Rudy - Webserver

A simple webserver in Erlang

Distributed Systems ID2201

KTH

By: Hannes Rabo <hrabo@kth.se>

Chapter 1

1 Introduction

This report presents the result after coding a simple webserver "Rudy" that can serve pages including text document and simple images. The server tries to follow specifications for the HTTP protocol according to the standard [1], but many features has been left out to simplify the process. The server can easily be extended to include more of these features in case they are needed. In the current state it can serve most relevant static content that modern websites utilize. This includes basic images (PNG, JPEG), PDF and static textfiles containing CSS, JS and HTML. To serve other types of content (except plaintext), the content was written into the response body and MIME-type added the header. The type was exclusively inferred from file extensions as other methods seemed unnecessary for this simple purpose

The server comes in two versions with different approaches to the concurrency problem. Version one *"rudy"* spawns a new Erlang processes for each incoming request and consequently processes everything concurrently. The second approach that was implemented instead used a consumer/producer model where incoming requests were put into the processing queue and served by a fixed number of consumer processes (this server is defined in packages *"rudy_cons"*).

For the server implemented using the consumer model all incoming requests were uniformly distributed among the workers. This means that if any requests took longer than others to fulfil, it would not redistribute the work. This should be a minor problem in this case as we are only serving static content and we can expect the heavy tasks to be uniformly distributed by chance.

2 Chapter 2 Tests

Experiments were conducted using the provided test program and complementary testing was performed with *Apache Benchmark* as this is often used to test performance of HTTP servers. During the testing the server had a artificial load of 40 ms to perform each request. The load came from a sleep statement in the process method which means that the CPU was not loaded as much as it could have been in a real scenario.

The initial testing was performed to find the best approach to the threading problem. The different versions of the server and with different settings was compared to each other by running the same benchmark sequence in all configurations. In this test, each concurrent process (in the benchmark tool) tried to perform 10 requests. The result is illustrated in figure 2.1.

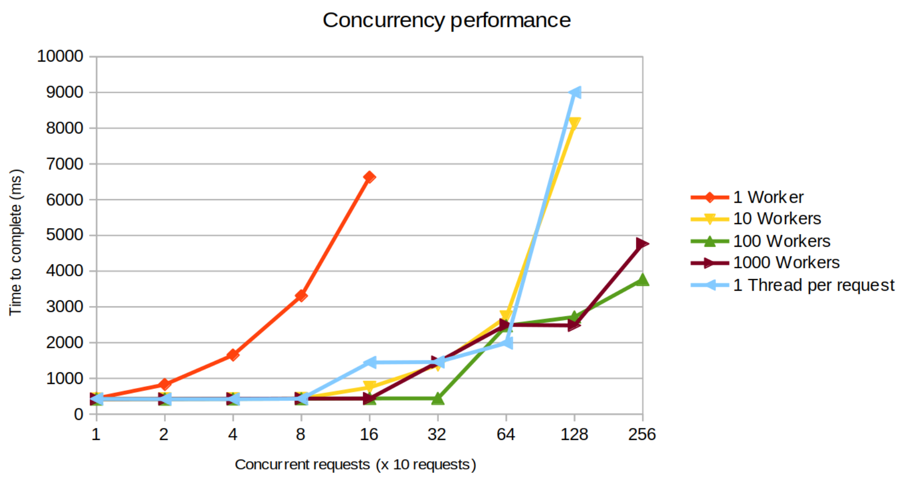

Fig. 2.1:

From this figure we can clearly see that a non-concurrent approach is unusable and some other technique is needed (see red line). The other approaches seemed to scale quite well for some time but then suddenly started to spike in execution time. At a first glance this seemed to be when the server got more concurrent requests than it could handle simultaneously. When looking closer we can see that this effect also appeared in the solution using the "1 request"-"1 process" which clearly does not have such a low limitation in the number of concurrent requests it could process.

The best performer in this test was the solution with 100 concurrent workers which scaled well up to around 256 concurrent requests. After this point the same type of problems as can be seen for the *10 workers* and *1 request-1 process* appear and the time rose to unusable levels. This leads me to believe that this is a operating system/hardware limitation in the number of concurrent connection that my computer can handle. In tests with Apache Benchmark this phenomenon instead showed up as a few connection timing out and increasing the average process time a lot.

When performing the same tests on a newer computer the problem with connections timing out also seemed to be smaller but still existing. This further strengthens the statement that this could because of hardware/operating system limitation.

When running apache benchmark with a total of 4000 requests and 1000 simultaneously connected clients (using 100 workers on the server) we get the following result.

```
Time taken for tests:   1.783 seconds
Failed requests:        0
Time per request:       0.446 [ms] (mean)
```

This means that we can serve approximately 135'000 requests each minute if we expect that the number of concurrent clients are 1000. This is a relatively high number for a initial attempt at a web server without any caching mechanics and is sufficient for most simple use cases.

3

Chapter 3

Improvements

There are many possible improvements for this server as it is quite rudimentary. The proposed extra tasks are all needed if we are going to create a server ready for production use. If we instead focus on my main goal, increasing productivity, the main problem here seems to be the requests that are timing out or taking a very long time to finish. One potential solution to this is to prevent the main loop from accepting more connections when we do not have any free workers. This would make the number of active connections drop to the same as the number of workers which possibly would make the connections more reliable as we are actively handling all incoming tcp connections instead of just letting them be there before we have time to complete the requests.

Another problem with this server is related DDOS attacks. As it only serves (and reads) from a fixed number of connections it is very easy to block any new users from connecting by just setting up this number of connections and send data slowly to prevent TCP time-outs. This means that the attacker does not have to have a lot of resources to effectively block the server entirely.

Bibliography

[1] W3C, “Hypertext transfer protocol – http/1.1.” Online [Visited: 2017-09-11], 06 1999. https://www.ietf.org/rfc/rfc2616.txt.