HW Report 1: Rudy Web Server

Ayushman Khazanchi

September 15th, 2021

1 Introduction

As part of Homework 1, the goal was to implement a basic web server in Erlang named Rudy that would service GET requests over the browser or a command-line. This was done by bringing together various small concepts around HTTP request parsing, understanding the HTTP protocol, implementing Socket communication, and serving responses on active sockets. One of the main topics that we try to understand via this assignment is the implementation and functionality of concurrency and distributed programming in Erlang.

2 Main problems and solutions

We begin by breaking down the components associated with our final web server. At first, we require a simple HTTP request parser. As we will be servicing only GET requests for now, we understand the HTTP protocol around making a GET request and use the *gen_tcp* library in Erlang to implement our request. The parser is built to service requests that have a URI, a version, any headers, and a body.

Rudy server is setup to use our HTTP parser using a socket connection. Initially, however the server serves only one request at a time. We can fix this by re-using the *handler* function that we built and calling it again after the request is fulfilled. This works to keep the server connections alive for multiple requests but leads to a different problem of the socket connection never closing. This is fixed by setting up *server.erl* which deals with the start and stop functionality of our Rudy server. The *stop* function in the *server.erl* provides a graceful way to shut down the server.

There is also a multi-threaded version of Rudy implemented in the *rudy_multi.erl* module. The test case of a 100 client calls is run against this implementation as well however we do not see much improvement in the timings here. In fact, the multi-threaded the implementation produces some unnatural response time spikes in a local Macbook shell environment.

The test cases run against both implementations of the server are shown below. They were run by using the internal IP address of the machine and the time is captured for 10 tests of 100 client calls each.

3 Evaluation

Figure 1 shows the performance of Rudy web server in terms of request response time over continuous client calls. The raw data for the same is shown in Table 1.

Request Attempt	1	2	3	4	5	6	7	8	9	10
Response Time	39657	36234	35812	37216	39153	42329	41405	39182	38091	40676

Table 1

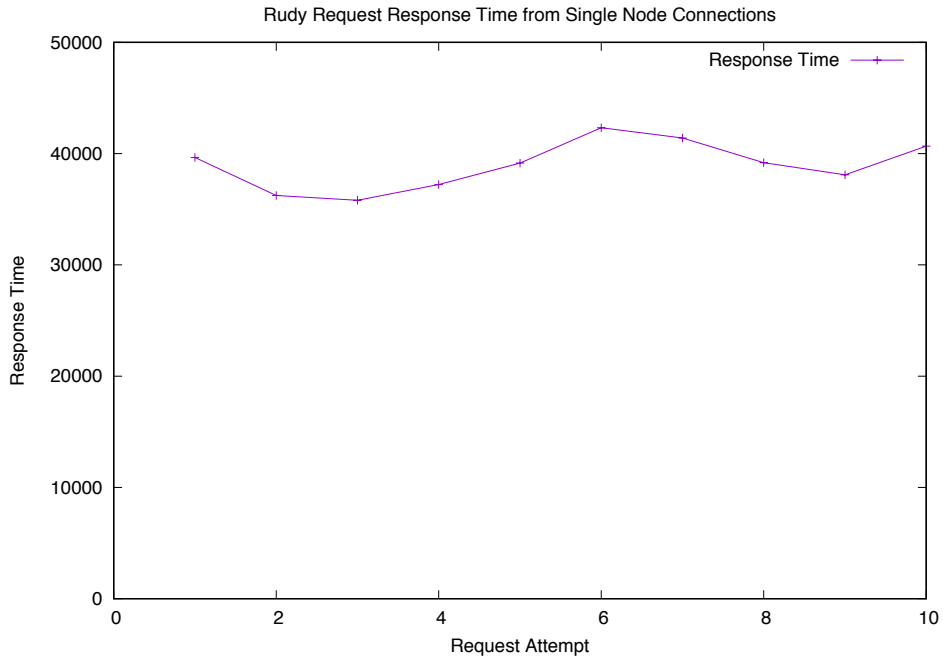

Figure 1

Figure 2 shows the performance of a multi-handler version of Rudy web server in terms of request response time over continuous client calls. The raw data for the same is shown in Table 2.

Request Attempt	1	2	3	4	5	6	7	8	9	10
Response Time	34847	35676	38258	39664	1041300	36395	38605	39425	42903	1045315

Table 2

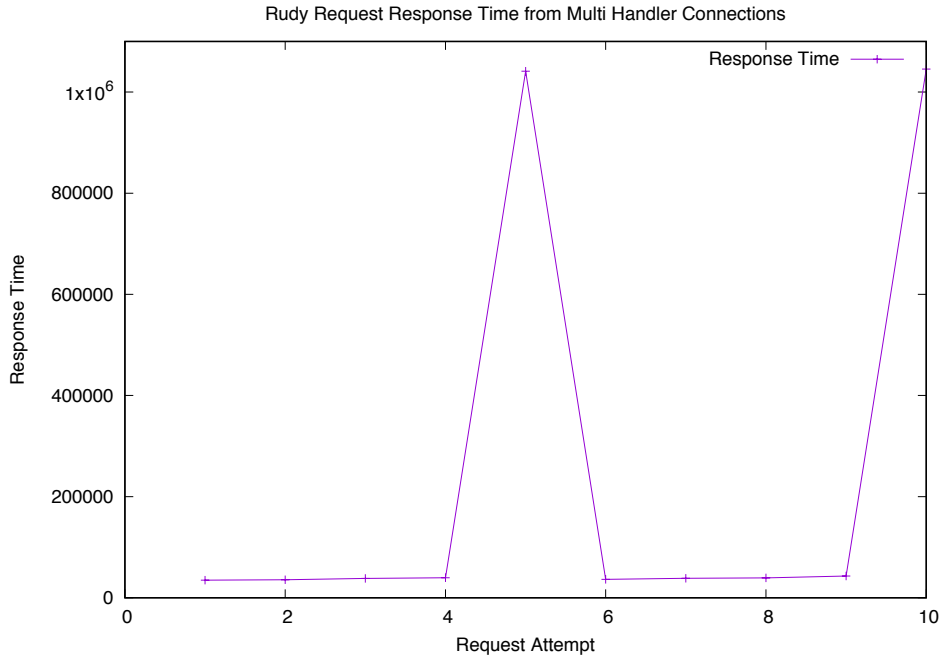

Figure 2

4 Conclusions

Through this exercise we've learned the basics of implementing a distributed, concurrent web server in Erlang. We were able to setup HTTP parsing and through it understand the basics of Erlang including Strings, Lists, socket connections, and modules. Implementing a multi-threaded solution of the Rudy server with a 100 handlers to serve requests helps to understand how to begin designing programs at scale and how Erlang can help provide a powerful environment to allow that. Apart from the few unnatural timings that we obtained from the multi-threaded solution testing, we can also see that the timings are fairly consistent and even with 100 simultaneous client calls our simple web server is able to maintain consistency at scale.