length: " << payload.size() << "\r\n";
5   socketStream << "\r\n";
6   socketStream << payload << flush;
7 }
```

scrabble-word-finder-server.cc

Scrabble Word Finder HTML

```
<form action="javascript:void(0);">
    <label for="letters" class="form-label">Letters:</label>
    <input type="text" class="form-control" id="letters" name="letters"><br>
    <button type="submit" class="btn btn-primary" onclick="getWords()">Submit</button>
</form>
<br />
<div id="scrabbleWords"></div>

<script>
    function getWords() {
        let letters = document.getElementById("letters").value;
        let result = fetch("/words?letters=" + letters, {method:"GET"}).then(data => {
            return data.json()
        }).then(res => {
            possibilitiesStr = "";
                for (var i = 0; i < res.possibilities.length; i++) {
                    possibilitiesStr += res.possibilities[i]+"<br>";
                }
                document.getElementById("scrabbleWords").innerHTML = "<p>Scrabble words:</p>" + possibilitiesStr;
        }).catch(error =>
            console.log(error)
        )
    }
</script>
```

scrabble-word-finder-server.cc

HTTP Key Takeaways

- 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.
- **HTTP** ("HyperText Transfer Protocol") is the predominant protocol for Internet requests and responses (e.g. webpages, web resources, web APIs).
- HTTP can be used to respond with data in any format: HTTP, JSON, images, etc.
- You should know the core components of requests and responses (request lines, headers, status line, payloads, etc.) but don't get too caught up in the specifics of different headers and other smaller details.
- There are many libraries for easily making HTTP requests and responses - we often don't code them out manually.
- On assign6, you'll be intercepting HTTP requests, possibly modifying them slightly, forwarding them to a server, and delivering the response back to the original client.

Plan For Today

- **Recap: wget** and Web APIs
- **HTTP Server Example:** scrabble
- Implementing **createClientSocket**

createClientSocket and createServerSocket

Let's see the underlying system calls and library functions needed to implement createClientSocket and createServerSocket!

- Goal: to see the kinds of functions required (you won't have to re-implement createClientSocket or createServerSocket)
- Goal: to see the design decisions and language workarounds involved

Clients

We have used **createClientSocket** in client programs so far to connect to servers. It gives us back a descriptor we can use to read/write data.

```
1  int main(int argc, char *argv[]) {
2    // Open a connection to the server
3    int socketDescriptor = createClientSocket("myth64.stanford.edu", 12345);
4
5    // Read in the data from the server (sockbuf descructor closes descriptor)
6    sockbuf socketBuffer(socketDescriptor);
7    iosockstream socketStream(&socketBuffer);
8    string timeline;
9    getline(socketStream, timeline);
10
11   // Print the data from the server
12   cout << timeline << endl;
13
14   return 0;
15 }
```

But how is the **createClientSocket** helper function actually implemented?

createClientSocket

```
int createClientSocket(const string& host, unsigned short port);
```

1. Check that the specified server and port are valid
2. Create a new socket descriptor
3. Associate this socket descriptor with a connection to that server
4. Return the socket descriptor

createClientSocket

```
int createClientSocket(const string& host, unsigned short port);
```

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**
4. Return the socket descriptor

createClientSocket

```
int createClientSocket(const string& host, unsigned short port);
```

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**
4. Return the socket descriptor

createClientSocket

- We check the validity of the host by attempting to look up their IP address
- **gethostbyname()** gets IPV4 host info for the given name (e.g. "www.facebook.com")
- **gethostbyname2()** can get IPV6 host info for the given name - second param can be AF_INET (for IPv4) or AF_INET6 (for IPv6)
- **gethostbyaddr()** gets host info for the given IPv4 address (e.g. "31.13.75.17")
  - First argument is the base address of a character array with ASCII values of 171, 64, 64, and 137 in **network byte order.** For IPv4, the second argument is usually **`sizeof(struct in_addr)`** and the third the **`AF_INET`** constant.
- These are technically deprecated in favor of **getAddrInfo**, but still prevalent and good to know
- All return a statically allocated **`struct hostent`** with host's info (or NULL if error)

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyname2(const char *name, int af);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

createClientSocket

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyname2(const char *name, int af);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Wait a minute - **gethostbyname** and **gethostbyname2** will give back different info (IPv4 vs. IPv6 addresses). How can the return type be the same?

- **Key Idea: struct hostent** will have a generic field in it which is a list of addresses; depending on whether it's IPv4 or IPv6, the list will be of a different type, and we can cast it to that type.

gethostbyname()

```
// represents an IP Address
struct in_addr {
    unsigned int s_addr // stored in network byte order (big endian)
};

// represents a host's info
struct hostent {
    // official name of host
    char *h_name;

    // NULL-terminated list of aliases
    char **h_aliases;

    // host address type (typically AF_INET for IPv4)
    int h_addrtype;

    // address length (typically 4, or sizeof(struct in_addr) for IPv4)
    int h_length;

