CSc 361: Computer Communications and Networks (Fall 2013)

Programming Assignment 1 (P1): Simple Web Server (SWS)

Spec Out: Sep 17, 2013
Code Due: Oct 10, 2013

1 Introduction

In this programming assignment, you will implement a simple web server (SWS) in C or C++, although C is highly recommended, following a simplified HTTP protocol and using the datagram (DGRAM) socket application programming interface (API) with the User Datagram Protocol (UDP). The purpose of this assignment is to allow you to get refreshed with C or C++ programming, and to get familiar with socket API, HTTP protocol, and the client-server application model.

2 Schedule

In order to help you finish the assignment successfully, the schedule of this assignment has been synchronized with both the lectures and the lab and tutorial modules. There are at least three tutorial modules arranged in the weekly lab sessions during the course of this assignment.

Date	Tutorial Module in CSc 361 Lab
Sep 18/19	P1 Spec go-through, socket programming
Sep 25/26	client-server programming
Oct 2/3	debugging and testing
Oct 9/10	last-minute help (if time permits)

3 Background

3.1 Lab Environment

In our Networks teaching lab (ECS360), each computer is connected to the UVicNet (and then to the Internet) through an Ethernet cable with hostname `n-`*greek* (e.g., `n-beta.csc.uvic.ca`), and also to a Linksys WRT54GL router through two Ethernet cables. The computer is running Linux (Ubuntu), so is the router (OpenWRT. Note that the commands on Ubuntu and OpenWRT may differ slightly in syntax due to the limited capability of OpenWRT—if uncertain, always type in the command to see the usage prompt). The computer has been specifically instrumented to emulate both your home computer connected to a home router (and then to the Internet) and a server

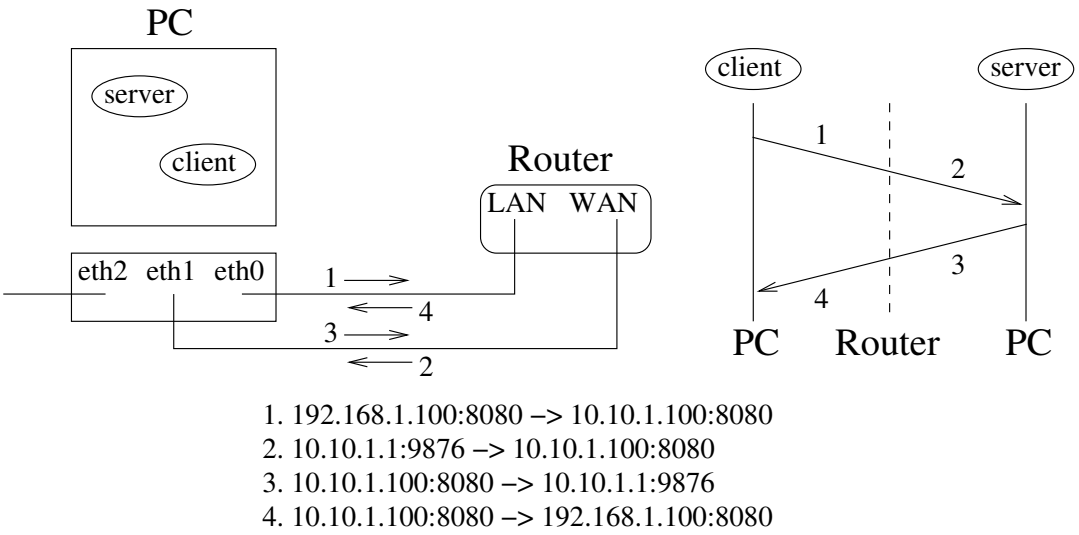

Figure 1: Network configuration in the lab.

computer running on the Internet at the same time. The packets between your "home computer" and the "Internet server" are exchanged through the router, just as what is happening when you access the Internet from home everyday. If you are interested in the motivation and design of the lab, please refer to [1]. Let the course instructor know if you have any suggestions on the lab setup.

The computer thus has three IP addresses: `142.104.74.xx` with name `n-`*greek* on `eth2` to access the UVicNet, `192.168.1.100` with name `pc-lan` on `eth0` to emulate your home computer, and `10.10.1.100` with name `pc-wan` on `eth1` to emulate an Internet server. The router can be reached with `ssh` from the computer at `192.168.1.1` or with name `rt-lan`. The lab instructor will provide more details in the lab on how to access the router, including username and password.

In order to let a packet sent from your "home computer" (`192.168.1.100`) to the "Internet server" (`10.10.1.100`) go through the router (`192.168.1.1`), when you send the packet, you need to specify its source IP address to be `192.168.1.100` (this can be achieved by using the `bind()` system call or the `-s` option in `netcat` or `nc` utility) and destination IP address to be `10.10.1.100`. The router has a network address translation (NAT) module, which will forward the packet to the "Internet server" with source IP address `10.10.1.1` and destination IP address `10.10.1.100`. Once the packet arrives at the "Internet server" and gets responded, the reply packet will have source IP address `10.10.1.100` and destination IP address `10.10.1.1`, so the packet is going to the router. Due to the NAT module on the router, the packet will be forwarded by the router to your "home computer" with source IP address `10.10.1.100` and destination IP address `192.168.1.100`. The same process happens for the follow-on packets exchanged between your "home computer" and the "Internet server", as shown in Figure 1. For more details, please refer to [1].

