Rudy: Seminar Report

Weherage, Pradeep Peiris

September 17, 2015

1 Introduction

Rudy is a simple web server implemented in Erlang to experiment the operations and procedures of Socket API.

For a web server a reliable exchange of data is vital, therefore TCP is used as the base communication protocol in server implementation. On top of TCP, the server accepts HTTP requests. A simple HTTP request parser in the server is designed to handle GET requests from the clients.

2 Main problems and solutions

Web Server is an application program that accepts its client requests (most of the time web browsers), retrieves the specified file in the request(or executes the specified server side procedure, eg. PHP, Java Servlet, etc) and returns a content/response back to the client.

The standardized communication protocol for both Clients and Server here is HTTP. HTTP allows a set of methods and headers that indicate the purpose of the request. The resource to which the HTTP method be applied are indicated by the Uniform Resource Identifier (URI), or as a location(URL).

HTTP presumes a reliable communication, and this can be solely handled within TCP protocol. TCP is a stateful protocol which takes care of handling lost packets and re-ordering them etc. Also it maintains isolated session between multiple senders and receivers. These features of TCP is well applicable for an application protocol like HTTP.

Rudy web server is included with a simple HTTP parser which can read HTTP requests from its client. To further simplify the solution only GET method from HTTP is considered. Rudy listens on HTTP request over TCP endpoints, the sockets.

Figure 1 shows scheme/lifecycle of TCP Socket in the server.

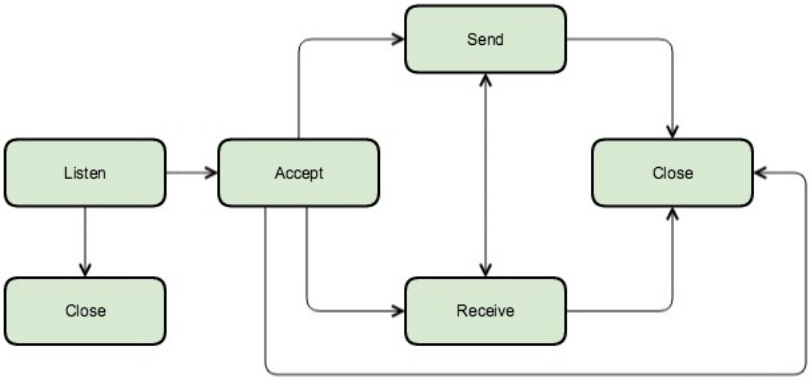

Figure 1: TCP Socket lifecycle in Server

TCP creates an isolated session for a client through the 'Listening Socket'.

```
{ok, ListenSocket} = gen_tcp:listen(Port, [list, {active, false},
 {reuseaddr, true}])
```

It is in charge of waiting for connection requests. Once the Listen Socket is open, any number of processes can take the Listen Socket and lock up to 'Accept' state and waiting for incoming requests.

```
{ok, AcceptSocket} = gen_tcp:accept(ListenSocket)
```

Now through Accept Socket the server can receive requests and send response back to the Client.

```
gen_tcp:recv(AcceptSocket, 0)
gen_tcp:send(AcceptSocket, Response)
```

Or close the communication session.

```
gen_tcp:close(AcceptSocket)
```

Putting it all together the Rudy web server is implemented to follow a simple process flow.

1. Start server with a single process
2. The server process listen on specified port and obtain the ListenSocket.
3. Once a client connected, the server process get a Client connection from ListenSocket and wait for a request from Client.

4. On client's request, the server process parse the request using simple the HTTP parser and send a text message to the client.

5. Once the request is handled, the server process wait again for a new client request over ListenSocket.

3 Evaluation

Rudy is first evaluated with a single client with incremental number of requests over locally and remotely. Throughput (Number of request per seconds) value at client end is used as the main measure for comparison. Then Rudy is set with a delay value at response to simulate the server side handling and compare the Throughput again at locally and remote clients. Finally Rudy is improved to handle multiple requests concurrently, and measure the improvement in Throughput with multiple user access.

3.1 One Client access with increase number of requests locally and remotely.

Locally, throughput value get increased with number requests. It seem Server could handle more requests locally than the number of requests issued from the client. Remotely it shows a average of 72 requests per second.

3.2 Rudy with response time delay

In previous version of Rudy, the response is send back immediately the request is parsed. But in real scenario, the server needs to access and read the resource specified in the HTTP request. And apply necessary transformation (e.g server side scripting etc) before the response. To simulate this behavior, 40ms delay is applied prior to the response.

