Report: rudy

Xiaoying Sun

September 9, 2020

1 Introduction

This report mainly includes the following aspects:

- Describe the procedures for using socket APIs.
- Describe the structure of a server process
- Summarize the impact of the amount of parallel threads and parallel handlers on response time.

2 Main problems and solutions

In order to implement the above procedures, we will need the functions defined in the Module *gen_tcp*. This module provides functions for communicating with sockets using the TCP/IP protocol.

The Figure 1 simply illustrates the procedure of how Erlang uses Socket APIs:

1. The server creates a listening Socket *Listen* to monitor the *Port* connected to the client.
2. The client calls the method $gen\_tcp : conncet()$ to establish a connection with the server and starts a three-way handshake.
3. The round-trip part in the middle represents the data exchange process between the client and the server. The $send()$ and $recv()$ methods are called.
4. In the last part, the client and server call the $close()$ method to close their sockets.

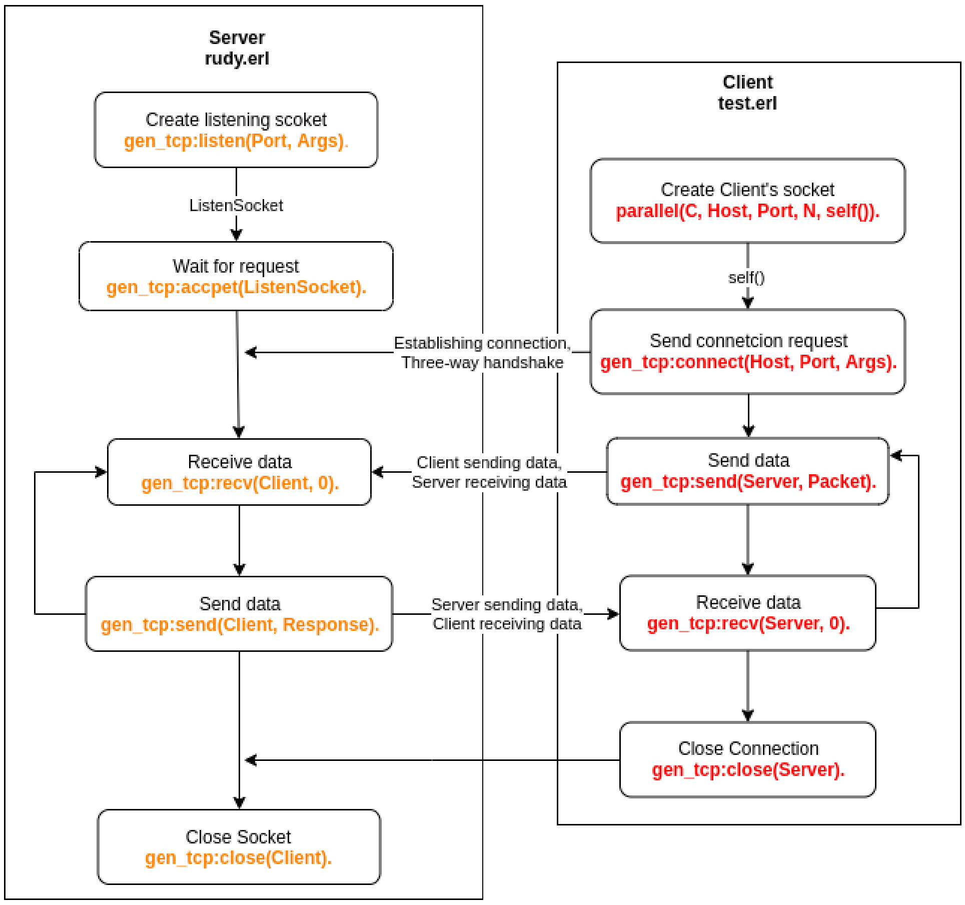


Figure 1: Erlang uses Socket APIs

When *rudy* creates a socket *ListenSocket* and let it monitor port 8080, it actually declares its possession of port 8080 to the TCP/IP protocol stack. When the declaration has been made, all TCP packets targeting port 8080 will be sent to *ListenSocket*.

And why can *rudy* create different socket to one port?

Actually, every socket includes two parts of information:

```
[server IP, server Port, client IP, client Port]
```

Therefore, *gen_tcp* : *accept*() can generate multiple sockets and as long as their client IP and client port are unchanged, they can monitor the same port. The only thing changed are server IP and server port.

3 Evaluation

To evaluate the performance of our server and have a clearer understanding of the parallel process. We call the *test* : *bench*() method to evaluate the response time of our server.

The server was tested with a test client which generated $C \times N = 100$ requests, where C represents the amount of parallel requesting threads, N represents the amount of sequential requests on one thread.By the way, the results strongly depend on the network status and CPU utilization so they're not unique.

