HW1: Rudy - A Small Web Server

Jian Wen

September 9, 2023

1 Introduction

This is the first homework project for distributed system. In this assignment, I learnt and applied some basic grammar of Erlang language, as well as the mechanism for a server-client transmission. In this assignment, I learnt from RFC 2616 about the structure of a request. Based on this, I have implemented the server that can response the http request from client by sending back simple reply.

As the first assignment of this course, I have learnt some basic knowledge of a distributed system and some features and programming skills in Erlang, which is fundamental for the following courses and projects. The web server implemented in this assignment is based on a server-client network, which is commonly used in the distributed systems.

2 Main problems and solutions

2.1 The first reply

In the rudy.erl, there are several functions init(Port), handler(Listen), request(Client),and reply(Request). the init() function will initialize the server and open a listening socket on a given port. This socket will be passed to handler(). A handler will wait for incoming connection from client and pass it to request(). After that, request() will parse the request by split the request string, and send to reply() to generate a reply to be sent back to client.

The functions with all blanks filled are shown below:

```
init(Port) ->
  ...
    {ok, Listen} ->  % Open a listening socket.
      handler(Listen),
      gen_tcp:close(Listen),
      ok;
  ...
```

```
handler(Listen) ->
  case gen_tcp:accept(Listen) of  % Listen to the socket for incoming connection.
    {ok, Client} ->
      request(Client),  % Receive and pass request to handle.
      handler(Listen);
  ...
equest(Client) ->
  ...
    {ok, Str} ->
      Request = http:parse_request(Str),  % Parse request through http module.
      Response = reply(Request),  % Generate reply.
      gen_tcp:send(Client, Response);  % Send response back to client.
    ...
reply({{get, _, _}, _, Body}) ->
  timer:sleep(40),
  http:ok(Body).
```

The code above did not consider deploying multiple handlers. The high-throughput version will be covered in the next section.

2.2 Optional task: increasing throughput

To increase the throughput of the server, concurrency will be needed. Luckily, Erlang provides good support for multi-process in programming, which made the implementation easier than Java or C++. To achieve this, I have created a pool of handlers. In this pool, I have created N new processes and each one of them runs a handler. This part is achieved by using recursion and spawn() function.

```
init(Port, N) ->
  ...
    {ok, Listen} ->  % Open a listening socket.
      handler_pool(Listen, N),  % Create a pool of N handlers.
      receive  % Wait for stop() command from input
        stop -> ok
      end,
      gen_tcp:close(Listen),
      ok;
    ...
handler_pool(Listen, N) ->  % Use recursion to create N handlers.
  if
    N == 0 ->
      ok;
    true ->
      spawn(fun() -> handler(Listen) end),  % Launch new process
      handler_pool(Listen, N-1)  % Recursion
  end.
```

3 Evaluation

Since concurrency is implemented, the evaluation then focuses on how total response time change as the number of handlers increase.

3.1 Total response time

In this part of evaluation, I have considered 100 requests (5 processes, each generate 20 requests) in total. By testing the bench time under different size of the handler pool, some relationship can be found. The result is executed 3 times and averaged.

Test case	Number of handlers	Response time (ms)
Case 1	1	4775
Case 2	2	2371
Case 3	4	1289
Case 4	8	1163
Case 5	16	1161
Case 6	32	975
Case 7	64	1132

Table 1: Server response time

It is interesting that if I use the original test bench code to directly generate 100 request, the server's response time won't change too much with different number of handlers. This might have to do with the time spent on the request generation. So to test the performance of concurrency, I have to update the test file so that it can generate request at multiple processes. Another good thing for this is the simulation will be more real, since each process can be considered as a client accessing a web page.

4 Conclusions

According to table 1, increasing the size of handler pool can increase server's efficiency when dealing with request from clients. However, the increase cannot goes further eternally because of the limitation of some other parameters. One explanation of this situation is that, the generation process of request at the client side and processing process at the server side may relatively get more obvious when each request is handled in time.

In this assignment, I implemented a simple web server using Erlang. Through coding, I have learned more about concurrency and http interaction between the server and clients. It's interesting to explore all the possible solutions and adjusting parameters to achieve a better performance in a system.