CSc 450/550: Computer Communications and Networks (Summer 2007)

Lab Project 1: A Simple Multi-Thread Web Server

Spec Out: May 11, 2007
Demo Due: May 30, 2007
Code Due: June 1, 2007

1 Introduction

In this project, students will build a simple, multi-thread "Web" server in C or C++. The project will allow students to refresh their C or C++, `socket` and `pthread` programming skills, and to better understand the Hyper-Text Transfer Protocol (HTTP) underneath the World-Wide Web.

2 Background

2.1 `socket` API

`socket` is the Application Programming Interface (API) to network services in many operating systems. For a "server"-like application, normally you will need to use the following system calls.

- `socket()`: to create a new socket
- `bind()`: to associate the socket to a local address
- `listen()`: to wait for an incoming connection request
- `accept()`: to accept the incoming connection request with a new socket
- `recv()`: to read from the socket
- `send()`: to write to the socket
- `close()`: to close the socket and release the resources allocated

You will want to read the manual page (e.g., `man socket`) to better understand these system calls and their input arguments and return values. You may need to use other `socket`-related system calls or auxiliary library calls as well. It is very important to check the return value of these function calls in your program to determine whether the intended operation is successful.

For more information on `socket` programming, please see [1].

2.2 `pthread` Library

`pthread` is the POSIX standards-based API to multi-threading programming in many operating systems. The following library calls are used to create, terminate and synchronize threads.

- `pthread_create()`: to create a new thread
- `pthread_exit()`: to terminate the calling thread
- `pthread_join()`: to suspend the calling thread and to wait for the target thread to finish

All threads in the same process share most data. To provide finer thread synchronization, you may want to use mutual exclusion (`pthread_mutex_lock()` and `pthread_mutex_unlock()`) and condition variables (`pthread_cond_wait()` and `pthread_cond_signal()`) in some cases.

Again, you may want to read the `pthread` manual page (e.g., `man pthread_create`).

For more information on `pthread` programming, please see [2].

2.3 HTTP Protocol

HTTP is a client-server, request-response application-layer protocol for the Web. In this project, only a very simplified version of HTTP/1.0 is to be implemented.

2.3.1 Simplified HTTP Request

As being stated in the HTTP/1.0 specification: "A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use." [3]

For example, when `wget -d www.csc.uvic.ca`, it shows

Message	Explanation (**required** or ignored by the simple web server)
`GET / HTTP/1.0`	**method, request URI, HTTP version (required)**
`User-Agent: Wget/1.10.2`	Request header (ignored)
`Accept: */*`	Request header (ignored)
`Host: www.csc.uvic.ca`	Request header (ignored)
`Connection: Keep-Alive`	Request header (ignored)
	a blank line indicating the end of request headers (required)

The simple web server will read the request line, which includes **method**, **request URI** and **HTTP version** separated by **string delimiters** (e.g., blank spaces or tabs), but will ignore all request headers, until it reaches the end of request headers indicated by **a blank line**.

Method and **HTTP version** are not case sensitive; however, **URI** is case sensitive.

The simple web server only supports the `GET` method, which obtains an HTTP object (often an HTML file) from the web server.

URI identifies the HTTP object. For example, `GET / HTTP/1.0` tries to obtain the default file `index.html` from the root of the web server document directory. If the web server's root document directory is `/tmp/www`, then `/tmp/www/index.html`, if existent, is retrieved and returned by the web server; otherwise, a `Not Found` error message is returned instead. `GET /icons/new.gif` will let the web server retrieve `/tmp/www/icon/new.gif`.

The simple web server only recognizes `HTTP/1.0` as the valid HTTP version supported.

The web server will return a `Bad Request` error message if an invalid request line or an incomplete request message is encountered. The minimal, valid request message contains a valid request line and a blank line indicating the end of the request.

