Report I: Rudy - a small web server

Chuan Su

September 11, 2018

1 Introduction

In this seminar, our task is to implement a tcp server together with a small benchmark program in Erlang programming language to measure server performance in terms of response time. This report presents the major problems encoutered as well as our solutions to improve server throughput under concurrent connections.

2 Main problems and solutions

2.1 Multithreaded-application

After completing the rudimentary server application, we discovered that it was implemented as a single-threaded application in which server may not be able to accepts socket connections from multiple remote clients concurrently. Furthermore all requests may stall while the single thread is blocked. To avoid the problem that blocking the main thread from accepting new requests we decided to allocate a separate thread for each connections.

Code snippet below illustrates our approach to asynchronous requests handling in Erlang.

```
handler(Listen) ->
  case gen_tcp:accept(Listen) of
    {ok, Client} ->
      spawn(fun() -> request(Client) end),
      .....
  end,
  handler(Listen).
```

2.2 {error, econnreset}

Rudy server started responding {`error, econnreset`} when performing our benchmark tests initiating over 10 parallel requests.

Code snippet below illustrats our approach to initiate parallel requests in the benchmark program.

```
run_test(_Host, _Port, 0) -> ok;
run_test(Host, Port, P) ->
  spawn(fun() -> test:request(Host, Port) end),
  run_test(Host, Port, P - 1).
```

Erlang man page explains that `econnreset` is an error code indicating connection reset by tcp peer. The call `gen_tcp:send/2` will return {`error, econnreset`} when it is detectd that *a TCP peer has send an* `RST` *packet.*

Reading about `RST` in TCP protocol we learned about that the cause of sending `RST` varies but lies on finer details of the 3-way handshake for establishing a TCP connection and **the queue for pending connections at listen socket**, aslo known as `backlog queue` in `Socket API` .

Today's TCP implementation uses two queues for incoming connections:

- a `SYN` queue for incomplete connections that are waiting for `ACK` packet
- an accept queue for established connections that are to be consumed by listen socket.

Connections from `SYN` queque will be moved to accept queue after receiving conresponding `ACK` packet. However `ACK` packge will be ignored (even without sending `RST` packet) if the accept queue has reached its maximum length specified by `listen(2)` method (see code snippets below).

```
/* copied from Linux man page */
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

On the ground of reliable connections design of TCP protocol tcp peer will re-transmit the `SYN/ACK` packet to the tcp client and **send `RST` packet once the maximum retries have reached**.

While performing stress tests initiating a number of parallel requests from benchmark program rudy server cannot keep up with the connections flow delievered from tcp clients with the default maximum length configuration of backlog queue , which result in sending the `RST` packet.

Increasing the `backlog` queue size to the amount of parallel requests helped resolve `econnreset` error in performing stress tests.

Code snippet below illustrats the method of setting maximum length of backlog queue in Erlang.

```
Opt = [list, {backlog, 200}],
case gen_tcp:listen(Port, Opt) of
  ...
end.
```

3 Evaluation

We have written a small benchmark program in Erlang `stress_test.erl` which allow us to evaluate the performance/throughput of Rudy server on handling parallel requests.

The evaluation result is presented on the table below:

	10	20	40	80	160	200	300
1	44366	46588	50240	57761	1063544	1065454	1075021
2	43822	44626	46793	51377	59892	64024	76423
4	43020	44018	45015	49591	55293	57161	62440
8	43207	44546	45986	48878	54118	56456	58572

The header row indicates *the amount of parallel requests* made to rudy server application while the first column indicates *the number of erlang scheduler threads* enabled in Erlang VM. And other cells indicate server processing time in microseconds.

With the support of SMP (Symmetric Multiprocessing capabilities), we can easily control how many schedulers to be created/enabled, for instance, Starting Erlang VM with comman `erl -smp +S 1` instructs Erlang VM to run erlang processes on a s single scheduler thread while `erl -smp +S 8` will enable 8 scheduler threads.

Benchmark test result tells that the performance differences are subtile on processing up to 80 concurrent requests regardless of the amount of scheduler threads. However application running with multiple scheduler can differ significantly from one running on single scheduler when the amount of parallel requests start reaching around 160.

Moreover, it is remarkable that the performance differences are not obvious between 4 and 8 schedulers under 200 concurrent requests. In fact, with below 40 concurrent connections, application with 8 schedulers support performs slightly poor in contrast to 4 schedulers, which proves that increasing threads will not provide performance benefit on processing a small amount of concurrent connections.

4 Conclusions

This seminar introduced me Erlang's message passing multi-threading model and enhanced my understanding of Socket API, Http and TCP protocol, especially TCP layer behavior upon connection overflow.