

CSc 361: Computer Communications and Networks (Spring 2010)

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

Spec Out: Jan 18, 2010
Code Due: Jan 29, 2010

1 Introduction

In this programming assignment, you will implement a simple web server (SWS) in C or C++, following a simplified HTTP protocol and using the stream (STREAM) socket application programming interface (API) with the Transmission Control Protocol (TCP). 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 four tutorial modules arranged in the weekly lab sessions during the course of this assignment.

Date	Tutorial Module
Jan 20	P1 Spec go-through, intro to socket programming
Jan 27	HTTP analysis and last-minute help
Feb 3	mandatory demo of P1

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-alpha.csc.uvic.ca`), and also to a Linksys WRT54GL router through two Ethernet cables. The computer is running Linux (Scientific Linux), so is the router (OpenWRT). 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 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.

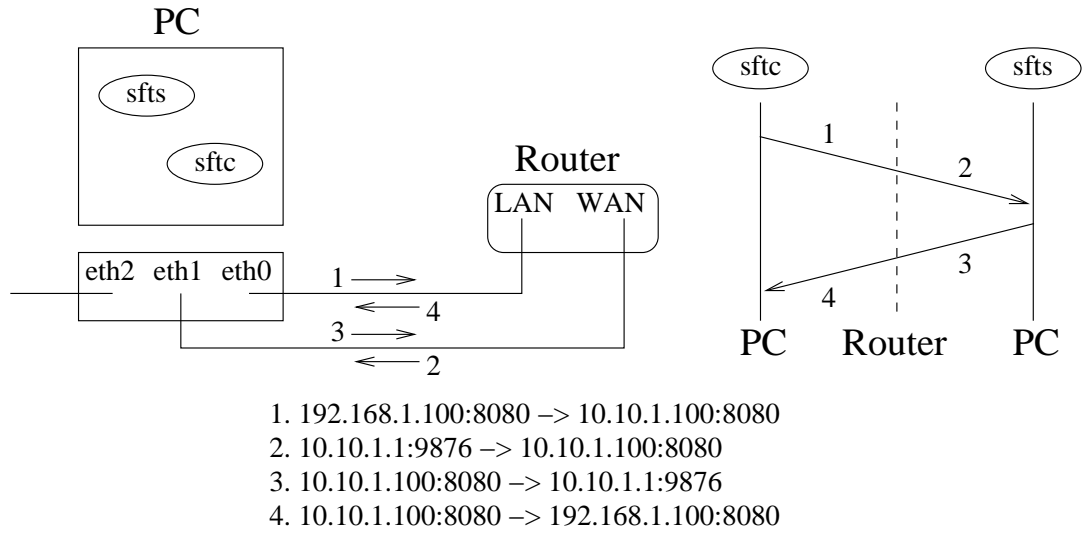


Figure 1: Network configuration in the lab.

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**.

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.

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
- **listen()**: to wait for an incoming connection request
- **accept()**: to accept the incoming connection request with a new socket
- **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 [1].

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.” [2]

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.” [2]

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 (required)
Server: Apache/2. ...	response header (omitted)
X-Powered-By: PHP/4 ...	response header (omitted)
Connection: c ...	response header (omitted)
Content-Type: text/ ...	response header (required)
	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**. It also returns **Date** and **Content-Type**, but will omit other 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
403	Forbidden	bad file permission
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 doesn't have permission to read the requested file, it will return HTTP/1.0 403 Forbidden followed by a blank line. If the simple web server can understand the request, but the requested object is not available (e.g., nonexistence), 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 [2].

