Basic Server-Client Structure

Roxanne MacKinnon, Jason Lee
Williams College

Abstract

In this project, we implemented a basic server with basic implementation able to deal with multiple clients at a time. The communication between client and server was through HTTP compliant requests, specifically appropriate ”GET” requests by the client. We also provide a cost-benefit analysis of HTTP 1.0 versus HTTP 1.1.

1 Introduction

One of the most common capabilities a server must be able to provide is retrieve and send requested files to clients. However, this simple command requires several components to work in concert in order to be achieved. The client has to get the server's attention, which comes in the form of a sent request of service. The server then must listen for both these messages and provide the service if possible. We dealt with multiple clients through a process switching fork function. We also allowed for the client to specify whether the server should respond with HTTP 1.0 or 1.1 protocol. We analyze these decisions in terms of their potential benefits and drawbacks and review some benchmarks.

2 Architectural Overview

We designed our server on C, and used its socket functionality to create a listening port. This socket will then be on a forever loop, listening for connection requests made by clients. For each client that attempts to connect (limited by the size of our buffer), we make a child process through fork(), allowing the original socket to be solely for listening for connections. This also allows for processing multiple clients at once, increasing the uptime of the server for more individuals. In this child process, the server will now listen for a potentially valid request by the client, which also specifies either HTTP 1.0 or 1.1. HTTP 1.1 varies from 1.0 in its keep-alive principle. Thus, the server waits for a specified amount of time after fulfilling a request before closing the connection. This allows for the client to send multiple HTTP requests during a single connection, which prevents the overhead associated with starting and ending connections. If the request is well-formed, and is for a file that exists and is world-readable, then the server will transmit the file to the client in 4096-byte blocks. Otherwise, the server will send an error code. Our server is built to support the 400 (Bad Request), 403 (Forbidden), and 404 (Not found) error codes. In addition to status codes, the server also sends the HTTP headers for ‘Content-Length’, ‘Content-Type’, and ‘Date’. Content-type is determined by simply looking at the file extension, and Content-length is determined with a call to stat(). Our server supports the 4 MIME types text/plain, text/html, image/jpeg, and image/gif.

For our HTTP/1.1 timeout heuristic, we had a static timeout of 200ms. This is because our fork() model made it difficult to pass data between multiple processes, so it was hard to dynamically change this timeout. In practice, it would have been better to do the calculation before forking.

One limitation of our server is that it imposes a hard limit on filename size. We could not figure out a way to accept requests of arbitrary length. On the other hand, this may be a good thing for security purposes.

3 Evaluation

We used a series of benchmarks to test our server. Specifically, we used the ab apache HTTP server benchmark tool to evaluate the abilities of the web server. At a concurrency level of 25 with 300 requests, the tests took .025 seconds. The document length was 585 bytes, making a total transfer of 203,100 bytes. The mean time per request was 2.056 ms but only .082 ms across all concurrent requests. Most importantly, 100% of the requests were fulfilled within 5 ms.

Our design prioritizes uptime over speed. We saw a similar prioritization of reliability over speed with our selection of process switching over either event-driven or multi-threaded approach. Process switching inherently has more context-switch overhead which means a slower operation. However, it is reliable and easier to debug; multi-threaded could cause issues with synchronization if the concurrency causes situations such as race conditions. Event driven can resolve these multi-threaded issues but comes at the cost of being a more complex system. Also, further testing remains to be seen whether our multi-process approach makes the server program more portable as well between different hardware.

Another difference yet to be measured with our system is the costs and benefits associated with HTTP 1.0 vs 1.1. HTTP 1.0 is completely sufficient for our server, as it is providing a very simple file retrieval task, but 1.1 could provide greater speeds if we were to provide more possible services. This has to do with TCP Slow Start, which is a congestion control strategy at the TCP layer. At the start of every connection, the transmission rate is set low to avoid adding too much to a potentially congested network. The speeds ramp up with each successful transmission. However, HTTP 1.0 would make subsequent service requests much slower than necessary due to the mandatory closure of connection after each service fulfillment. Therefore, it can never ”ramp up” its transmission speeds. HTTP 1.1 fairs better as it would keep the connection alive for a time, its keep-alive interval, and would be able to maintain a higher rate of transmission after several transmissions. However, it is not strictly better since the server may be made to wait for a subsequent request by the client that may never come. This can prove to be inefficient when other clients are made to wait on this period unnecessarily due to congestion.

4 Conclusion

The coding of the web-server proved to be more convoluted than expected. It was interesting to see how much must be laid out to provide the simplest of requests. Moreover, it was interesting to make the system robust before worrying about additional features. I believe this to be a useful principle to keep in mind in the future. I am curious to see whether the reliability of the system should dictate its design or whether the function will inspire the engineering of the system. It is an interesting tradeoff we hope to delve deeper into in the future. In addition, it was truly impressive how the rules were laid out for our computers to communicate with one another. The portability of the code makes sense, but it is still very cool to see. Although they can only communicate with one another within the network, the universality of the language is always very cool, not unlike how it is cool to see the manner in which bacteria, fish, and humans run on the same operating system, DNA (maybe that’s a stretch).

The tradeoffs of such an accessible universal language are an interesting one. It makes it easy for someone to provide a service, but it also makes spoofing easy as well. It is important to keep in mind how to deal with such attacks in the future, but I appreciate the openness of this communication between computers, or maybe more importantly, end-users. The good-faith required is perhaps naive or required, but it is still appreciated.

Given the nature of the internet where many of the types of services are shared among different websites, it makes sense that large website providers could automate this process for their commercial clients. The headache of dealing with these distributed systems is likely strong enough to motivate payment for boiler plate code. I am certainly glad to hear we will use libraries in the future that will provide this web-server's implementation in the future projects.

One thing I noticed while testing the server on my browser was how much communication is done in the background. For example, after accessing the web-server, it confirmed the connection. However, when I looked it up in my browser, the autofill activated. In that moment, the server reconnected with the browser, meaning the browser sent for a connection without me pressing enter. This internal caching and predictive request was fascinating to see.