HTTP Web Server

George Taylor - 1522968

***Abstract*—This is the documentation section for the HTTP Web Server assignment as part of the Networks module. It aims to give a high level overview of the structure of the project and give some indication as to it's scope.**

I. Compilation and Execution

The web server can be compiled using by running the make file found in the source directory. To begin the server, run the `./server` file providing a port number and root directory to serve files from. The `-d` flag can be optionally added to run the server as a daemon process. For example:

```
# ./server -d 8080 /home
```

will run the server as a daemon process on port 8080 using the `/home` directory as root.

II. Scope

From the start of the assignment I intentionally designed the server with scalability in mind, using RFC 2616 as a guide. This is evident through the inclusion of enums for different request types, unused parameters in the request struct, and abstract functions for parsing requests.

III. System Flow

When the process is initiated, the first check is for the parameters of the server. The inclusion of the `-d` flag starts the process as a daemon, this was included in order to facilitate ease of testing as I wanted to be able to print to the console for debugging purposes. The server also requires the user to specify a port number and a directory to act as the route of the server (equivalent to the Apache `/var/www/` directory). If the `-d` flag has been set the process will fork a child, then the parent process will terminate and the child process will disconnect itself from the standard IO streams.

The new server process will then set up it's connection and bind itself to the specified port to start listening for any incoming connections. When it receives a connection the server will fork; the new child process will handle the client requests then terminate, and the parent process will continue to listen for incoming connections.

A. Serving a request

The client handler receives the request by dynamically allocating space on the heap until the entire request has been read. It will now allocate space for the request structure which is initialised by a method that parses the raw request. If the parser does not recognise the request method then a 501 response will be given.

Once the request has been parsed it will be passed along with the client structure to a method that serves an appropriate response. The method first determines what resource is trying to be accessed using the root directory and the request URI. It will then determine whether the resource is a file or a directory (serving a 404 response if it is neither).

The server successfully implements persistent connections, limiting the overhead required to set up and close down TCP connections. The connections are persistent for either 10 requests or a 30 second timeout (configured by `CONNECTION_REQUEST_LIMIT` and `CONNECTION_TIMEOUT` respectively). The server will check for a `keep-alive` value in the `Connection` field of the header and hold connections open accordingly.

Whenever a response is served to the client, the relevant method will first check the method of the HTTP request. If the request is a `HEAD` request then the server will withhold the content from the response.

A variety of headers are attached to the responses such as 'Server', 'Date', 'Allowed' and 'Content-Language'. The 'Connection' header will be added indicating whether the connection should be held open or closed.

B. Serving a Directory

If the requested resource is a directory then the server is designed to produce a directory listing page allowing the client to browse the file system. The server will generate a directory listing by iterating over all of the objects in the directory adding a html link to a response buffer that is then returned to the user. During this several checks take place to determine whether objects in the directory are files or other directories. If the directory cannot be accessed then a 404 response will be served and if the user does not have permission to access the directory then a 403 response will be served. If any other error occurs while attempting to read the directory then a 500 response will be given.

C. Serving a File

To serve a file the server will first send the "200 OK" response. The server then determines the length and content type of the file, sending these as response headers. The server will then open the file in read-only mode and allocate a buffer of a fixed size to be used in streaming the file. The file will then be read into the buffer in chunks which are sent to the client over the socket before being replaced by the next chunk. There is a significant amount of checking taking pace when sending the file in order to ensure that the entire file is sent correctly without dropping any chunks. The reason for splitting the data into packets in this way is because it is not uncommon to find files that are measured in gigabytes, it would therefore be extremely inefficient, and potentially impossible, to load the entire file into memory before sending it.

D. Closing down the connection

When the server has either timed out or reached the limit of responses for a connection it will close the connection to the client, free the memory used (including removing the request object) and terminate. At this stage the entire client process that was created when the client connected will terminate.

At any stage in the execution, if the program (including all forked processes) receives the `SIGINT` signal two flags will be set, one for the parent server and one for a client server, that allow the program to terminate cleanly and manage it's own memory. This is done by setting the number of remaining requests to 0 in the case of serving a client.

E. Resetting the TCP connection

Throughout the development of the server I experienced a problem whereby if I closed the server I would then not be able to bind to the same socket when it started back up. Using `netstat` I found that when this occurred the port was in the `TIME_WAIT` state. `TIME_WAIT` is a feature of the TCP protocol that acts as a timeout on the orderly shutdown of the socket. To resole this my options were:

- Reduce the system wide timeout to eliminate the wait. This would have been a poor solution as it is system specific and I did not want to interfere with the established TCP protocol.
- Send a TCP `RST` flag rather than the `FIN` flag sent when the server normally closes the socket. The purpose of this solution is to pass the responsibility for closing the socket onto the client thus avoiding the `TIME_WAIT` state on the server side. However I decided that passing issues to the client rather than handling them on the server was an inappropriate solution.
- Accept the issue. In the end I decided that this was an issue that should be accepted by the server. The `TIME_WAIT` flag is implemented as part of the TCP protocol and so I didn't want to mess with it. Additionally, I was producing the issue regularly as a result of constantly starting and stopping the server whilst testing. This is not an accurate usage for a HTTP server which would ordinarily be started and left to run uninterrupted.

IV. What would I do differently next time

Unfortunately due to an extremely busy month for me there was a lot that was lacking in the system that I had hoped to achieve (as can be seen by the frameworks put in place for various features). With the way I have implemented the file streaming it would have been a relatively simple task to extend this to accept byte ranges. Perhaps my biggest concern with the current implementation is the lack of appropriate security. There is currently nothing in place to prevent the client from traversing the file system freely (even using `../` to access parent directories). This is a major flaw that would have been easily solvable with more time.

With the well designed architecture of the system it should be a relatively simple task to extend the web server to cover a larger subset of RFC 2616. This well designed architecture demonstrated it's worth when converting the system from redirecting to an `index.html` for directory listings, to serving a directory listing straight from the directory. This particular change only required me to remove a condition in a conditional statement.

V. Conclusion

Overall I am pleased with what I achieved. With a well designed architecture the system is extremely scalable and with extra time I am confident that I would have been able to implement some of the more advanced features.