No.of servers	Response time for $C \times N$ requests(s)						
	$(1, 100)$	$(2, 50)$	(5.20)	$(10, 10)$	$(20, 5)$	$(5, 20)$	$(100, 1)$
1	4.250	4.107	4.102	4.264	4.266	7.971	104.359(error)
2	4.286	2.147	2.100	3.452	2.328	14.502	126.648(error)
5	4.280	2.134	0.865	0.824	1.482	2.800	111.389
10	4.284	2.118	0.851	1.448	3.452	3.559	53.576
20	4.281	2.149	0.880	1.461	2.253	1.662	14.425

Table 1: The response time for multiple servers with C parallel threads to handle N sequential requests

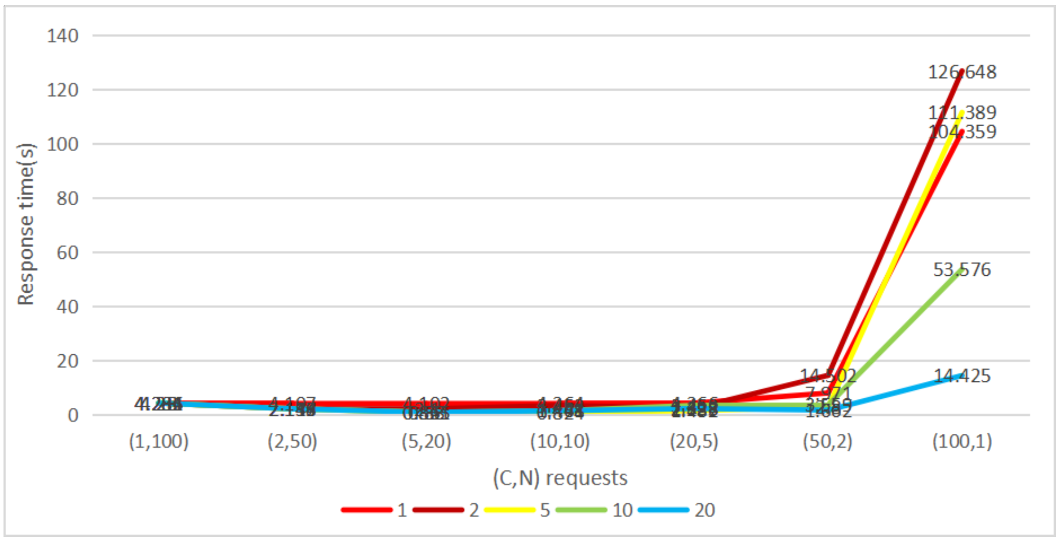


Figure 2: The performance for parallel requests towards the server with one or many parallel handlers

In figure 2, the response time climbed sharply in column $(100, 1)$ and the server with 1 or 2 handlers occurred thread blocking.

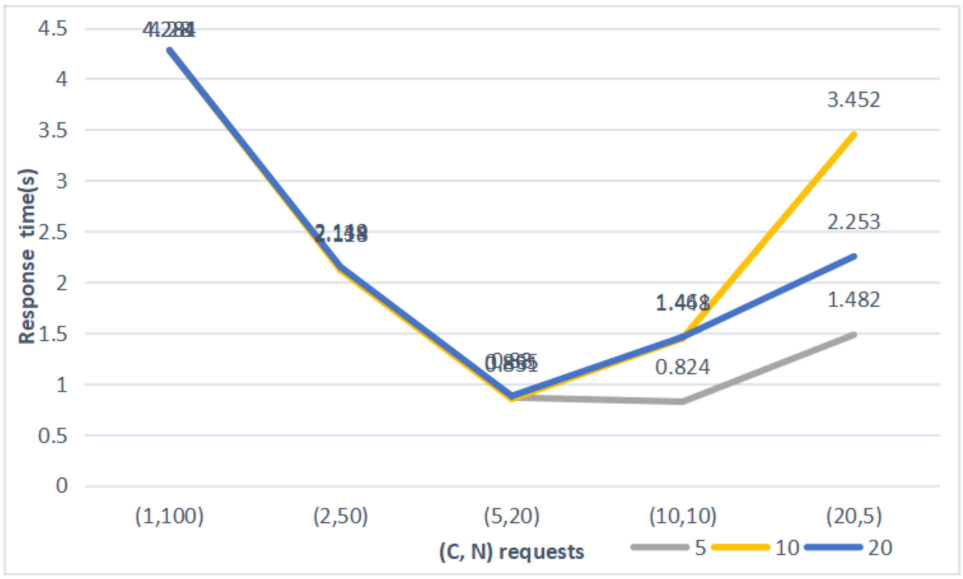


Figure 3: The performance of the server with 5, 10 and 20 parallel handlers

As we can see from figure 3, at first, the response time decreased with the addition of threads, however, when the number of threads reached a certain value, the response time increased with the addition of threads instead.

