Event-Based Multi-Threaded File Servers

Aaron M. Taylor, Devin P. Gardella
Williams College

Abstract

This project gave us experience working with the fundamentals of how information is passed through the internet through a software implementation of a basic server. We chose to implement our server using a hybrid thread-based and event-queue based system for different types of operations. In the process, we learned about the HTTP protocol, establishing connections through system calls, sending and receiving information, and the joys of string parsing in C++. This project provided insight into the fundamental workings of servers used in many settings around the world. Understanding these elements is key to working with distributed systems of all kinds.

1 Introduction

This project centered around the creation of a server in c/c++. The server we choose to implement borrowed the best elements of two different ways to build the server. At the core of our server was an event-based structure that would handle the sending of data for all GET method requests by clients. Run by one of two permanent threads, this was implemented through a queue which stored Request structs. These structs hold all of the necessary data to handle the connections and are passes into most of the methods within our server.

The permanent threads of our server are as follows:

```
std::thread navi(mainListener, port);
std::thread mailman(eventProcessor);
```

The navi thread in the server is the main listening thread. It handles recieving all data from the sockets and parses the incoming requests. In turn, it takes these requests, and either deals with them immediately if they are simple, such as HEAD and OPTIONS, or passes them to the event queue if they require more processing, as in the case of GET. On top of this, we implemented a small-scale multithread system to handle HTTP/1.1 requests. Anytime a HTTP/1.1 request was made, we would spawn a light-weight thread with a timeout to handle listening to these established connections. These threads would operate in a manner similar to the main listener. Each of these threads would be detached from the main thread and would survive for no more than 20 seconds. To get this timeout duration, we implemented a simple herestic for decrementing this timeout in times of heavy traffic. This implementation allows for threading to take care of the lightweight tasks simply, and handles the heavier weight tasks of sending large amount of data in the more modern event-queue technique, combining the best aspects of

both approaches.

2 .htacess Support

Support for a .htaccess file would need a few more features added to our server for implementation. Firstly, in our Request structs we would need a new field that remembers the IP address of the client. This IP address can be accesses from the Socket struct created by a the accept system call. Additionally, because the .htaccess file will be accessed frequently, it should be loaded into a C++ object, probably a hash table of some sort, for constant time lookups of IP addresses. If this data structure contains the IP address of the client, the request could proceed, otherwise it would terminate and respond to the client with the appropriate error message and headers.

Further features could utilize the stat() system call to provide varying levels of access to different clients. For example, if a certain client that was on the .htaccess listing wanted access to protected files or files in a higher level directory, they could be allowed to do so by the server. This would enable access to the server on a higher level for system administration. If other HTTP methods such as POST and PUT were implemented, files on the server could be modified as well based on the permissions provided by the .htaccess list. However, with the ability to change files comes many security issues to worry about, so ensuring that the .htaccess file itself is secured and preventing clients from engaging in unwanted activity are essential implementation worries of client side modifications to the server files.

3 HTTP/1.0 vs. HTTP/1.1

The advantages of HTTP/1.0 are best seen when requests are singular and there are no embedded files, such as images, audio, or other media within the webpage that need to be loaded subsequently. In this scenario, there is no need to keep the connection open longer than to send the request and receive the response. Once the information is transmitted, keeping the socket open and continuing to listen simply imposes more of a burden on the server, and presumably may crowd the network with packets from the lower layers that keep the connection active and alive. Additionally, multiple HTTP/1.0 reqests can be sent to the server at the same time, and those can be sent and received in what is basically parellel. This would be beneficial if there are very large files whose transfer time dwarfs the slow-start, set-up, and tear-down times required for the creation of a new connection. In that case, there may be benefits to having parellel requests that are running across the network at the same time.

HTTP/1.1 if beneficial when many sequential calls are made to the server from the same client. This way, the overhead involved with creating a new connection is avoided. The situations when this is most beneficial would be webpages with lots of embedded content that also needs to be loaded such as images or css stylesheets, since these requests will be sent in quick sucession to the server. While some parellelism may be lost, for the size of most files that are used in web servers, usually comprising of text of highly compressed media for quicker transfers, the overhead of creating connections is a significant piece of a simple HTTP/1.0 request. For these reasons, HTTP/1.1 is very useful in situations which have lots of embedding. This is a

condition that browsers could actually check for after looking at the base page that is loaded, and thus the server doesn't actually have to worry about which is best, rather just supporting the client request in the way that it was requested.

4 Problems with Implementation

The major issue that arose because of how we implemented the server came when we tried to pass open filestreams between threads through structs. We found that when we tried to put the open filestream into a struct and store it in the event queue, it could not be open when removed from the event queue. This meant that for each call of the respondToGET() function we have to open and close the filestream each time, resulting in a source of overhead for the event queue. This perhaps was the biggest issue with our implementation. While a solution to this problem may have been possible, the complexity of transferring an open file pointer between different threads and objects proved to be a big time suck with no results beign seen, so at the advice of our professor we decided to leave as it is, opening and closing the file with each call to respondToGET().

Certain optimizations do exist. Instead of reading only one packet of data per respond-to-get function, we could have implemented it in such a way that we would read in more data, resulting in less overhead to the multiple open filestream calls. However, as we stuck with the function that sent a packet at a time because it remained more elegant, and would implement the workarounds if necessary for performance. It also distributed the server's functionality more evenly between clients, although this may not be noticable with an increase from one to four packets per message call.

Another issue we dealt with was attempting to use regular expressions. While these are incredibly useful structures in general, we realized rather quickly and after some frustration that the GNU g++ compiler has only a very partial implementation of the ¡regex¿ class, so the general useful functionality of regular expressions was not available to us. Instead, we looked to other techniques, such as a the method strtok(), which parses a string into tokens, and case analysis based on the kinds of input that we expect to be receiving. While regular expressions may have made our code shorter and somewhat nicer, we were able to implement all features just fine without them.

We also wanted to create a more dynamic getTimeout() function. This would provide more useful timeout values as the amount of threads present of the server increases. The current funtion provided a base value of 15 seconds with a linear function for the remaining 5 seconds based on the number of connections currently active. One useful idea might be to base this function off the size of the event queue as well as the number of active connections, since this will provide a dynamic function based not only on the number of connections, but also the load of GET requests that are currently being dealt with.

5 Conclusion

This project allowed us to gain hands on experience with the basic operations of distrubuted systems and servers. It was a unique and new experience to biuls a server from the ground up. These are systems that we use and interact with every day without thinking about it. By actually

creating a server, it gives us both an appreciation and an understanding of how these services are provided, as well as the challenges and implementation concerns and techniques that go into them. Moving forward this will prove a vauble experience as we explore Distributed Systems for the remainder of this course.