CS118 Winter 2017
Computer Network Fundamentals
Project 1: Web Server Implementation Using BSD Sockets

Zhi Li (504341757) and Thomas Chang (004632787)

Table of Contents

SERVER DESIGN

Overview

Our code implements a web server program conforming to HTTP protocol using C++ and BSD sockets. There are two modes for our program, Per Client Process Mode and Constrained Multitasking Tasking Mode. The first one accepts one argument <port number>, and the other one accepts two argument <port number> <max number of process>. Both of them attempt to optimize the server using concurrent programming, but adopt different approaches, which will be explained in more detail later. Once the program runs, the client will be able to establish connection with the server using specified port number. Specifically, the client can send HTTP request to the ask for files and the server will first output the HTTP request to the terminal and then respond accordingly by sending HTTP response as well as the desired files. Here is a list of details about what our server program supports.

1. HTTP version: HTTP/1.1
2. Request method: GET
3. Request url: can decode space space in the url
4. Request headers can be recognized by the reponse: Host, Connection, User-agent, Accept-language
5. Response status: 200, 400, 404, 505
6. Response headers will be generated if applicable: Connection, Date, Server, Last-modified, Content-Length, Content-Type
7. Content-Type supported: text/html img/gif img/jpeg img/x-icon (for favico)
8. If the client does not specify any path, e.g: localhost:<port numer>, it has the same effect as: localhost:<port numer>/index.html

Structures and Classes

After examining HTTP request and response closely, we decided to use object oriented programming to handle the client's request. To utilize the similarity between the request and response, we first designed HTTPMessage class to handle the common structural elements of both types of messages. And then we designed HTTPResponseMessage and HTTPRequestMessage classes, which are derived from HTTPMessage, to handle specific functionalities of request and response respectively.

In high level, the server receives data from the client as a buffer of bytes, then we construct HTTPRequestMessage by parsing the raw bytes into its HTTP message elements, from which we construct the HTTPResponseMessage. The basic design of our webserver is shown below:

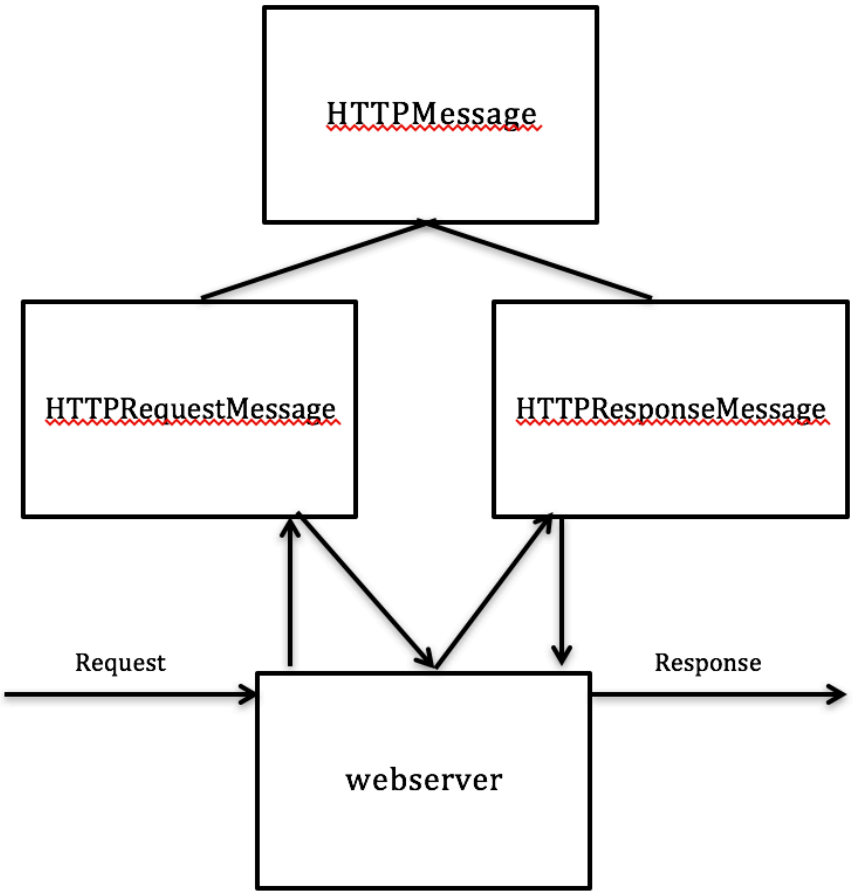

Figure 1High level Structural Design

Concurrent Optimization

At first, although an iterative server is probably enough for a small number of people, we decided to optimize the server using concurrent programming. Here, we adopted two approaches, Per Client Process and Constrained Multitasking Tasking. For Per Client Process, we spawn a new process for each client. However, each time we generate a new process, we have the overhead for copying and context switching. Eventually, we may reach a point that adding an addition process actually decreases overall performance. Therefore, we also implement the constrained-multitasking server, which limits the number of processes created by the server. For this option, the user needs to supply both port number and the maximum number of process he or she wants. Then the program will spawn corresponding number of processes. However, although this approach eliminates the overhead of creating new process each time, the downside is that the user does not have an easy way to shut down the program. We admit it does bring

inconvenience in the testing, but the justification is that server is supposed to be always on!

