Seminar Report : Rudy

Vincent Delaunay

September 18, 2014

1 Introduction

Rudy is a small webserver. It listens to a connection, accepts sockets, parses requests and return the parameters.

This seminar is about how to handle tcp/ip connections and maximize throughput. It is very important for distributed systems as being being distributed is all about communicating. Thus it has ot be done in an efficient manner.

In this report I will present Rudy as well as RudyMproved, another webserver using parallelization to improve performances, serving static files and implementing a better parsing.

2 Listening to a port and accepting incomming connections

The webserver listens to a port for any tcp connection attempt, and accepts it.

Listening to a port is pretty straightforward using the builtins : every function that we need here is in the gen_tcp module

```
gen_tcp:listen(Port, Opt)
gen_tcp:accept(Listen)
```

We just need to make sure we have an infinite loop for accepting new incomming connections, and since we're in Erlang, the loop has to be done using recursion.

3 Handling the connection and receiving from the socket

Once the connection has been accepted, and the socket created, we want to receive the requests and then process them. This is actually where the *implementation determines the most the performance. The code examples are not strictly as in the source, but have been refactored for this document.*

In Rudy, we have the simplest approach : once a connection is accepted, it enters a handling method and reads the socke using a blocking call :

```
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            Recv = gen_tcp:recv(Client, 0),
            % Parse request.
            % Respond.
```

4 Parallelization

*Using parallelization, we want to avoid blocking the whole webserver when waiting for a single client to send its request. The code examples have been refactored to fit this document, but you can find the full implementation in RudyMproved.erl*and then process them

Using spawn to create new processes :

```
  loop(Listen) ->
        case gen_tcp:accept(Listen) of
              {ok, Sock} ->
                    Handler = spawn(fun () -> handle(Sock) end),
                    loop(Listen)
              % ...
  handle(Sock) ->
      case gen_tcp:recv(Sock, 0) of
            {ok, Data} ->
                    % Parse request.
                    % Respond.
```

5 Parallelization : using gen_tcp:controlling_process

Erlang offers an other way to manage reading from sockets. The code examples have been refactored to fit this document, but you can find the full implementation in RudyMproved_2.erl

Still parallelizing, another option we have is to remove the receive call (gen_tcp:recv). Using gen_tcp:controlling_process, we can bind the newly spawned process to its socket so that all data send to the socket is passed to the process as events.

```
  % The socket Sock has been created with the option {active, true} or {active, onc
  loop(Listen) ->
        case gen_tcp:accept(Listen) of
```

```
            {ok, Sock} ->
                  Handler = spawn(fun () -> handle(Sock) end),
                  gen_tcp:controlling_process(Sock, Handler),
                  loop(Listen)
            % ...
handle(Sock) ->
    receive
        {tcp, Sock, Data} ->
              % Parse request.
              % Respond.
    {tcp_closed, Socket} ->
        io:format("Socket ~p closed~n", [Socket]);
    {tcp_error, Socket, Reason} ->
        io:format("Error on socket ~p reason: ~p~n", [Socket, Reason])
```

But then we face a new issue. Because of {active, true}, or {active, once}, there actually is a race condition between the first packets and the call to gen_tcp:controlling_process. A packet received to soon won't be redirected to the right handler process. This issue is documented in several links shown as comment in the source code. The work-around that we implement here is a pre_handle function, which will block until the call to gen_tcp:controlling_process has been fully executed.

```
% The socket Sock has been created with the option {active, false}
loop(Listen) ->
      case gen_tcp:accept(Listen) of
            {ok, Sock} ->
                Handler = spawn(fun () -> enter_handle(Sock) end),
                gen_tcp:controlling_process(Sock, Handler),
                Handler ! under_control,
                loop(Listen);
            % ...
enter_handle(Sock) ->
    receive
        under_control -> ok
    end,
    handle(Sock).

handle(Sock) ->
    inet:setopts(Sock, [{active, once}]),
    receive
        {tcp, Sock, Data} ->
              % Parse request.
              % Respond.
        % Closed socket
        % Errors
```

6 Parsing

Once the request has been received, we need to parse it so we can extract its features. Namely, for HTTP : Method, Location, Version, Body..

In Rudy we craft a small pattern recognition module to match the known patterns.

Another option is to use the functin tokens, from the builtin string module

```
[Method, Resource | _ ] = string:tokens(Data," "),
case Method of
    "GET" -> Response = serve_get([".",Resource]);
    _ -> Response = response("501 Not Implemented", "501 Not Implemented")
end,
```

7 Response

Once the request parsed, we can compute what to respond to it, to send it over the socket.

