CS 118

Project 1: Concurrent Web Server with BSD Sockets

Report

Student 1: Haonan Zhou

UID: 104180068

SEASNet Login: haonan

Student 2: Yichen Pan

UID: 604152505

SEASNet Login: yichen

Server Design

On a high level, the server first creates a socket for itself. Then, the socket descriptor was bound with information about the server, such as server address, and port number, which is accepted as its only argument when executing. After that, the server goes on an infinite loop, where it listens to any client that sends a request. When a request is received, the server forks a child process, in which a function is invoked to handle all communication between the server and the client. After the communication is finished, the server continues to listen to any client message and repeat the process described above.

Design of each function is described below, in more details:

+ int main:

- The main function first opens a socket with the C library function socket(). It also creates a "sockaddr_in" struct, which holds information such as the server address and port number. The main function then use the C library function bind() to associate the information in the "sockaddr_in" struct with the socket descriptor created earlier. Then the listen() function is called, which marks the server socket as passive so that it can accept any connection requests from a client.
- After that, the server enters an infinite loop, in which it waits for the client to send a request. If a request is received through the accept() function, a process is forked. The parent process simply closes the accepted socket. The child process, on the other hand, calls a function dostuff(), where all the server-client connection is handled. After this function is returned, the server will keep listening for another client request.

+ dostuff()

- Input:
    - int sock: socket descriptor of the client socket
- Output: None
- This function handles all communications between the server and a client. A buffer of length 1024 bytes is created to store data before transmission. The server first reads the client request into the buffer. Then the buffer is passed into the parse_msg() function to parse the information that is required to include in the response message. The parsed information is stored in an "http_msg_t" struct. This struct is passed into the format_response() function to organize the information into a properly formatted HTTP message. This response message is then written to the client socket.
- The next step is to send data to client. The file required by the client is included in the http_msg_t struct. The server tries to open this file. If succeeds, the first 1024 bytes are read into the buffer and then written to the client socket. The server

continues this process until there is no more data to read in the file. Then the function would exit.

+ parse_msg()

- Input:
  - char* msg: a string of HTTP message from the client
- Output: a pointer to an http_msg_t struct, which contains all information that the server needed to construct a response message
- This function parses information from a client HTTP message. Since the message is known to be formatted in a certain way, the function can call the parse_field() function multiple times to get a specific demanded field. Two of the fields, however, needs to be parsed carefully. First, a dot needs to be added to the beginning of the file path so that the server knows to find the file in the current folder. Second, the connection field is not located in the beginning of the message. Therefore, function strstr() is called to locate the substring in the message that starts with "Connection:". After that, the parser can copy the information following the substring into the struct.

+ parse_field()

- Input:
  - char* msg: a string of HTTP message
  - char* field: the first field is copy into this string
- Output: a char pointer that points to the char immediately following the parsed field
- This function scan through an HTTP message and copies the first field into a string. Any space, newline, or carriage return characters at the beginning of the message are skipped. The function copies the first field in the message until it scans another space, newline, carriage return, or EOF.

+ format_response()

- Input:
  - http_msg_t* msg: a pointer to an http_msg_t struct, which contains all information needed to construct the response message.
- Output: a string of server HTTP response message
- This function first determines the validity of the client request. For example, our simplified server should reject any request that does not have a "GET" method, although this is not what a real server should do. In another scenario, the server should send out "404 Not Found" in the header when a client may request a file that does not exist. But if the client sends a valid request, the server would send "200 OK."
- After that, the sprintf() function is called multiple times to format the HTTP response message.

+ content_type()

- Input:
  - char* filename: a string that contains the requested file name
- Output: a string that contains MIME type of the file
- This subtle function checks the extension of the file and return the correct MIME type. It only recognizes three extensions: JPEG, GIF, and HTML.

Difficulties:

- In Part A, the response message did not show correctly at first.
  - It turns out the problem is that there is a bug in the parse_field() function. At first, our code only finishes copying a field from the HTTP message whenever it sees a space, newline, or EOF. However, the client message that is sent out by the browser has both a carriage return and a newline character at end of each line. As a result, carriage return is included in the parsed field. If the server later writes this field to client, it will reset the cursor whenever a carriage return is encountered. Then each line will be overwritten by the following line. It is easy to solve this problem: simply stop copying character in to field when a carriage return is detected.
- In Part B, an image could be displayed in the browser.
  - When the server writes the response message to the client, it calls write() with maximum buffer size. I solved this by changing that argument to the actual length of the response. And the browser can now display an image.

A Brief Manual:

- To compile:
  - 'make' command
  - The code will be compiled into an executable file called "webserver"
  - Any warning would be suppressed during compilation
- To run server:
  - './webserver <port number>' command
  - For example: './webserver 1118' will run the server on port number 1118

Sample Output:

- Part A:

- We run the server with command ‘./webserver 6300’, and access address ‘localhost:6300/blah.html’. The following client message is printed on the server console:

```
The message from client:
GET /blah.html HTTP/1.1
Host: localhost:6312
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
Cache-Control: max-age=0
```

- This message is sent to the server by the client (browser). The headers that would be useful in the response message are:
  - method: GET
  - HTTP version: HTTP/1.1
  - requested file: blah.html
  - Connection: keep-alive

- Part B:
  - During testing and debugging, we also printed the response message in the console:

```
HTTP/1.1, 200 OK
Connection: keep-alive
Date: Sun May  3 12:01:18 2015
Server: My WebServer
Last-Modified: Sat May  2 09:57:34 2015
Content-Length: 62
Content-Type: text/html
```

  - The response is output of the format_response() function.