ID2201 HT23 Distributed Systems, Basic Course (50929)

Report Homework 1

by Eugen Lucchiari Hartz

September 13, 2023

1 Introduction

In this homework a small web server was developed using Erlang. To implement this a socket API was used. First a HTTP parser and then a simplified web server that uses the parser was developed. Some experiments and tests with the server are performed to analyze its efficiency. Furthermore, possible improvements of the server are discussed in this report.
Accordingly, this assignment is mainly about the structure and functioning of a server. Servers are a fundamental part of distributed systems, as they host and manage resources, services or data that can be accessed and shared for example by multiple clients across a network.

2 Main problems and solutions

The main problem in implementing the HTTP parser was for me to understand the code and consequently the intention of the various functions. Accordingly, the intentions of the various HTTP functions are are now briefly explained.

- parse_request/1: Parses an HTTP request message and extracts and returns the request line, an optional sequence of headers, a carriage return line feed (CRLF) and an optional body from the input
- request_line/1: Parses the request line of an HTTP request, this request line consists of a HTTP method (in this case focus on GET), a request URI, and an HTTP version
- request_uri/1: Parses the URI from the request
- http_version/1: Parses the HTTP version from the request and determines whether it's version 1.0 or 1.1
- headers/1: Two of them, one consumes a sequence of headers and the other one consumes individual headers
- message_body/1: Extracts the message body from an HTTP request, assuming that the body is everything that is left in this scenario.
- ok/1: generates an HTTP reply with status code 200 (OK)
- get/1: generates an HTTP GET request containing the specified URI and HTTP version

Subsequently, a simple server was implemented using the HTTP parser. During its implementation sometimes error messages came up, such as the following:

```
27> rudy:init(8080).
** exception error: undefined function http:ok/1
     in function  rudy:request/1 (rudy.erl, line 31)
     in call from rudy:handler/1 (rudy.erl, line 19)
     in call from rudy:init/1 (rudy.erl, line 9)
```

Fortunately, the implementation mistakes I made were easy to solve. I had forgotten to export the ok/1 function from the http.erl file. I have also received a similar error message when I have forgotten to run the http.erl file by doing c(http). before executing the rudy.erl file and its init function for example.

Another mistake I made that caused an error message was that I had to find out what I need to specify as Host when running the benchmark test.

```
73> test:bench(foo, 8080).
** exception error: no match of right hand side value {error,nxdomain}
     in function  test:request/2 (test.erl, line 21)
     in call from test:run/3 (test.erl, line 15)
     in call from test:bench/2 (test.erl, line 6)
74> test:bench(Eugens-MacBook-Pro.local, 8080).
* 1:30: syntax error before: '.'
74> test:bench('Eugens-MacBook-Pro.local', 8080).
4202923
```

After passing the correct host to the function the mistake was fixed and after fixing another syntax error which happen on a daily basis because I need some time to get used to Erlang's Syntax the benchmark test got executed properly.

As with the HTTP parser, I will again give a brief summary of the individual functions of the implemented server called rudy:

- start/1: starts the server, so registers the server process with the name rudy and spawns a new process to execute the init/1 function, which initializes the server.
- stop/0: stops the rudy server, so sends an exit signal to the rudy process using the exit/2 function
- init/1: Initializes the server, takes a port number, opens a listening socket, and passes the socket to the handler/1 function
- request/1: reads the request from the client connection, parse the request using the http parser and pass it to reply/1
- reply/1: takes as argument an HTTP method, URI, headers and a body, it then constructs an HTTP response with the message "Hello World!" in my case

This assignment was also about testing how many requests per second we can serve. It should also be investigated if the artificial delay added to the reply function of the server rudy is significant, or if it disappears in the parsing overhead.

Furthermore, it should be tested what happens when you run the benchmarks on several machines simultaneously.

3 Evaluation

To test how many requests per second we can serve (first without artificial delay) I run the benchmark test 10 times and report the result and show them in the tables below:

Without artificial delay, results first in microseconds for 100 requests:

Result in microseconds	Result in requests per second
55094	1812.25
66320	1506.15
38515	2593.60
76189	1310.51
59110	1692.88
64510	1549.38
56663	1764.59
75322	1327.08
81789	1222.41
68595	1456.43
Average: 66501	Average: 1693.42

Example of calculation to find out how many requests per second:
If we need for example 55094 microseconds for 100 requests we need 550,94 microseconds for 1 request. 1000000 microseconds are one second. So for the first case it would be for example: Requests per second = 1 / 0.055904 seconds which is equals to approximately 1812.25 requests per second.

With artificial delay timer:sleep(40) in the reply function again for 100 requests:

Result in microseconds	Result in requests per second
4418303	225.81
4229332	236.45
4224068	236.69
4227011	236.56
4218174	237.18
4215271	237.33
4220908	236.94
4224259	236.68
4278409	233.84
4292004	232.91
Average: 4232075	Average: 235.29

Accordingly, as there is a big difference between the result with and without the artificial delay, this artificial delay added to the reply function of the server rudy is significant. To be more concrete without the artificial delay the server can serve around 7 times more requests per second then with the delay .

Now the question is what happens when you run the benchmarks on several machines simultaneously. To test this there's another file called parallel_test.erl which simulates parallel requests. The bench function in this test has, in addition to the arguments host and port, the arguments C and N. C is the number of parallel clients, N is the number of requests each client makes. I run this modified benchmark test 10 times and report the result and show them in the diagram below

Without artificial delay, results first in microseconds with 1000 clients and again for 100 requests:

Result in microseconds	Result in requests per second, rounded to three decimal places
6890636	0.145
6945214	0.144
7018542	0.142
7004376	0.142
6990491	0.143
6889654	0.145
7755911	0.129
6972497	0.143
8244767	0.121
8215958	0.122
Average: 7106786	Average: 0.133

Looking at this result you can see that it makes a drastic difference how many clients or machines try to send requests to the server at the same time. To be more concrete when there's only one client (one machine) the server can serve over 1000 (more concrete 12716) times more requests per second than when there are 1000 clients (machines) who are sending (in our case 100) requests to the server at the same time.

Possible improvements

There are several possibilities to improve the implemented server, for example:

- Increasing throughput by implementing a multi-threaded solution where the server lets each request being handled concurrent. This can be achieved by create a pool of handlers and assign requests to them. If there are available handlers, the request will be put on hold or ignored.
- HTTP requests may not arrive in a single string and may be divided into multiple blocks. A solution could be to scan for a double CR, LF to mark the end of the header section and handle requests divided into multiple blocks, ensuring the completeness of requests.

- Extend the server so it can deliver files by parsing the URI request and and separate the path and file from a possible query or index.
- Trap exceptions and make the server more robust by making sure that the server is not shut down abruptly before all current requests are handled

The first mentioned improvement (Increasing throughput) has been implemented and test results are listed below:

According to implementation: handler_pool/3 function manages the pool. The spawn_handler/2 function spawns new handlers. The find_idle_handler/1 function looks for idle handler processes in the pool.

Result in microseconds with artificial delay	Result in requests per second, rounded to three decimal places
7093777	14.08
7209059	13.87
6952677	14.39
7004376	14.28
7172506	13.95
Average: 7080479	Average: 14.11

Even with the artificial delay this improved version of the server is more than 100 times faster then the other version of the server without delay.

4 Conclusion

In summary, the assignment was very useful to better understand the structure of a server process. In addition, one understands better how the HTTP protocol works and what it contains. It was very interesting to investigate about the efficiency of the server with different tests and under different conditions.