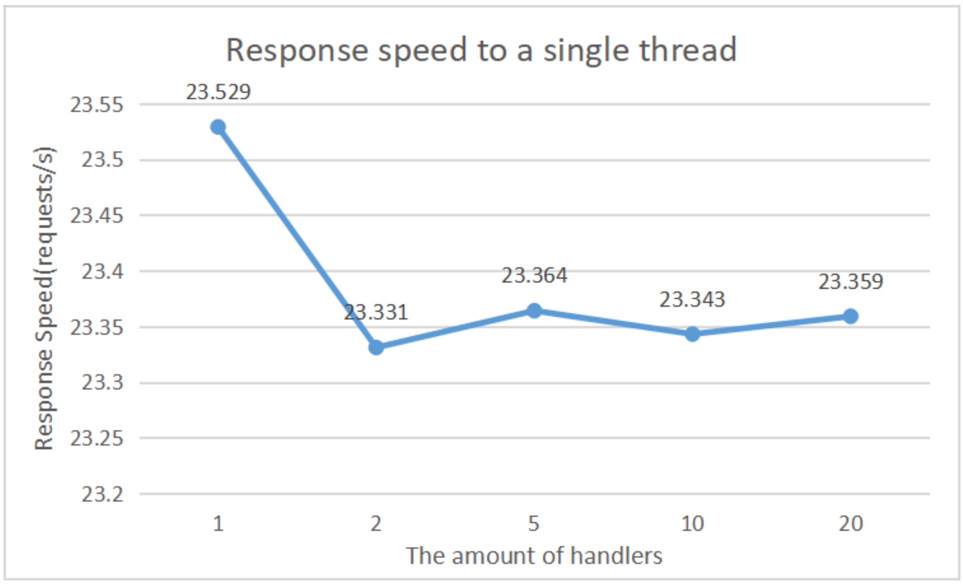


Figure 4: The performance of the one or multiple server towards a single request

Figure 4 and table 2 illuminates for a single thread, the response speed

	The amount of handlers				
	1	2	5	10	20
Response speed(requests/s)	23.529	23.331	23.364	23.343	23.359

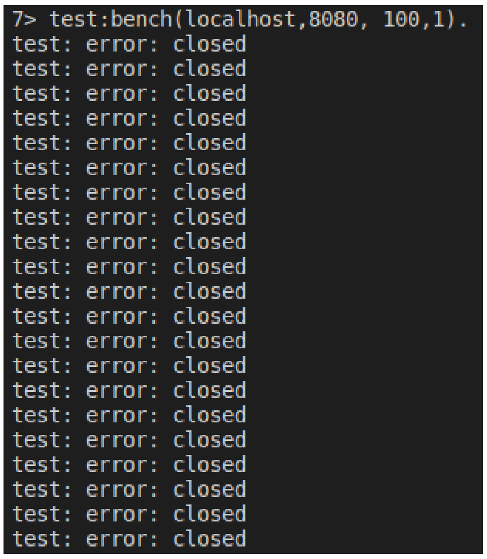
Table 2: Response speed for a single thread

of multiple processors is not as fast as a single processor.

4 Conclusions

From the column of $(100, 1)$, which means created one thread for each request and executed 100 times, we found that at the end of communication, the thread was blocked and the server failed to accept the user's connection request.

Why did it happen?



```
7> test:bench(localhost,8080, 100,1).
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
test: error: closed
```

Figure 5: Thread blocking

For the one-request-one-thread model, every time a request is accepted, a thread will be created to response the business logic of this request. Apparently, if the client sends a large amount of requests, the server will create the same large amount of threads, which is unrealistic. There are the following reasons:

1) Creating a thread consumes system resources. Hence, limited hardware resources limit the amount of threads in the system.

2) After one request is completed, system needs to destroy this thread. If the request volume is very large while the business logic is very simple, it will cause frequently create and destroy a thread.

3) When there are many threads in the system, the context overhead will be extremely large. For example, if the requested task is an IO-intensive task, which often needs to be blocked, it'll definitely cause frequent conetxt switching. (The last case isn't reflected in our test.

Above all, creating multiple threads can speed up the response to a certain extent. However, when the time of threads' creation and destruction is longer than the time of executing the task, the response time will increase on the contray.

On the other hand, multiple servers only works when dealing with multiple clients. Otherwise, multiple handlers may be less efficient when processing a single request.