Rudy - a small web server

Xin Ren

September 9, 2017

1 Introduction

In this homework, a small web server and test modules acting as web clients are built with Erlang socket API. The server can handle HTTP requests that divided into more than one packets.

http.erl: module mainly used to parse http request
rudy.erl: the web server module
test.erl: module for benchmark test
test_body.erl: module to test the function to check message completeness

2 Main problems and solutions

1. How to decide if an HTTP request is completed?
   After checking rfc2616, I found out that if an HTTP request contains body, it usually has a Content-Length header whose value is the length of the message body. When it is absent, a request end with "\r\n\r\n" is completed, if not, match the length of the message body with the Content-Length value. Transfer-Encoding header is not considered here.

2. While trying to test if the function to check message completeness worked, the server socket closed after received the first part of the message. After debugging with TA, I found out that one line of code was missing to form a loop.

```
recv_server(Server)->
  receive
    {send, Msg} ->
      io:format("Sending Http Requests: ~p~n", [Msg]),
      gen_tcp:send(Server, Msg),
      recv_server(Server);
    {response, Recv} ->
      case Recv of
        {ok, Response} ->
          io:format("Received Http response: ~p~n", [Response]),
          ok;
        {error, Error} ->
          io:format("test_body: error: ~w~n", [Error])
      end,
      gen_tcp:close(Server)
  end.
```

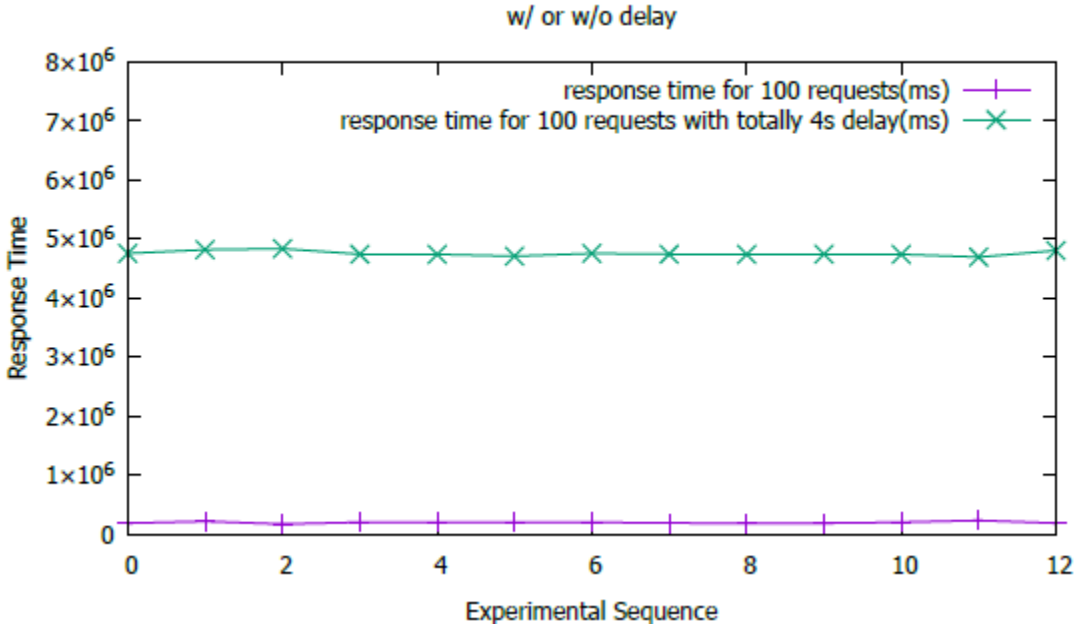
3. The idea of the test_body module is to send packets to the server and close the server socket once response is received, which requests the client to listen to the

response once it starts sending requests. And this means sending requests and listening to the response should be concurrent. Two new processes were spawned in the test_body module for this purpose.

```
start(Host, Port) ->
  Opt = [list, {active, false}, {reuseaddr, true}],
  {ok, Server} = gen_tcp:connect(Host, Port, Opt),
  spawn(fun() -> recv(Server) end),
  register(server, spawn(fun() -> recv_server(Server) end)).
```

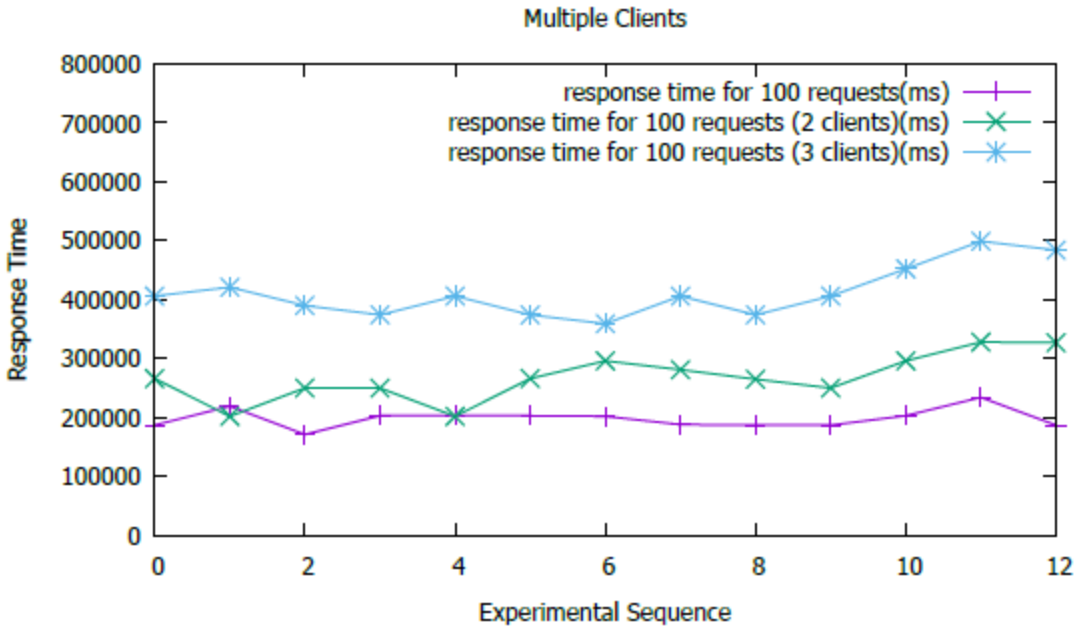
3 Evaluation

Some tests were done towards the performance of the server. Please see following figures.



In above figure, we can see that when 40ms delay was added to every request handling, the server took extra 4s to handle 100 requests, which means the artificial delay is significant. Because the server can only handle one request at the same time.

And in below figure, we can see that the response time for a single client got longer if more clients accessed the server at the same time. This is also due to that the server can only handle one request at the same time.



4 Conclusions

From the work, I have leant how to build a server and a client with Erlang socket API, and how to setup connections between server and client for messages exchange.