Report 1: Rudy - a small web server

Abdullah Abdullah

September 15, 2021

1 Introduction

This seminar consisted of implementing one of the basic architectures in distributed systems known as Client-Server architecture using Erlang programming language. The client and server communicate through HTTP using TCP which is one of the main protocols of the Internet protocol suite.

The task was to implement a small web server which consisted of Http parser, usage of socket API and gen tcp library.

Communicating and Messaging aspects of distributed systems is mainly covered in the seminar.

2 Main problems and solutions

Most of the code was implemented by following the document file provided. Some methods needed to be called in order to complete the rudimentary server application.

2.1 Missing Methods

After a socket is opened for Listening, it is passed to handler.

```
handler(Listen)
```

After connection, passing client to request and continue accepting new requests to handler

```
request(Client),
handler(Listen),
```

Parsing the request Str with http module parse request method.

```
Request = http:parse_request(Str),
```

2.2 Increasing Throughput

The current implemenation of the server had some downfalls and had low throughput. First the implementation of server was done as a single threaded or single process application so which means that the server was not able to handle the socket connections from multiple client requests concurrently. Another drawback was that if for any reason the process is blocked and requests keep coming from client side then it may stall because the main process is blocked and cannot handle them at the moment.

To increase throughput a better approach was to create a pool of handlers and assign requests to them which is also discussed in the document. Multiple handler processes run in parallel to increase the throughput.

So to implement this approach the handler process is spawned using spawn_link. spawn_link according to Erlang docs returns the process identifier of a new process started by the application. A link is created between the calling process and the new process, atomically. So if either process is killed the other will also be killed.

```
multiple_handlers_pool(0,_) ->
  ok;

multiple_handlers_pool(N,Listen) ->
  spawn_link(fun()-> handler(Listen) end),
  multiple_handlers_pool(N-1,Listen).
```

Here all handlers will wait for connection on socket and if connection is established, its request is dispatched to one of the available handler.

2.3 Problem in Parallel Requests from Benchmark

Adding multiple handlers pool increased the throughput in some way but through our test file we were sending 100 requests from client side using single client process which could be improved to send client side requests concurrently using multiple client processes which at the end contribute towards overall throughput of the application.

The approach was same as followed for creating processes for multiple handlers

```
run_parallel_tests(_, _, 0, _, _) ->
ok;
run_parallel_tests(Host, Port, N, R, Ctrl) ->
  spawn(fun() -> test_request(Host, Port, R, Ctrl) end),
  run_parallel_tests(Host, Port, N-1, R, Ctrl).
```

But during testing of 100 parallel requests the Rudy server started responding with a strange error of **{{badmatch, {error, econnreset}}}**.

The Erlang Docs mentions econnreset as error which occurs when connection is reset by the peer which means the result of gen_tcp:send/2 will return econnreset error when TCP result is sending RST Packet. Furhermore, from docs it was concluded that the connection is reset because the backlog value for gen_tcp:Listen is default to 5 and the backlog value defines the maximum length that the queue of pending connections can grow to. So when we make parallel requests from the bench file the rudy server cannot keep up with the connections from the request because of the backlog value which results in sending RST Packet and we get the error {{badmatch,{error,econnreset}}. To resolve this we have to pass backlog value (equal to the number of parallel requests you want to handle) to Opt array in gen_tcp:Listen

```
Opt = [{backlog, 100}],
gen_tcp:listen(Port, Opt)
```

After doing these changes, now parallel requests from multiple clients can be made and will be handled by multiple handlers at the server side.

3 Evaluation

The benchmark program test.erl was used to evaluate the throughput of rudy server on handling single and parallel requests.

Figure 1: Single Process Server and Client

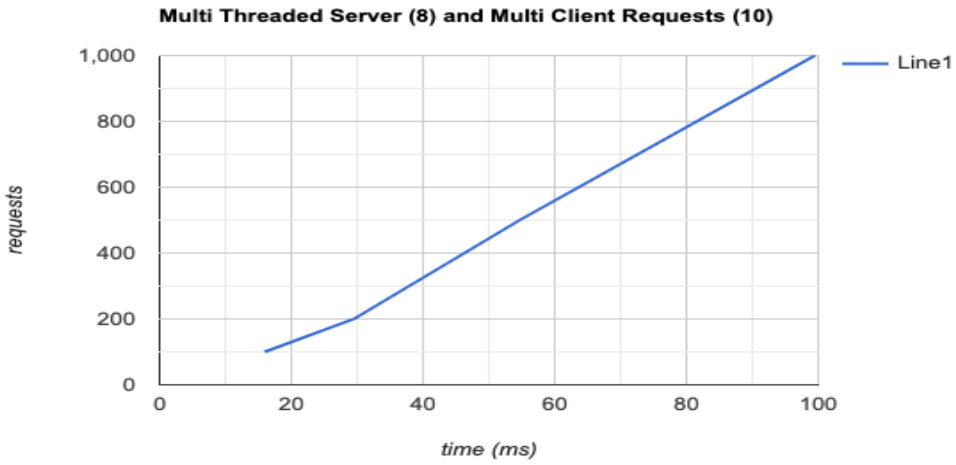

Figure 2: Multi Process Server and Client

Different number of requests from benchmark file were sent to Rudy server. From tests it was concluded that about 2800 requests per second can be served with the single handler and single client request. The adding of artificial delay is significant and the response time gets longer.

For figure 1 single handler process and single client process was used. The client process sends 100, 200, 500 and at last 1000 requests to Rudy server.

For figure 2 multiple handler processes (8) and multiple client processes (8) were used. The client processes sends 10 requests each, 20 requests each, 50 requests each and at last 100 requests each to Rudy server.

For parallel testing and evaluating the throughput, I ran different tests where the number of handlers were changed and the client processes were constant and making constant number of requests. The result is time in seconds which shows the time of the server responding to all the requests from client. 100ms delay was added to each request to test the parallel process more clearly.

No of Handler Process	No of Client Process	Num of Requests total	Time in seconds
2	10	100	4.04
4	10	100	2.12
6	10	100	1.42
8	10	100	1.03
10	10	100	0.82

Table 1: Multiple handlers and multiple client processes

4 Conclusions

The seminar introduced me to implement a client server program using Erlang programming and I learnd about http parser and how a request is processed and parsed.

I also got a chance to know about distributed nature of Erlang programming by add multi threading to the server and client sides so that multiple handlers can listen to the socket and handle parallel requests. There was a significant increase in throughput when multiple process handlers were created to handle parallel client requests.