CS110 Spring 2021

Lecture 14: Network Clients & Servers

Principles of Computer Systems

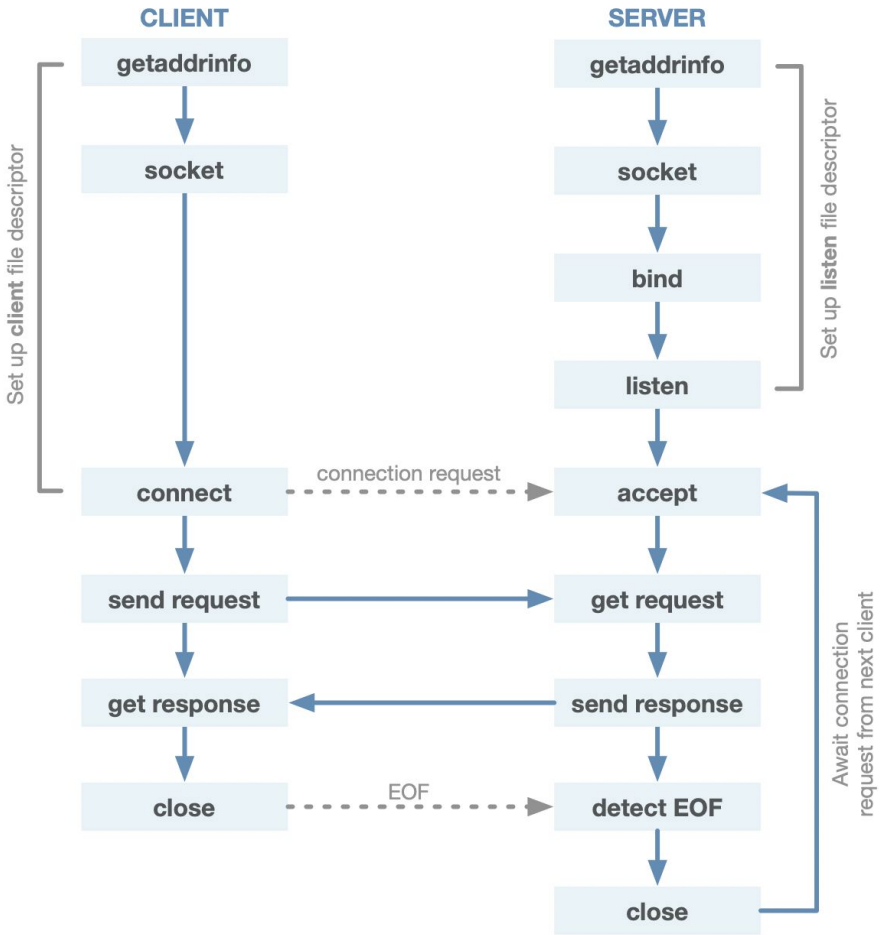
Stanford University, Dept. Of Computer Science
Lecturer: Jerry Cain & Roz Cyrus

Version 1

Accessing Code Examples

- Today's lecture examples reside within:
  `/usr/class/cs110/lecture-examples/networking`.
  - First **`ssh`** into a myth machine (ssh yourusername@myth.stanford.edu). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, **`cd`** into the above directory.
  - To get started, type:
    `git clone /usr/class/cs110/lecture-examples cs110-lecture-examples`
    at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type **`git pull`**. Doing so will update your local copy to match whatever the master has become.

