COM SCI 118 Computer Network Fundamentals

Project 1: Concurrent Web Server using BSD Sockets

Guillaume Lam: 304253718/guillaum
Jasmine Mok: 604283974/classjmo

Project 1 Report

Server Flow & Structure

The structure of the web server is based off of the example client-server code provided along with the CS118 Ubuntu image. At the highest level, the flow of the code is as follows:

1. Server is created via BSD socket() function and bound to user specified port and IP address.
2. After the server begins listen(), it undergoes an infinite loop waiting for clients to connect. The server currently accepts at most 10 concurrent connections.
3. Should a client attempt to connect to the server, accept() is called and a new process is forked for the client in order to accommodate for concurrent request handling. The client and server now have established a TCP connection with each other.
4. Client connections are passed to the handle() function via their assigned socket descriptor, where the HTTP request from the client is parsed for the HTTP method and file requested.
5. It should be noted at this point that there are several error checks in place to ensure HTTP request correctness, the details of which can be seen by inspecting the code's comments
6. Having parsed the HTTP request, handle() now tries to open the requested file, returning a 404 error code upon failure.
7. Having parsed the HTTP request, handle() now begins to process any valid requests by constructing the HTTP response. The header is constructed and sent to the client, along with small packets containing the requested file's data.
8. Upon finishing transmission of the requested file, the current client-server TCP connection closes, and the process supporting it dies.

Challenges

The primary challenge in this project was segmenting large amounts of file data into smaller packets due to an apparent packet size limit for HTTP responses while testing the server. This problem primarily occurred when transmitting jpeg and gif files. The first method tried was to simply read in a set amount of file data per packet, but complications arose during execution. For some reason, the buffer used to read in file data would not get completely filled much of the time. The alternative method employed was to simply read in file data character-by-character into a small data buffer (256 bytes) before attaching it to a packet to be sent out. This is the current manner in which file requests are handled by the server. Other than this, there were no demonstrably difficult components to this project; discussion sections were very helpful in providing guidance.

Setting Up and Testing the Server

Please follow these steps to compile and execute the server:

1. In the directory containing server.c and the Makefile, execute *make* to produce an executable called *server* that can be run to set up the server.
2. Run the *server* executable and provided a port number as well. It is recommended to use 4001, 4002, or 4003 as those ports were used in developing the server code.
3. Open any web browser from here, the server may be accessed at *localhost* or 127.0.0.1. HTML, jpeg, and gif files have been provided for testing purposes. These files are located in the same directory as the *server* executable.
4. To request *cat.jpg* from the server, navigate the web browser to *localhost:xxxx/cat.jpg* or *127.0.0.1:xxxx/cat.jpg* where *xxxx* is the port number specified when running the *server* executable.
5. Repeat step 4 to test HTML and gif requests.

Sample Outputs

GET /cat.gif HTTP/1.1
Host: localhost:4003
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/*;q=0.8*
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,;q=0.7*
Connection: keep-alive

The above text is an example of an HTTP request that will be outputted to the console upon the establishment of every client-server connection.

No port number provided for server.
Unable to create server socket at port 4000.

The server code has safeguards in place in the event of missing parameters or failed setup. The above messages are some examples of what you might see in such an event.

Client request was unable to be processed.

Requested file could not be served.

Sample output has been provided for client-side errors. This can range from accessing a non-existent file or the HTTP request containing an unsupported method.

```
HTTP /1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>
        <body>
                <h1>Meow the Jewels</h1>
        </body>
</html>
```

An example of the HTTP response sent to the client after receiving and processing a valid HTTP request is shown. This output is provided simply to allow the grader to see what is transmitted to the client. The HTTP response is never outputted to the console or to the client.