4 Requirement

4.1 Basic Features

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

109 4.1.1 Invoking the Server

110 The syntax to run the simple web server is

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

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

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

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

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

122 4.1.2 Server Operations

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

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

126 `Seq no. YYYY MMM DD HH:MM:SS Client-IP:Client-Port request-line; response-line;`
127 `[filename]`

128 The **Seq no.** is a simple sequence number which identifies every log line. For example, if the
129 server, on the noon of **Sep 12**, successfully served a request `GET / HTTP/1.0` from the client at port
130 `4096` on host `192.168.1.100`, it should print out

131 `Seq no. 1 2009 Sep 12 12:00:00 192.168.1.100:4096 GET / HTTP/1.0; HTTP/1.0 200 OK;`
132 `/tmp/www/index.html`

133 `OR`

134 `Seq no. 1 2009 Sep 12 12:00:00 10.10.1.1:5678 GET / HTTP/1.0; HTTP/1.0 200 OK;`
135 `/tmp/www/index.html`

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

140 4.1.3 Terminating the Server

141 The simple web server should continue serving HTTP requests until the **q** key is pressed in the
142 terminal running the server. In order to respond to console input, you may need to use `select()`
143 system call. When the **q** key is pressed, the server should finish serving all ongoing requests, close
144 all created sockets, and release all allocated resources before exiting gracefully.

145 After terminating the server, or if the server is aborted, some socket resources may still be used
146 by the system. If you rerun the server on the same port immediately (or if there is already a program
147 on the same port), normally you will get a `bind()` error. To avoid this problem, you should use
148 `setsockopt()` with level `SOL_SOCKET` and option `REUSEADDR` before `bind()` to allow port reuse.

149 4.2 Bonus Features

150 Only a very simplified version of HTTP/1.0 is to be supported by the simple web server. However,
151 students have the option to extend their design and implementation to

- 152 • include more features in the HTTP protocol (e.g., persistent HTTP connection, HTTP request
153 pipelining, etc),
- 154 • or implement the simple web server with multiple processes (using `fork()` system call) or
155 threads (using `pthread` library [3]),
- 156 • or make the server portable on Intel x86-based PC running Linux and Broadcom MIPS-based
157 Linksys WRT54GL running OpenWRT with cross-compilation.

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

160 If you want to design and implement a bonus feature, you should contact the course instructor
161 for permission at least **two** week before the code due date and clearly indicate the feature in code
162 submission. The credit for **correctly implemented** bonus features will not exceed 25% of the full
163 marks for this programming assignments.

164 5 Self-Testing

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

```
167 nc -s <source_ip> <host> <port>
```

168 to send your HTTP request to your implemented web server. For example, `echo -e -n 'GET /`
169 `HTTP/1.0\r\n\r\n' | nc -s 192.168.1.100 10.10.1.100 8080` to let the request to go through
170 the router, or `echo -e -n 'GET / HTTP/1.0\r\n\r\n' | nc 10.10.1.100 8080` to bypass the
171 router.

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

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

```
175 mv www.tar.gz /tmp  
176 cd /tmp  
177 tar -zxvf www.tar.gz
```

178 A sample request sequence `http.request` is also included in `www.tar.gz`
179 However, your simple web server may be tested against different document directories and
180 request sequences during code evaluation.

181 6 Submission

182 The entire programming assignment, including the code and documentation, should be submitted
183 electronically through Connex at [4] on or before the due date.

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

```
187 cd direcoty  
188 tar -zcvf pl.tar.gz .
```

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

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

```
193 make
```

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

```
195 make clean
```

196 All assignments will be tested on `n-delta.csc.uvic.ca`

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

200 The code itself should be sufficiently self-documented. For more information on acceptable
201 coding style, please see the `code-style.pdf` file in Resources section of our Lab web site on connex.

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

205 7 Marking

206 This lab project is worth 10% in the final grade of this course.

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

208 References

209 [1] <http://beej.us/guide/bgnet/>

210 [2] <http://www.w3.org/Protocols/rfc1945/rfc1945>

- 211 [3] <http://www.llnl.gov/computing/tutorials/threads/>
- 212 [4] <http://www.csc.uvic.ca/labs/csc3611>
- 213 [5] <http://courses.seng.engr.uvic.ca/courses/2010/spring/csc/361>