Sending is as straightforward as Listening : we just use the gen_tcp:send function.

If we want to serve a file :

```
serve_get(Path) ->
    % Read the requested file.
    case file:read_file(Path) of
        {ok, Data} -> Response = response("200 OK", Data) ;
        {error, _} -> Response = response("404 Not Found", "404 Not Found")
    end,
    Response.

response(Code, Str) ->
    B = iolist_to_binary(Str),
    iolist_to_binary(
        io_lib:fwrite(
            "HTTP/1.0 ~s\nContent-Type: text/html\nContent-Length: ~p\n\n~s",
            [Code, size(B), B]
        )
    ).
```

8 Evaluation

How do our implementations perform under stress ?

With the subject, we are given a benchmark module to test our webserver. Since I have presented two other variants, we will look at the results for the three implementation (namely Rudy, RudyMproved and RudyMproved_2).

For the sake of testing, the implementations have been modified not to read any file, but just do the parsing, wait 40ms and reply a string.

```
5> rudy:start(8000).
6> rudymproved:start(9000).
7> rudymproved_2:start(10000).
8> benchmark:start("localhost",8000).
4351825
9> benchmark:start("localhost",9000).
4300618
10> benchmark:start("localhost",10000).
4312894
```

This is the time needed to reply to 100 requests. We can see it is the same for all of our three implementations.

The total time spent by the webserver just waiting is 0.04*100 = 4s. We can see a total overhead of 0.3s, so 0.003s overhead per request.

9 Evaluation in a parallel context

But what would happen if, instead of waiting for every request to be replied to before sending the next one, the benchmark would send all the requests at the same time ?

To represent this parallel context, I wrote an other benchmark, namely benchmark_parallel.erl.

To do so, we edit the run function to make it spawn the requests :

```
spawn(fun()->request(Receiver,Host, Port)end),
```

The tricky part is to notice when the request has been completed, or when an error occured. This been adressed using a Receiver, instanciated before spawning the requests :

```
receiver(N, Pid, Errors, Init) ->
    io:format("Errors : ~p/~p~n",[Errors,Init-N]),
    if N == 0 ->
        Pid ! {all_done, Errors};
    true ->
```

```
        receive
            done -> receiver(N-1,Pid,Errors,Init);
            error -> receiver(N-1,Pid, Errors+1,Init)
        end
    end.
```

The spawn requests tell the Receiver when they have been responded to, or if they have been rejected in return to gen_tcp:connect() or gen_tcp:recv()

When we run it (on a small dual-core netbook) :

```
10> benchmark_parallel:start("localhost",8000,100). % Rudy
Errors : 35/100
106020280  % 106s
11> benchmark_parallel:start("localhost",9000,100). % RudyMproved
Errors : 0/100
3273744 % 3.3s
12> benchmark_parallel:start("localhost",10000,100). % RudyMproved_2
Errors : 9/100
5491838 % 5.5s
```

Here, we can see some drastic variations between implementations. RudyMproved serves flawlessly in 3.3s, which is less than with the previous benchmark. RudyMproved_2 isn't as good, almost 10% failures, and 5.5s. But Rudy completely fails to deliver. 35% failures in 106s !

Increasing the number of requests (300) stresses theses results even more :

```
13> benchmark_parallel:start("localhost",8000,300). % Rudy
Errors : 242/300
111855043 % 112s
14> benchmark_parallel:start("localhost",9000,300). % RudyMproved
Errors : 6/300
4247593 % 4.2s
15> benchmark_parallel:start("localhost",10000,300). % RudyMproved_2
Errors : 15/300
14170132 % 14s
```

10 Conclusions

We have successfully implemented a minimalist webserver, with some variants. We can tell from the results that in case of a parallel load, the server really needs to be parallel as well in order to deliver. And in almost any real setup, especially in distributed systems, you will have multiple systems who might want to access your resources at the same time, so it is parallel.

A good question to ask, as it seems, would be : why does Rudy perform so poorly in the parallel setup ? Why doesn't this scenario boils down to the ẅorst caseünparallel scenario ? My answer is that when Rudy is flooded, as we can see, it starts loosing packets. Then, tcp adapts to it, resending them (more flood, and we already lost the time of a packet sent) and/or lowering the sending rate. Thus increasing the total time.

I have to confess being confused by the poor performances of RudyMproved_2, as it seemed to be the solution recommended by the erlang community. It makes sense though, as we have this overhead from gen_tcp:controlling_process and even more the IPC thing to avoid the race condition. But still, maybe it is a good and really clean way to implement the next step...

... Process pools. That's what alledgedly will allow us to perform better, by removing the overhead of spawning a new process for each request.