Project 1 Report

Contributors

Young Hoon Kang, 904469956
Parts of the HTTP request/response
Web server

Michael Wang, 204458659
Parts of the HTTP request/response
Web client, abstractions in WebUtil/FileRequest/FileResponse

High-level Description

In this project, we implemented a simple web client and server (two versions, synchronous and asynchronous) that could transfer files from the server directory to the client using the HTTP protocol. The client/server has support for request timeouts as well as HTTP/1.1 (persistent connections and pipelining). The client is called “web-client”, the synchronous web server is called “web-server”, and the asynchronous web server is called “web-server-async”. To implement this, we used a series of system calls to set up and create a network connection for both the server and the client.

For the synchronous server, we first set up and create a socket. Then, we bind the socket to an address (given as an argument) and set it to listen to incoming connections. When it detects an incoming connection, the server accepts the incoming connection, creating a new socket. It then spawns a new thread to handle the connection, while the main thread continues to listen for incoming connections. The worker threads handle connections with support for HTTP/1.1, i.e., persistent connections and pipelining. Each thread runs an infinite loop that waits for a request from a client, with a timeout of 10 seconds. If there is no request within 10 seconds or the socket is closed from the client’s side, it breaks out of the loop and the worker thread terminates. If there is a request, the server responds to it appropriately (send a file if it exists, otherwise send 404; if the request is malformed, send a 400). After each request, the server checks to see if the request was persistent, i.e., whether the request used HTTP/1.1 and did not have a “Connection: close” header. If it was nonpersistent, it breaks out of the loop; otherwise, it continues to loop and wait for requests.

Similarly for the asynchronous server, we first set up and create a socket. Then, we bind the socket to an address and set it to listen to incoming connections. The server maintains a file descriptor set, readset, that keeps track of sockets open for reading, which includes the listening socket. It also maintains a hashmap that maps open each open connection (socket) to the time it was last accessed. The server then runs a loop continuously. In the loop, the server checks whether there are any incoming connections or incoming requests. If there is an incoming connection, the server accepts the connection and adds the newly created socket to the readset as well as the hashmap (with the current time). If there is an incoming request, the server responds to it the same way as the synchronous server (and closes the connection if the request is nonpersistent). At the end of the loop, it checks whether any connections have timed out, and closes those connections and removes them from the readset. That is, it checks whether the time the socket was last accessed is within the receive timeout, and if it is not, the connection is timed out.

The web client takes multiple URL(s) as arguments, which the client tries to download. To provide support for HTTP/1.1, the client first organizes the URLs by mapping each (host, port) combination to a list of files the client wants to retrieve from host:port. The client then tries to establish a connection to each (host, port) by creating a socket and connecting to the server. The client sends an HTTP request to the server for the first (of possibly many) file using HTTP/1.1 as the version. The response is saved if the status is 200 OK and it has the "Content-length" header. If the response uses HTTP/1.1 and does not include a "Connection: close" header, the client then pipelines requests for the rest of the files to the server. Otherwise, the client tries again to establish a connection to the server and send another request using the same method. This continues until requests have been sent for all files.

Obstacles

After we implemented the web server and client, we ran into some problems. One of the first problems we ran into was when we attempted to send large files. For files of size of around 100 megabytes, we saw the send() function only sent a part of the file before terminating. This was because we had set the timeout value to be 3 seconds, which was too short for files of this size. To fix this, we set the timeout value to 10 seconds, which was enough to send files up to 1 gigabyte. In addition, to make sure that sending large files did not overload send(), we set an upper limit of 1 MB. Another problem that we encountered was when receiving requests. When the server received a request, it would not wait for the request to finish, and would instead reject the first part of the request. Then, the server will read the next part of the request and reject it as well, continuing to reject parts of the request individually until reaching the end. To fix this bug, we made sure that the server would read until the end of the request by reading until CRLF and an empty line after that. Similarly, we made sure that the client would read the response until the end by reading until CRLF and then an empty line.

Testing

To test our code, we first built our server and client using the command make in the provided ubuntu distribution with vagrant. Then, we tested the default arguments provided in the spec, and then tested non-concurrent non-persistent well-formed requests. We also checked that the web server and client worked for hosts other than localhost, and for other port numbers. Then, we checked for malformed requests, including requests for directories and files that did not exist. We checked for concurrent connections by opening up many sessions of vagrant and sending requests for large files at once. We checked our implementation of HTTP/1.1 by testing persistent requests from client to server, including the use of the “Connection: close” header; we gave multiple URLs to the client and checked whether the requests were pipelined properly. We also checked both the client and server’s handling of HTTP/1.0 requests; if the server/client received a HTTP/1.0 request/response, the server would respond with a HTTP/1.0 response and the client would close the connection after receiving the response.