2.3.2 Simplified HTTP Response

As being stated in the HTTP/1.0 specification: "After receiving and interpreting a request message, a server responds in the form of an HTTP response message." [3]

When `wget -d www.csc.uvic.ca`, the web server returns

Message	Explanation (**required** or omitted in the simple web server)
`HTTP/1.1 200 OK`	**HTTP version, status code, reason phrase (required)**
`Date: Wed, 02 May ...`	response header (omitted)
`Server: Apache/2. ...`	response header (omitted)
`X-Powered-By: PHP/4 ...`	response header (omitted)
`Connection: c ...`	response header (omitted)
`Content-Type: text/ ...`	response header (omitted)
	a blank line indicating the end of response headers (required)
index.html content	**HTTP object(s) returned (required, if any)**

The simple web server will return the response line, which includes **HTTP version**, **status code** and **reason phrase** separated by **blank spaces**, but will omit all response headers shown above. However, the simple web server will return **a blank line** indicating the end of response headers and the start of the returned HTTP object, if any.

The response line is not case sensitive, but it is suggested to follow the convention shown above.

The simple web server only returns `HTTP/1.0` as the valid HTTP version supported.

The simple web server only supports the following list of status codes and reason phrases.

Status Code	Reason Phrase	Explanation
`200`	`OK`	good request with the requested object to be returned
`400`	`Bad Request`	bad request not understood by the server
`404`	`Not Found`	good request with no matching request object

If the simple web server can understand the request, and the request object is successfully retrieved, it will return `HTTP/1.0 200 OK` followed by a blank line and the content of the requested object. If the simple web server cannot understand the request, it will return `HTTP/1.0 400 Bad Request` followed by a blank line. If the simple web server can understand the request, but the requested object is not available (e.g., nonexistence, bad file permission, etc), it will return `HTTP/1.0 404 Not Found` followed by a blank line.

The simple web server does not have to support persistent connections.

For more information on HTTP/1.0, please see [3].

3 Design

Students will have the freedom of designing their simple, multi-thread web server to be implemented in C or C++ with `socket` API and `pthread` library. The following is just a possible design.

When the server is invoked, among other things, it will create a "dispatcher" thread to accept incoming TCP connections, which transport the HTTP request and response. It can also create a few "worker" threads in advance to be assigned by the dispatcher to incoming connections, or these threads are created on demand by the dispatcher when a connection comes. The worker thread actually reads and processes the HTTP request, and returns the HTTP response and the requested object if the request is successful (or an error message otherwise). The worker threads could be terminated after an HTTP request is served, or be kept in an idle thread pool for future requests.

4 Requirements

4.1 Basic Features

Basic features are required in all implementations in order to get the full marks for this lab project.

4.1.1 Invoking the Server

The syntax to run the simple web server is

```
./sws <port> <directory>
```

This will invoke the simple web server binary `sws` in the current directory, and instruct the web server to listen at the TCP `port` for incoming connections and to retrieve requested objects under `directory`. If a wrong syntax is used when invoking the server, the server should print out error messages showing the proper usage and exit gracefully.

When it is invoked succesfully, for example, the simple web server will print out the following message if being invoked as `./sws 8080 /tmp/www`

```
sws is running on port 8080 and serving /tmp/www
press 'q' to quit ...
```

The client is not allowed to use `../../` to retrieve objects out of the root document `directory`; such requests will be responded with `HTTP/1.0 400 Bad Request` by the server.

4.1.2 Server Operations

When running, the simple web server should respond to incoming requests at TCP `port` for all network interfaces on the machine running the server.

Once a request is served, the server should print out a log message in the following format.

```
MMM DD HH:MM:SS Client-IP:Client-Port request-line response-line [filename]
```

For example, if the server, on the noon of May 11, successfully served a request `GET / HTTP/1.0` from the client at port `4096` on host `192.168.0.100`, it should print out

```
May 11 12:00:00 192.168.0.100:4096 GET / HTTP/1.0 HTTP/1.0 200 OK /tmp/www/index.html
```

`filename` indicates the location of the HTTP object returned, if the request is successfully served. The log messages should be ordered chronically and not be scrambled due to the fact that there are multiple worker threads.

4.1.3 Multi-Threading

The simple web server should use multi-threading (up to five "worker" threads) to handle multiple, concurrent HTTP requests in a non-blocking fashion. When there are more than five concurrent connection requests, the additional requests (at least five extra requests) should be queued in `listen()`. In other words, the simple web server should be able to handle ten concurrent HTTP requests without dropping any of them.