Sockets Interface Recap

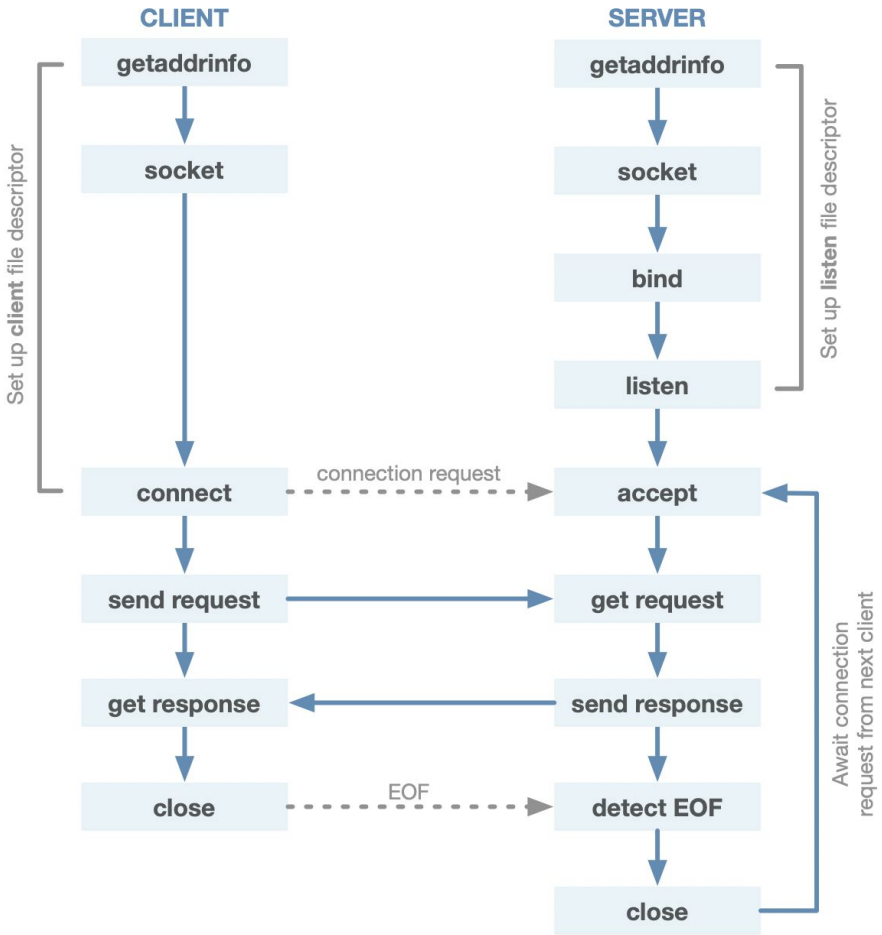


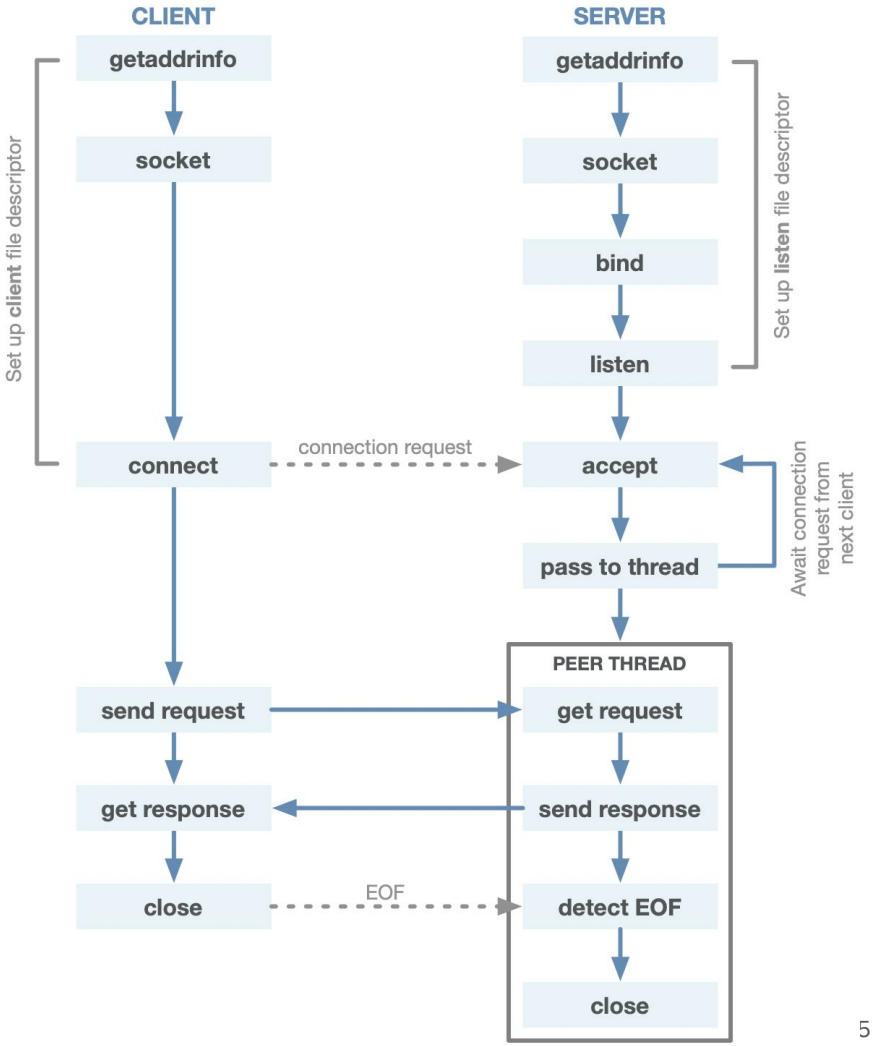
Networking Recap


```
int main(int argc, char *argv[]) {
    int server = createServerSocket(12345);
    ThreadPool pool(4);
    while (true) {
        int client = accept(server, NULL, NULL); // the two NULLs could instead be used
                                                 // to surface the IP address of the client
        pool.schedule([client] { publishTime(client); });
    }
    return 0;
}
```

