Report: Rudy - a small web server

Muhammad Azeem ([azee@kth.se](mailto:azee@kth.se))

September 8, 2020

1. **Introduction**

A basic web server was made using Erlang. A number of learnings materials were covered/implemented in the server, including:

* Concurrent programming through spawning processes
* HTTP protocol usage in a web server
* gen\_tcp library for Erlang socket API
* The io and file Erlang libraries for file reading
* The webserver was tested via various means such as GET requests through the same computer, through a mobile device as well as using Postman.

Furthermore, analysis was performed by experimentation. The data was plotted and then the critical analysis was performed to explain on the observed results.

Lastly, improvements were discussed an improvement as well of allowing file reads was implemented.

1. **Main problems and solutions**

* Firewall issues: Firewall were blocking access to the web server, which was disabled in order to test the server.
* Port number. The suggested port number in the exercise was 8080, however, I later switched to 8081 since 8080 was being in use by some of my other programs. At one point I also tried running 2 servers at once, which also required having 2 different ports.
* Non-public IP address. Unlike production web servers I couldn’t make API calls using a public IP address of course. However, I was able to make API calls within the same network.
* The server was rather rudimentary with only HTTP GET requests being accepted. I did make a slight improvement of allowing reading of files.
* There is no connection pool in this server. Which is understandable to some extent considering the limited work it is expected to perform. However, as we allow file reading and more clients to make calls, we can add a pool or processes which can take responsibility of a new request hence avoiding blocking new requests when an existing request is in processing.
* Error handling. A HTTP request can fail in between. Similarly, a file-read can also fail. Some basic Error handling and logging of errors was done in order to keep track of any issues while running the server.
* The HTTP request parsing can be a complicated task. However, in order to simply it, only the information needed for processing was parsed out leading to a much simpler parser implementation.

1. **Evaluation**

The test file was benchmarking was ran to get data on the speed of HTTP calls (request + response) from the server.

The HTTP calls were made 100 times consequently and the total time to make the calls and get a response was calculated.

Table 1 shows times as it took to make 100 HTTP calls. As well as max calls per second. Note the difference between calls were basic calls (simple textual GET request), calls with 40 μs (microsecond) delay. The calls for files were very variable depending on the file size.

|  |  |  |
| --- | --- | --- |
| Test Case | Total time for 100 calls | Requests per second |
| Same machine:  Text response with no added delay | 878291 | 113.89 |
| Same machine:  Text response with 40 msec added delay | 4726217 | 21.16 |
| Different machine in the same network with 40 msec delay | 7,906,961 | 12.65 |

Note: Average time for a 1 page txt file call (data of 10 calls): ~ 0.3 second  
Time really varied here depending on file size, increasing as much as > 3 seconds depending on file size. With the time increasing proportional to increase in file-size. (This is of course because of both, the time taken to read the file, as well as time taken to transmit it).

Some further observations:

* Delay in request in file is very dependent on the file size itself.
* There is a lot of difference in between HTTP calls with added 40msec delay and the ones without it. This could have been even more, depending on the machine being used.
* The delay on communicate between 2 machines a lot more than using the same machine. This is explainable, the added medium of message transmission over a network causes the delay.
* And of course, the delay is also now dependent on the performance of both the machines as well as network medium (Wi-Fi, Ethernet) and number of hops (a small network with single hop as compared to a bigger network requiring multiple hops).

Graph 1 shows the increase in time as requests are being made: Note this had to be shown as a log graph in order to simply observation of data.

Graph 1: The blue line shows the increasing time as more calls are made.   
X-axis: Sequence of API call, Y-axis: Log(time).

1. **Improvements:**

Many improvements can be made to this currently minimalist web server. The following discussion the effects and important of some improvements as well as discussing the implementation of one improvement done i.e. the ability to request files.

* 1. – Increasing throughput

Using a single process application works for a simple scenario of having a single client asking for rudimentary tasks such as asking for a simple text that is already hardcoded in the code.

However, this approach is inefficient with anymore processing required or increase in number of clients. If an API call requires making an IO call, reading from the database, calling another web-service or even doing some time consuming computation; there need to be other threads who can takeover any new incoming API requests. Similarly, we need more threads in case of multiple clients in case a client makes a API request while the system is already working to respond to an existing API request.

Making threads or spawning processes on the go is a slow process. A technique used to deal with this is called a ‘Connection pool’. A connection pool can have several threads waiting to take over the processing when a new API request is made. This way every new API request is dealt by another thread, hence, making for a smoother client-side experience as well as leading to a much lower number of requests being lost.

Lastly, a connection pool size can be set so that the web service can only have so many threads. Which leads to a web-server not taking hoarder the complete resources of the machine it is using. This also acts as a security measure.

* 1. – HTTP Parsing

A HTTP request can be of several types and can have many parameters, a body as well as several headers which provide useful information about the request. The current server does not process the headers and only processes GET requests.

This could be enriched to parse other forms of requests to more usability.

The headers provide useful information such as the client-information, the body-size, etc. All of which could be included to better understand the HTTP request and hence provide a better and more useable web-server.

* 1. – Deliver files

The main part of the web-server only responds with textual data.

An enchantment has been to now take the URI as the path of the file and return the binary file. The code of this can be seen under the methods ‘send\_file(..)’ and ‘read\_lines(..)’.

Some further improvement is possible by formatting the response depending on the mine-type of the file being sent.

* 1. – Robustness

The sever could be made much more robust in the various following ways by:

* Allowing control messages as API requests which can perform some setting changes, update parameters as well as shutdown the server (there is of course a need to implement security features in order to make allow control messages).
* Sockets can be kept active or closed depending on their need. Such as closing a socket if a request hasn’t been processed properly.
* Being able to respond with various HTTP Status in the response body is also of great importance. As of now only 200 OK responses which is an unconventional way of sending every type of response.