4.1.4 Terminating the Server

The simple web server should continue serving HTTP requests until the `q` key is pressed in the terminal running the server. When the `q` key is pressed, the server should finish serving all accepted requests, close all created sockets, and release all allocated resources before exiting gracefully.

After terminating the server, or if the server is aborted, some socket resources may still be used by the system. If you rerun the server on the same port immediately (or if there is already a program on the same port), normally you will get a `bind()` error. You can either terminate the other program, wait for a while, use another port, or use `setsockopt()` with level `SOL_SOCKET` and option `REUSEADDR` before `bind()` to allow port reuse.

4.2 Bonus Features

Only a very simplified version of HTTP/1.0 is to be supported by the multi-thread web server. However, students have the option to extend their design and implementation to include more features in the HTTP protocol (e.g., persistent HTTP connection, HTTP request pipelining, etc) and to improve server performance (e.g., caching, etc).

If you want to design and implement a bonus feature, you should contact the course instructor for permission at least one week before the code due date and clearly indicate the feature during project demo and in code submission. The credit for correctly implemented bonus features will not exceed 20% of the full marks for this lab project.

5 Self-Testing

You can test your multi-thread web server with any HTTP/1.0-compliant client, even Telnet. Using Telnet, you can connect to your web server running at `port` on `host` by

```
telnet <host> <port>
```

You can then manually type the HTTP request after Telnet says

```
Escape character is '^]'.
```

And the HTTP response should follow after a correct HTTP request is successfully processed. You can also use multiple Telnet sessions to test the multi-threading feature of your web server.

To assist your testing, you will be provided a gzipped sample document directory file `www.tar.gz`, and you can recover the sample document directory in `/tmp` by

```
mv www.tar.gz /tmp
cd /tmp
tar -zxvf www.tar.gz
```

A sample request sequence `http.request` is also included in `www.tar.gz`

However, your simple web server might be tested against different document directories and request sequences during project demo and code inspection.

6 Demonstration

Your simple web server should be demonstrated in the lab section for which you have registered on the demo due date. **Lab projects not demonstrated will have their code submission not marked. It is expected that your lab project be working at the time of demonstration.** During the project demo, lab instructors will go through a demo checklist together with the student, and then provide the checklist to the student. Students will have the chance to improve their design and implementation until the code due date.

The demo is intended to allow students to demonstrate their projects and to help them improve their design and implementation, not to be coding or debugging assistance, and only focuses on required features (and bonus features if indicated). There is no guarantee on the correctness and grade of the project, which can only be determined after the code inspection.

7 Submission

The entire lab project, including the code and documentation, should be submitted electrically through `csc450l` (`l` is for letter L) at `http://www.csc.uvic.ca/~submit/index.cgi` on or before the code due date.

Only the source code (including header files and Makefile) and documentation (including Readme) should be included in a single `tar.gz` file to be submitted. No object or binary files are included in the submission. If `directory` is your project directory, to create such a gzipped tarball, you can

```
cd direcoty
tar -zcvf p1.tar.gz .
```

This packing and naming convention should be strictly followed to allow your submission to be properly located for grading.

In `directory`, you need to include a `Makefile`, which compiles and builds the final binary executable (`sws`) automatically by typing

```
make
```

The same `Makefile` also removes all object and binary files when you type in

```
make clean
```

All projects will be tested on `linux.csc.uvic.ca`

In `directory`, you also need to include a `Readme` plain text file, which contains your student number, registered lab section and a brief description of your design and code structure, as well as allowed bonus features, if any.

The code itself should be sufficiently self-documented. For more information on acceptable coding style, please see [4].

IMPORTANT: All submitted work should be yours. If you have used anything out there, even a small component in your implementation, you should credit and reference properly, and your contribution can be determined accordingly. For academic integrity policies, please see [5].

8 Marking

This lab project is worth 15% in the final grade of this course for CSc 450 students, and 10% for CSc 550 students.

For mark posting and appeal policies, please see the official course outline at [5].

References

[1] http://beej.us/guide/bgnet/

[2] http://www.llnl.gov/computing/tutorials/pthreads/

[3] http://www.w3.org/Protocols/rfc1945/rfc1945

[4] http://www.csc.uvic.ca/~csc450l/references/Code-Style.html

[5] http://courses.seng.engr.uvic.ca/courses/2007/summer/csc/450