- Multithreading can significantly improve the performance of networked applications.
- Our time server can benefit from multithreading as well. The work a server needs to do in order to meet the client's request might be time consuming—so time consuming, in fact, that the server is slow to iterate and accept new client connections.
- As soon as **`accept`** returns a socket descriptor, spawn a child thread—or reuse an existing one within a **`ThreadPool`**—to get any intense, time consuming computation off of the main thread. The child thread can use a second processor or a second core, and the main thread can quickly move on to its next **`accept`** call.
- Here's a new version of our time server, which uses a **`ThreadPool`** (which you're implementing for Assignment 5) to get the computation off the main thread.

Sockets Interface Recap





Networking: 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.
- 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) // no pathname found
        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;
}
```

Networking: wget

- **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.

Networking: wget

- 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.

Networking: wget

- **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. 🙂

Networking: wget

- 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.

Next Up: APIs!

APIs

- 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.

Networking: Word Finder

- I want to implement an API server that's architecturally in line with the way Google, Twitter, Facebook, and Instagram 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 **`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.
  - **`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:

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

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

Networking: Word Finder

- 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'
  ]
}
```

Networking: Word Finder

- 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);
```

Networking: Word Finder

- 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
}
```

Networking: Word Finder

- 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 time, 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 few 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.

Networking: Word Finder


```
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());
}
```

Networking: Word Finder

- 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;
}
```

Networking: Word Finder

- 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.
- Our **`scrabble-word-finder-server`** provides 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.

```
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;
}
```

Networking: Word Finder

- Try this in your browser!
  - Run **`./scrabble-word-finder-server`** on a myth machine from the **`/usr/class/cs110/lecture-examples/networking`** directory.
  - Make a note of which myth machine you're connected to, and the port number displayed when you run the program above (which should be port 13133).
  - Then in your browser (make sure you're either on the Stanford network or VPN into the network), type in:
    - http://myth[number]:13133/[stringtoparse]
    - For example, if you connected to myth63 and you want to see all the possible words that can be formed from the string "example", type: http://myth63:13133/example
  - Hit enter and watch the results load!
    - If you reload the first results, you should see "cached" change from false to true if that entry wasn't cached before.

Cool Bonus: Networking Chat

We can use **`netcat`** to send data between two computers.
In one terminal tab, run:

```
myth61$ netcat -lnkv 12345
Listening on 0.0.0.0 12345
```

This will listen for an incoming connection. In the context of servers, 0.0.0.0 means all IPv4 addresses on the local machine. In another tab, run the following, ensuring that the host and port match the myth machine used in the first session above (this can be on another myth or your local computer):

```
myth66$ netcat myth61.stanford.edu 12345
hi!
```

Whatever you enter in one terminal will be visible in the other terminal:

```
myth61$ netcat -lnkv 12345
Listening on 0.0.0.0 12345
Connection received on 99.73.69.255 53146
hi!
```

End of Lecture 14