3.2 socket API

`socket` is the 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

- `recvfrom()` or `recv()`: to read from the socket
- `sendto()` or `send()`: to write to the socket
- `close()` or `shutdown()`: 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 [2].

3.3 HTTP Protocol

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

3.3.1 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 simple 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.

3.3.2 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: Fri, 12 Sep ...`	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 permissions, 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].

4 Requirement

4.1 Basic Features

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

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 simple web server to wait at the UDP `port` for incoming requests 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 successfully, for example, the simple web server will print out the following message if being invoked as `./sws 8080 /tmp/www`

```
sws is running on UDP 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/1.0 404 Not Found` by the server.

4.1.2 Server Operations

When running, the simple web server should respond to incoming requests at UDP `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 `Sep 12`, successfully served a request `GET / HTTP/1.0` from the client at port `4096` on host `192.168.1.100`, it should print out

```
Sep 12 12:00:00 192.168.1.100:4096 GET / HTTP/1.0; HTTP/1.0 200 OK; /tmp/www/index.html
```

or

```
Sep 12 12:00:00 10.10.1.1:5678 GET / HTTP/1.0; HTTP/1.0 200 OK; /tmp/www/index.html
```

if the request has gone through the NAT running on the router (note that the source IP address and port number has been changed from the viewpoint of the server), and `filename` indicates the location of the HTTP object returned, if the request is successfully served. The log messages should be ordered chronically.

4.1.3 Terminating the Server

The simple web server should continue serving HTTP requests until the `q` key is pressed in the terminal running the server. In order to respond to console input, you may need to use `select()` system call. When the `q` key is pressed, the server should finish serving all ongoing 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. To avoid this problem, you should 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 simple 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),
- or implement the simple web server with multiple processes (using `fork()` system call) or threads (using `pthread` library [4]),
- or make the server portable on Intel x86-based PC running Linux and Broadcom MIPS-based Linksys WRT54GL running OpenWRT with cross-compilation.

In addition, students are encouraged to propose their own bonus features for approval. However, no extra feature should affect the format of server log messages that are used for evaluation purposes.

If you want to design and implement a bonus feature, you should contact the course instructor by email for permission at least **two** week before the code due date and clearly indicate the feature in code submission. Once accepted, the credit for **correctly implemented** bonus features will not exceed 20% of the full marks for this programming assignments.

5 Self-Testing

You can test your simple web server with any HTTP/1.0-compliant client, even `netcat` (or `nc`). Using `nc`, you can connect to your web server running at `port` on `host` and send the request by

```
echo -n "GET / HTTP/1.0\r\n\r\n" | nc -u -s <source_ip> <host> <port>
```

For example, `nc -u -s 192.168.1.100 10.10.1.100 8080` to let the request go through the router, or `nc -u 10.10.1.100 8080` bypass by the router. The `echo` commend outputs the intended request with the right format, and redirects as the input for `nc`.

And the HTTP response should follow after a correct HTTP request is successfully processed.

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 `runme1st.sh` shell script is included in `www.tar.gz` to be executed first to set file permissions properly for testing purposes. A sample request sequence `http.request` is also included in `www.tar.gz`, including the expected response line for your reference.

However, you also need to create your own test cases (at least five test http requests with the associated test directory and expected responses), as your simple web server will be tested against different document directories and request sequences during the code demo and evaluation.

IMPORTANT: You will also be provided some sample code involving socket and client/server programming in the lab tutorials. However, you are required to design and implement your code on your own, as well as the test cases. Copy&paste the code/cases from someone or somewhere without reference and credit, including the lab samples, is not allowed.

6 Submission

The entire programming assignment, including the code and documentation and your own test cases, should be submitted electrically through connex at `http://connex.csc.uvic.ca/` on or before the due date. The site will start to accept submissions one week before the due date.

Only the source code (including header files and Makefile), documentation (Readme) and test cases should be included in a single `tar.gz` 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 directory
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 (the lab instructor will provide a sample `Makefile`)

```
make
```

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

```
make clean
```

All assignments will be tested and evaluated on `n-`*greek*`.csc.uvic.ca` in ECS360. Please make sure to develop and test your code on the lab platform.

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 the allowed bonus features, if any. You need to include your own test cases in `p1.tar.gz` too.

The code itself should be sufficiently self-documented, which will not only help you build and maintain your code well but also assist the code demo and evaluation.

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].

7 Marking

This assignment is worth 20% in the final grade (breakdown will be provided in lab). Demo will be arranged after the submission, where the marker will ask questions on the code design and implementation. For mark posting and appeal policies, please see the official course outline at [5].

References

[1] `http://abraham.cs.uml.edu/~heines/academic/papers/2010sigcse/CD/docs/p133.pdf`

[2] `http://beej.us/guide/bgnet/`

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

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

[5] `http://courses.seng.engr.uvic.ca/courses/2013/fall/csc/361`