Project 1: Simple HTTP Client and Server

Revisions

More hints and resources may be added later.

Overview

In this project, you will learn socket programming and the basic of HTTP protocol through developing a simple Web server and client.

All implementations should be written in C++ using BSD sockets. **No high-level network-layer abstractions (like Boost.Asio or similar) are allowed in this project.** You are allowed to use some high-level abstractions, including C++11 extensions, for parts that are not directly related to networking, such as string parsing, multi-threading. We will also accept implementations written in C, however use of C++ is preferred.

Task Description

The project contains two parts: a Web server and a Web client. The Web server accepts an HTTP request, parses the request, and looks up the requested file in the local file system. If the file exists, the server returns the content of the file as part of HTTP response, otherwise returning HTTP response with the corresponding error code.

After retrieving the response from the Web server, the client saves the retrieved file in the local file system.

The basic part of this project only requires you to implement at least HTTP 1.0: the client and server will talk to each other through **non-persistent** connections.

How to approach the project

You may want to approach the project in three stages:

1. Implementing **HTTP message abstraction**
2. Implementing **Web client** module
3. Implementing **Web server** module.

HTTP message abstraction

As the first step, you need to implement several helper classes that can help you to parse and construct an HTTP message, which can be either HTTP request or HTTP response. For example, you may want to implement an `HttpRequest` class that can help you to customize the HTTP request header and encode the HTTP request into a string of bytes of the wire format. Some high-level pseudo code would look like:

```
HttpRequest request;
request.setUrl(...);
request.setMethod(...);
vector<uint8_t> wire = request.endcode();
```

Note that you only needs to implement HTTP `GET` request in this project.

You may also want to implement an `HttpRequest` constructor method that creates an `HttpRequest` object from the wire encoded request.

```
HttpRequest request;
request.consume(wire);
```

Similarly, you may also want to implement an `HttpResponse` class that can facilitate processing HTTP responses.

Web server

After finishing HTTP abstraction, you can start building your Web server. The eventual output is a binary `web-server`, which must accept three command-line arguments: hostname of the Web site, a port number, and a directory name.

```
$ web-server [hostname] [port] [file-dir]
```

For example, the command below should start the Web server with host name `localhost` listening on port `4000` and serving files from the directory `/tmp`.

```
$ ./web-server localhost 4000 /tmp
```

The default arguments for `web-server` must be `localhost`, `4000`, and `.` (current working directory).

The Web server needs to convert the hostname to an IP address and opens a listening socket on the IP address and the specified port number. Through the listening socket, the Web server accepts the connection requests from clients and establishes connections to the clients. Through the established connection, the Web server should receive the HTTP request sent by the client, and return the requested file in terms of HTTP response. The Web server **must** handle concurrent connections. In other words, the web server can talk to multiple clients at the same time.

After implementing the Web server, you can test it by visiting it through some widely used Web clients (e.g., Firefox, wget) on your local system.

Web client

After finishing the Web server, you can start building your Web client. The eventual output is also a binary `web-client`, which accepts multiple URLs as arguments.

```
$ ./web-client [URL] [URL]...
```

For example, the command below should start the Web client that fetches `index.html` file from your webserver:

```
$ ./web-client http://localhost:4000/index.html
```

The Web client first tries to connect to the Web server as specified in the URL. Once the connection is established, the client constructs the HTTP request and sends it to the Web server, expecting a response. After receiving the response, the client needs to parse it to distinguish success or failure codes. Finally, if the file is successfully retrieved, it should be saved in the current directory using the name inferred from the URL.

You can also test your implementation by fetching data from some real websites or the web server you just implemented.

Hints

About HTTP abstractions:

- How many classes you need to create for the HTTP abstraction?
  - Can you use inheritance to reduce your workload?
- If we have the complete HTTP message, it is trivial to decode it.
  - How do we know we have received the complete message? especially for HTTP response?
  - For HTTP GET request, we know it ends with `\r\n\r\n`, but what if we only get part of it from `read` or `recv`, e.g., only `\r`?
  - For HTTP response, is it possible to decode the whole response before we get the complete message?
- You may assume the size of requested files is less than 1GB.
- Your implementation must support three error codes: `200 OK`, `400 Bad request`, `404 Not found`. All the other error codes (e.g., `403 Forbidden`, `501 Not implemented`, `505 HTTP version not supported`) are optional.
- What to return if the HTTP Request has a URL of "/"?
  - If the server has index.html and request is "/index.html", the server MUST return index.html
  - If the server has index.html and request is "/", the server may return index.html or 404, both implementations are correct.
  - If the server does not have index.html and request is "/index.html" or "/", the server MUST return 404.

Here are some hints of using multi-thread techniques to implement the Web server.

- For the Web server, you may have the main thread listening (and accepting) incoming **connection requests**.
  - Any special socket API you need here?
  - How to keep the listening socket receiving new requests?
- Once you accept a new connection, create a child thread for the new connection.
  - Is the new connection using the same socket as the one used by the main thread?
- Note that only HTTP 1.0 is required for the basic part of this project. HTTP 1.0 uses non-persistent connection. In other word, for each connection, only two messages are exchanged: a HTTP request and a HTTP response.
  - Does this assumption simplify the job of the child thread?