Refer the Figure 3 for the outcome, which shows a constant throughput over increase number of requests in both locally and remotely.

3.3 Rudy with concurrent request handler

In previous version of Rudy, all the client requests were handled by a single process. We can improve it to have multiple processes listen on same Listen Socket and handle client request parallelly.

```
handler(Listen) -> case gen_tcp:accept(Listen) of
{ok, Client} ->
spawn(fun() -> handler(Listen) end),
request(Client);
{error, Error} -> error
end.
```

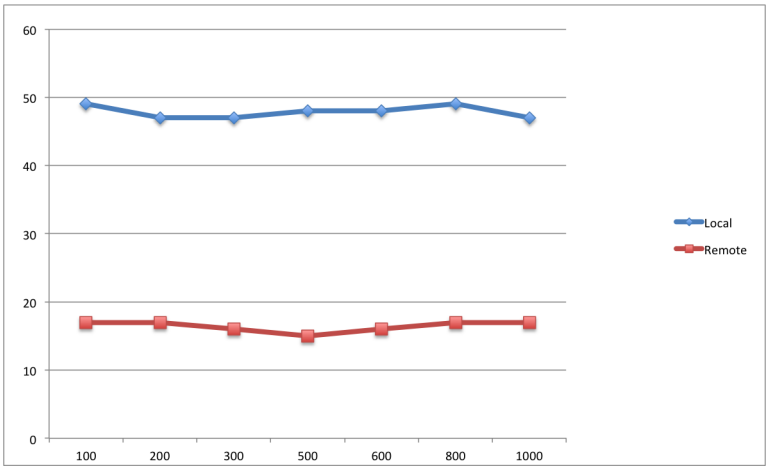

Figure 2: Throughput vs Number of requests, with delay in Server response

Figure 4 shows the test result for two version of Rudy servers with 5 concurrent client accesses over incremental number of requests. And It shows a significant improvement with having multiple processes handling client requests concurrently.

4 Conclusions

With final improvement of Rudy, it can now process client request concurrently. But it was encountered, when there are more concurrent users Rudy clients tend to fail in obtaining connections. The reason could be for each request the process has to be waited for connection from Listen Socket and then spawning a new handler again for the next client. In TCP, once the Listen Socket is open, any number of processes can take the Listen Socket and waiting for incoming requests. Therefore, Rudy could be further improved if we have a pool of handlers already set up and bind with Listen Socket. So that new client requests can be simply assigned to the handlers in the pool.

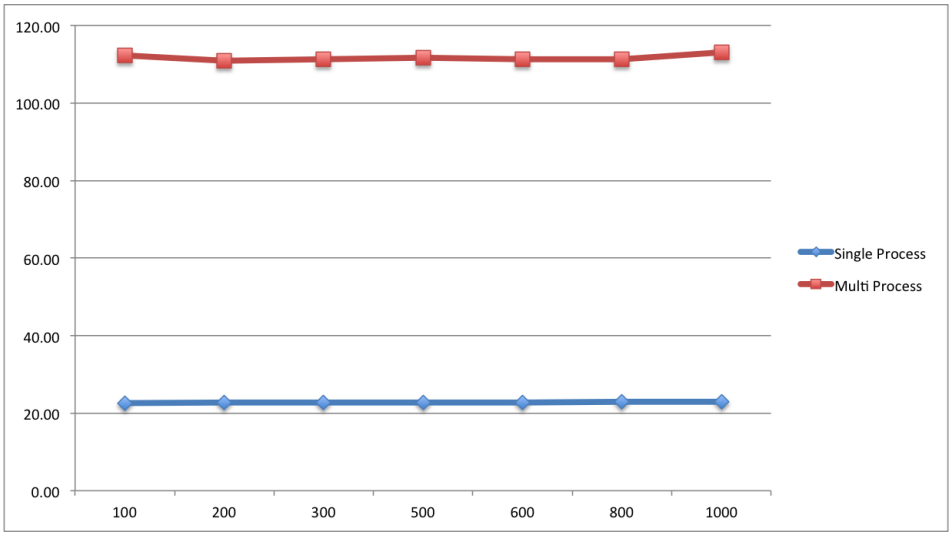

Figure 3: Throughput vs Number of requests, with Server handling concurrent requests