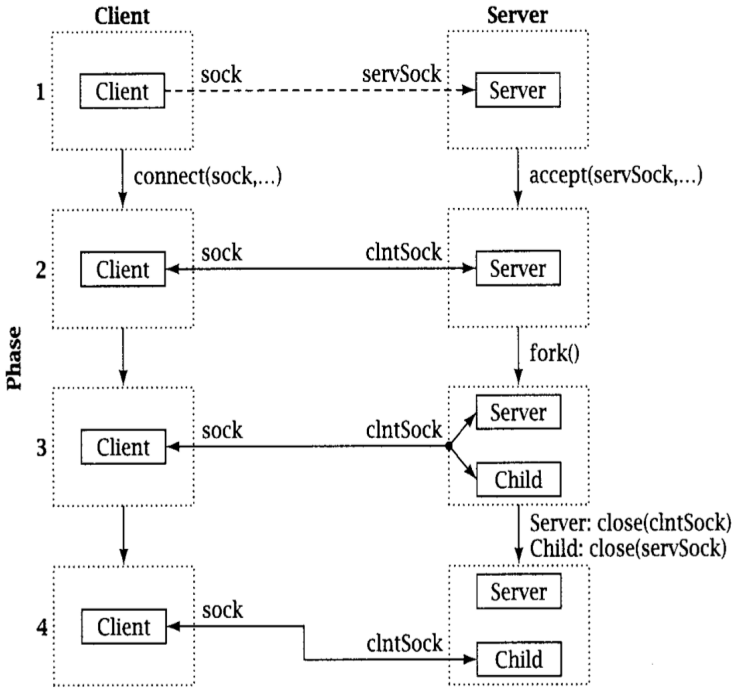

Figure 2 Per Client Process High Level Logic (Donahoo & Calvert, 2001)

DIFFICULTIES

The first difficulty we encountered was to design our server program to be reliable and adaptable. As result, we decided to use OOP described above. We believe this design makes the code more structural and can be easily expanded if we also need to develop for the client side.

Secondly, we had a hard time deciding the buffer size. We could write a while loop to read more bytes the client sent, but we felt like this might introduce a security program by giving this control to the client. Therefore, we finally chose to use 8192 bytes as the buffer size because this should be big enough for normal HTTP request and response. The program also generates warning when it used up all 8192 bytes.

Third, we had difficulty implementing persistent connection, which is the default of HTTP 1.1. We thought that for persistent connection, the server should not close the socket about it responds. However, that means if the client did not specific “Connection: close”, the server’s socket would never close! Therefore, instead of relying on client, we think we should close the connection after some amount of time, such as 10 seconds. we leave that part for future improvement.

Our last notable difficulty was managing work between partners. At one point, we both simultaneously developed code that accomplished very similar tasks, but poor communication made us have to discard some work in the end.

COMPILATION AND RUN INSTRUCTIONS

Instruction for the usage:

1. Put your website files in the "website" directory
2. Change directory to "*src*" directory
3. Type **make**
4. Type **./webserver** + <port_number> + (optional: <max number of process>)
   (At this point, the webserver is on. You can open the browser to connect to the server. For exmpale, type "localhost:<port_number>/index.html")
5. Type **ctrl + c** to stop the webserver program
6. Type **make clean** to clean up

Note: if using constrained multitasking mode, once the program runs, the processes will fork and you will have no control over it. The easiest way to shut down the program at that point is to restart the computer.

EXPLANATION OF SAMPLE OUTPUTS

The webserver outputs all requests it receives to the console. For example, connecting to the server through a Firefox web browser gives the following console output:

Figure 3 Screenshot for Part A

This is a sample request sent from a Linux machine to connect through the server at port 8081. The first line specifies the HTTP method and the HTTP version. In this specific example, the client is requesting the get the default webpage and is using HTTP 1.1. The host field refers to the webserver, and the user-agent field gives information about the client.

I'm cool.

I'm cool.

I'm cool.

Just want to test

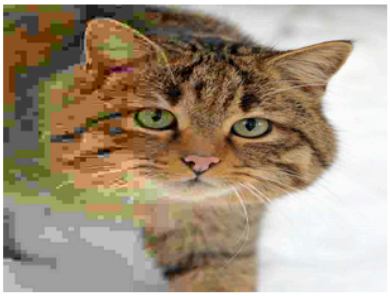

Figure 4 Screenshot for Part B

Upon receiving a valid request, the server will send a response in HTTP format to the client. If the request is for an html file, the result will show in the client web browser. The default webpage is index.html, which references several image files. The server will also send these as well when requested by the client. If the server is connected to through the web browser, the default webpage will be the following:

Bibliography

Donahoo, M. J., & Calvert, K. L. (2001). *TCP/IP Sockets In C, Practical Guide for Programmers.* Morgan Kaufmann Published.