If you want to approach the problem using asynchronous programming model, here are some hints:

- Understand the working mechanism of `select` socket API. In fact, you can treat `select` as a monitor of all your connections (even the listening socket).
- Use `select` to figure out what event happened on which connection, and process the event correctly.
    - how to distinguish the listening socket and the others?

There is some sample code included in the archive:

- A simple server that echoes back anything sent by the client: server.cpp, client.cpp
- Domain name resolution: showip.cpp
- A simple multi-thread countdown: multi-thread.cpp
- Asynchronous server using `select`: async-server.cpp, random-client.cpp

Other resources

- Guide to Network Programming Using Sockets

Environment Setup

The best way to guarantee full credit for the project is to do project development using a Ubuntu 14.04-based virtual machine.

You can easily create an image in your favourite virtualization engine (VirtualBox, VMware) using the Vagrant platform and steps outlined below.

Set up Vagrant and create VM instance

Note that all example commands are executed on the host machine (your laptop), e.g., in Terminal.app (or iTerm2.app) on OS X, cmd in Windows, and console or xterm on Linux. After the last step (`vagrant ssh`) you will get inside the virtual machine and can compile your code there.

- Download and install your favourite virtualization engine, e.g., VirtualBox
- Download and install Vagrant tools for your platform
- Set up project and VM instance
    - Clone project template from zipfile provided or create and sync your own git repository (If you are not familiar with `git`, please google it or ask TAs for help)

```
cd ~/ee3d3-proj1
```

    - Initialize VM

```
vagrant up
```

    - To establish an SSH session to the created VM, run

```
vagrant ssh
```

If you are using Putty on Windows platform, `vagrant ssh` will return information regarding the IP address and the port to connect to your virtual machine.

- Work on your project

  Make sure all files in `~/ee3d3-proj1` folder on the host machine will be automatically synchronized with `/vagrant` folder on the virtual machine. For example, to compile your code, you can run the following commands: (If you are not familiar with `make` and `Makefile`, please google them or ask TAs for help)

```
vagrant ssh
cd /vagrant
make
```

Notes

- If you want to open another SSH session, just open another terminal and run `vagrant ssh` (or create a new Putty session).
- If you are using Windows, read this article to help yourself set up the environment (all links at bottom of this document).
- The code base contains the basic `Makefile` and two empty files `web-server.cpp` and `web-client.cpp`.

```
$ vagrant ssh
vagrant@vagrant-ubuntu-trusty-64:~$ cd /vagrant
vagrant@vagrant-ubuntu-trusty-64:/vagrant$ ls
Makefile  README.md  Vagrantfile  web-client.cpp  web-server.cpp
```

- You are now free to add more files and modify the Makefile to make the `web-server` and `web-client` full-fledged implementation.

Submission

Note: ONE AND ONLY ONE team member needs to submit the project for the whole team. To submit your project, you need to prepare:

1. A `.tar.gz` archive named `<UID1-UID2-UID3>.tar.gz`, which MUST have the following files:
   - All source code (all hpp and cpp files)
   - Makefile (no binaries): We will run the `make` command and all the necessary binaries MUST be generated.
   - The client binary MUST be named `web-client`
   - The server binary MUST be named `web-server`
   - You must submit BOTH versions of your client and server code i.e. client and server code for HTTP 1.0 (AND client and server code for HTTP 1.1 if you extend your system to also support HTTP1.1 You are not required to do so, but may choose to). In this case the HTTP 1.0 client and server binaries MUST be named as above while HTTP 1.1

client and server binaries MUST be named `web-client-1.1` and `web-server-1.1` respectively
    - There should be NO sub-directories
2. A brief PDF project report that describes
    - the high level design of your server and client;
    - the problems your ran into and how you solved the problems;
    - additional instructions to build your project (if your project uses some other libraries);
    - how you tested your code and why.
    - the contribution of each team member (up to 3 members in one team) and their UID

Put all these above into a package and submit to Blackboard mymodule.

Please make sure:

1. your code can compile
2. no unnecessary files in the package.
3. you `strictly` follow the above rules

`Otherwise, you will not get any credit.`

Grading

Your code will be first checked by a software plagiarism detecting tool. If we find any plagiarism, you will get zero (0).

Your code will then be automatically tested in some testing scenarios. If your code can pass all our automated test cases, you will get the full credit.

Links

BSD Sockets: http://en.wikipedia.org/wiki/Berkeley_sockets

Select socket API:

https://linux.die.net/man/2/select

Guide to Network Programming using sockets

http://beej.us/guide/bgnet/

VirtualBox:

https://www.virtualbox.org/wiki/Downloads

Vagrant tools:

https://www.vagrantup.com/downloads.html

Vagrant setup for Windows machine:

http://www.sitepoint.com/getting-started-vagrant-windows/