    // NULL-terminated list of IP addresses
    // This is really a struct in_addr ** when hostent contains IPv4 addresses
    char **h_addr_list;
};
```

Note: `h_addr_list` is typed to be a **`char *`** array, but for IPv4 records it's really **`struct in_addr **`**, so we cast it to that in our code.

Why the confusion?

- **h_addr_list** needs to represent an array of pointers to IP addresses.
- **struct hostent** must be generic and work with e.g. both IPv4 and IPv6 hosts.
- Thus, **h_addr_list** could be an array of **in_addr *s** (IPv4) or an array of **in6_addr *s** (IPv6).
- No void * back then, so char ** it is.

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**

```
int createClientSocket(const string& host, unsigned short port) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) return -1;
    ...
```

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**

```
1  int socket(int domain, int type, int protocol);
2
3  int createClientSocket(const string& host, unsigned short port) {
4      ...
5      int s = socket(AF_INET, SOCK_STREAM, 0);
6      if (s < 0) return -1;
7      ...
```

The **socket** function creates a socket endpoint and returns a descriptor.

- The first parameter is the protocol family (IPv4, IPv6, Bluetooth, etc.).
- The second parameter is the type of the connection - do we want a reliable 2-way connection, unreliable connection, etc.?
- The third parameter is the protocol (0 for default)

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**

```
int connect(int clientfd, const struct sockaddr *addr, socklen_t addrlen);
```

connect connects the specified socket to the specified address.

- Wait a minute - we could be using IPv4 or IPv6. How can we have the same parameter types for both?

connect()

```
int connect(int clientfd, const struct sockaddr *addr, socklen_t addrlen);
```

There are actually **multiple** different types of we may want to pass in. **sockaddr_in** and **sockaddr_in6**. How can we handle these possibilities? C doesn't support inheritance or templates.

- First idea: we could make a new version of **connect** for each type (not great)
- Second idea: we could specify the parameter type as **void** * (but then how would we know the real type?)
- Third idea: we could make the parameter type a "parent type" called **sockaddr**, which will have the same memory layout as **sockaddr_in** and **sockaddr_in6**.
  - Its structure is a 2 byte *type* field followed by 14 bytes of *something*.
  - Both **sockaddr_in** and **sockaddr_in6** will start with that 2 byte type field, and use the remaining 14 bytes for whatever they want.
  - **connect** can then check the type field before casting to the appropriate type

connect()

```
int connect(int clientfd, const struct sockaddr *addr, socklen_t addrlen);
```

We will make the parameter type a "parent type" called **sockaddr**, which will have the same memory layout as **sockaddr_in** and **sockaddr_in6**. Its structure is a 2 byte *type* field followed by 14 bytes of *something*. Both **sockaddr_in** and **sockaddr_in6** will start with that 2 byte type field, and use the remaining 14 bytes for whatever they want.

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

sockaddr_in

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

- The **`sin_family`** field should always be initialized to be **`AF_INET`** for IPv4 to distinguish what struct type it really is.
- The **`sin_port`** field stores a port number in **network byte order**.
  - Different machines may store multi-byte values in different orders (big endian, little endian). But network data must be sent in a consistent format.
- The **`sin_addr`** field stores the IPv4 address
- The **`sin_zero`** field represents the remaining 8 bytes that are unused.

sockaddr_in6

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

- The **`sin6_family`** field should always be initialized to be **`AF_INET6`** for IPv6 to distinguish what struct type it really is.
- The **`sin6_port`** field stores a port number in **network byte order**.
- The **`sin6_addr`** field stores the IPv6 address
- **`sin6_flowinfo`** and **`sin6_scope_id`** are beyond the scope of what we need, so we'll ignore them.

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**

```
1  int createClientSocket(const string& host, unsigned short port) {
2      ...
3      struct sockaddr_in address;
4      memset(&address, 0, sizeof(address));
5      address.sin_family = AF_INET;
6      address.sin_port = htons(port);
7
8      // h_addr is #define for h_addr_list[0]
9      address.sin_addr = *((struct in_addr *)he->h_addr);
10     if (connect(s, (struct sockaddr *) &address, sizeof(address)) == 0) return s;
11     ...
```

htons is "host to network short" - it converts to network byte order, which may or may not be the same as the byte order your machine uses.

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**
4. Return the socket descriptor

```
1  int createClientSocket(const string& host, unsigned short port) {
2      struct hostent *he = gethostbyname(host.c_str());
3      if (he == NULL) return -1;
4      int s = socket(AF_INET, SOCK_STREAM, 0);
5      if (s < 0) return -1;
6      struct sockaddr_in address;
7      memset(&address, 0, sizeof(address));
8      address.sin_family = AF_INET;
9      address.sin_port = htons(port);
10
11     // h_addr is #define for h_addr_list[0]
12     address.sin_addr = *((struct in_addr *)he->h_addr);
13     if (connect(s, (struct sockaddr *) &address, sizeof(address)) == 0) return s;
14
15     close(s);
16     return -1;
17 }
```

Recap

- **Recap: wget** and Web APIs
- **HTTP Server Example:** scrabble
- Implementing **createClientSocket**

Next time: Distributed systems and MapReduce