Report : Rudy - a small web server

Zhanbo Cui

September 15, 2021

1 Introduction

In this homework, I got familiar with Erlang and implemented a small web server with increased throughput. Concurrency, message and communication is three major aspects in distribution system and they are covered in this homework. The web server is implemented by using 2-tier client-server architecture, and this architecture is the most commonly used model for distributed applications.

2 Main problems and solutions

2.1 The first reply

The clients communicate with server through TCP protocol and there are some blank code block in homework.

```
init(Port) ->
        ...
    case gen_tcp:listen(Port, Opt) of % open a socket
        {ok, Listen} ->
            handler(Listen), % pass this socket to handler for listening/1
        ...
handler(Listen) ->
  % accept/1 can take out an established connection in the queue
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
           % pass the established connection to request/1
            request(Client),
           % calls itself recursively
            handler(Listen);
        ...
request(Client) ->
  % recv/2 receive the message from the client with established connection
    Recv = gen_tcp:recv(Client, 0),
        {ok, Str} -> % Str is the request message
           % The request message is parsed with parse_request/1
            Request = http:parse_request(Str),
        ...
```

2.2 Increasing throughput

We could create a new process (a new handler) for each incoming request, but it is not an ideal solution. Because if Rudy receive thousand of requests at the same time, it can't keep all reply processes going together. So a better approach would be to build a handler pool and set a reasonable number of handler processes in this pool.

```
 create_handlers(Listen, N)->
    case N of
0 ->
    ok;
N ->
    spawn(fun()->handler(Listen) end),
    create_handlers(Listen, N-1)
    end.
```

3 Evaluation

3.1 Change the number of handler

Firstly, I change the number of Rudy's initial handler processes in each test case and keep the number of client and the the number of request for each client to be fixed (4 client, each of them send 10 request). Each test case will repeat 3 times and calculate the average task finish time.

Test case	No.Initial handler	Average time(ms)
1	1	1657
2	2	832
3	3	585
4	4	423

Table 1: Change the initial number of handler

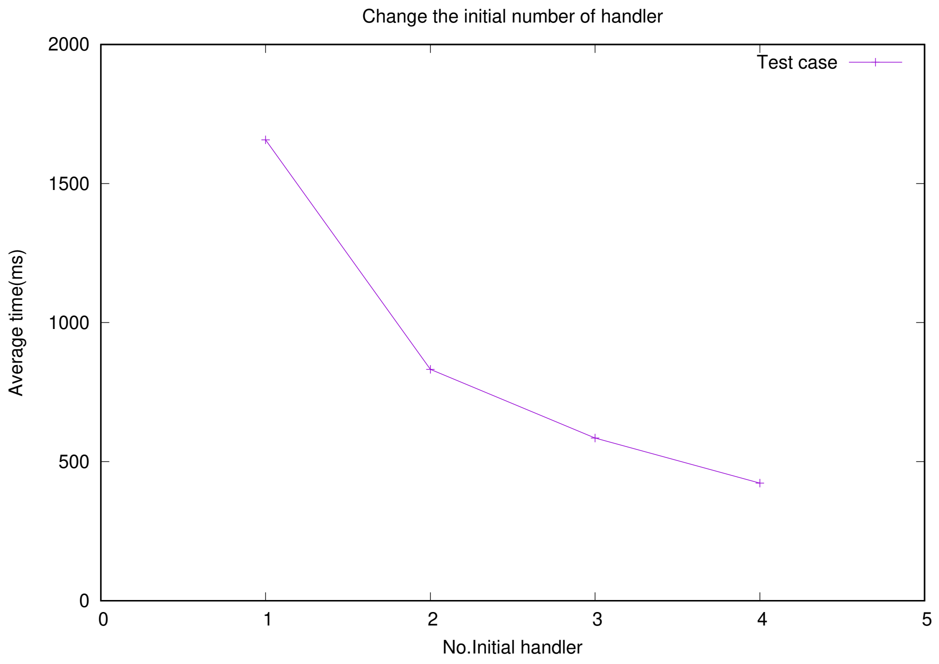

Figure 1: Change the initial number of handler

3.2 Backlog

Secondly, I set the number of Rudy's initial handler processes to 100, that means there are 100 handler processes are waiting to parse the request, but the parameter backlog is default value. Then I change the number of client and calculate the average task finish time.

Test case	No.Client	No.Request per client	Average time(ms)
1	1	100	4198
2	2	50	2104
3	4	25	1055
4	5	20	850
5	10	10	1439
6	20	5	2264

Table 2: Change the number of client

I guess the speed may be limited by the number of SMP (Symmetrical Multi Processor). This guess is probably correct, but the more critical factor is the parameter backlog. The default value of backlog is 5, that is why our finish time has been significantly increased when the number of clients is greater than 5.

```
Opt = [list,{active, false}, {reuseaddr, true}, {backlog,64}],
```

Test case	No.Client	No.Request per client	Average time(ms)
1	1	100	4314
2	2	50	2110
3	4	25	1051
4	5	20	847
5	10	10	426
6	20	5	216

Table 3: Add the value of backlog to 64

4 Conclusions

The Table 1 shows that the performance improvement are linear, it looks like we can get a faster response time if we continue to increase the number of initial handler processes, but this is not true, we must consider how many connection can be established between server and client.

To compare Table 2 and Table 3, the performance of Rudy has increased after changing the backlog parameter. Therefore, when designing a real server, we can set this parameter specifically under taking into account the load pressure.

Through this homework I gained a first insight into function programming and a new understanding of concurrency, TCP protocol and distributed system architecture. I also realised that when designing a system, we also need to consider the constraint that hardware imposes.