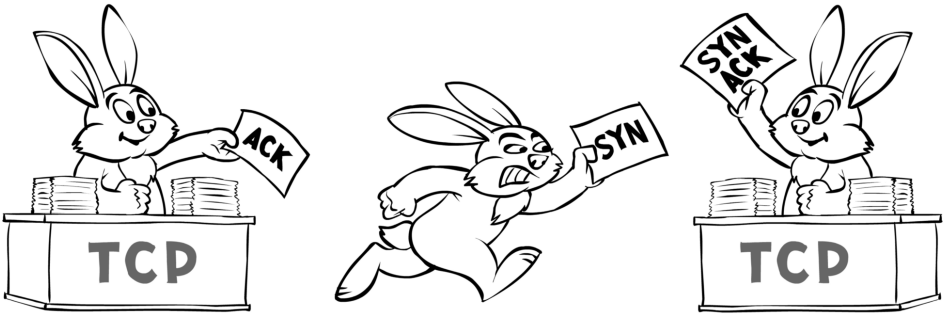
Sachin Malhotra Follow
Masters Student at University of Southern California | Student Researcher @ISI | edorado93.github.io
Mar 26, 2017 · 9 min read

Load Testing HAProxy (Part 2)



This is the second part in the 3 part series on performance testing of the famous TCP load balancer and reverse proxy, HAProxy. If you haven't

gone through the previous post, I would highly suggest you do so to get some sort of context.

Load Testing HAProxy (Part-1)

Load Testing ? HAProxy ? If all this seems greek to you, don't worry. I will provide inline links to rea...

medium.com

This post will focus on the TCP Port exhaustion problem and how we can deal with it. In the last post we talked about how we can tune the kernel level and process level ulimit settings. This post is focussed on modifying the sysctl settings to get over the port exhaustion limits.

SYSCTL Local Port Range and Orphaned Sockets

Port exhaustion is a problem that will cause TCP communications with other machines over the network to fail. Most of the times there is a single process that leads to this problem and restarting it will fix the issue, temporarily. It will however come back to bite in a few hours or days depending on the system load.

Port exhaustion does not mean that the ports actually get tired. Of course, that is not possible because the computer is not human and the ports are not capable of getting tired. The truth is much more insidious.

Port exhaustion simply means that the system does not have any more *ephemeral ports* left to communicate with other machines / servers.

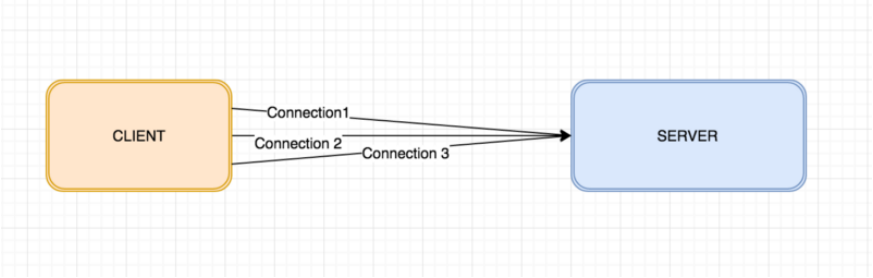
Before going further, *let us understand what constitutes a TCP connection and what really does an inbound and an outbound connection means.*

In majority of the cases whenever we talk about TCP connections and high scalability and ability to support concurrent connections, we usually refer to the number of inbound connections.

Say, the HAProxy is listening on port 443 for new inbound connections. If we say that the HAProxy can support X number of concurrent connections, what we really mean are X number of incoming connections and all of them are established on port 443 on the HAProxy machine.

If these connections are inbound for the HAProxy, then these have to be *outbound for the client machines where the connection originated.* Any sort of communication from the client requires them to initiate outbound connections to the servers.

> *When a connection is established over TCP, a socket is created on both the local and the remote host. These sockets are then connected to create a socket pair, which is described by a unique 4-tuple consisting of the local IP address and port along with the remote IP address and port.*



3 TCP connections from client to server/proxy

If you understood the concept of the quadruple, you will realise that in an outbound connection or rather multiple outbound connections to the SAME backend server, 2things always remain the same i.e. Destination IP and Destination Port. Assuming we are only taking into account a single client machine, the client IP will also remain the same.

This means that the number of outbound connections is dependent on the number of client ports that can be used for establishing the connection. While establishing an outbound connection, the source port is randomly selected from the ephemeral port range and this port gets freed up once the connection is destroyed. That's why such ports are called as ephemeral ports.

By default, the total number of local ephemeral ports available are around 28000.

```
sachinm@ip-192-168-0-100:~$ sysctl net.ipv4.ip_local_port_range
net.ipv4.ip_local_port_range = 32768    61000
```

Now you might be thinking that 28k is a pretty large number and what can possibly cause 28k connections to get used up at a single point of time? In order to understand this, **we have to understand the TCP connection lifecycle.**

During the TCP handshake, the connection state goes from

SYN_SENT → SYN_RECV → ESTABLISHED. Once the connection is in ESTABLISHED state, it means that the TCP connection is now active. *However, once the connection is terminated,* the local port that was being used earlier does not become active immediately.

The connection enters a state known as the *TIME_WAIT state for a period of 120 seconds before it is finally terminated.* This is a kernel level setting that exists to allow any delayed or out of order packets to be ignored by the network.

> *If you do the math, it won't take more than **230 concurrent connections per second** before the supposedly large limit of 28000 ephemeral ports on the system is reached. This limit is very easy to reach on proxies like HAProxy or NGINX because all the traffic is routed through them to the backend servers.*

When a connection enters the TIME_WAIT state, it is known as an ***orphaned socket*** because the TCP socket in this case is not help by any socket descriptor but are still held by the system for the designated time i.e. 120 seconds by default.

How to detect this?

Enough with all the theoretical stuff. Let's jump in and see how we can identify if this limit has been hit on the system. There are two commands I absolutely love to use to find out the number of TCP connections established on the system.

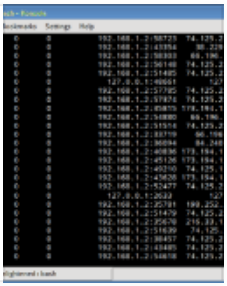
ss (Socket Statistics)

The socket statistics command is a sort of replacement of the famous netstat command and is much faster than the netstat command in rendering information because it fetches the connections info directly from the kernel space. In order to get a hang of the different options supported by the ss command, check out

10 examples of Linux ss command to monitor network connections

In a previous tutorial we saw how to use the netstat command to get statistics on...

www.binarytides.com



The `ss -s` command will show the total number of TCP established connections on the machine. If you see this reach the 28000 mark, it is very much possible that the ephemeral ports have been exhausted on that machine. ***BEWARE:*** This might be higher than the 28k number if multiple services are running on the same machine on different ports.

Netstat

The netstat command is a very famous command that provides information about all sorts of connections established on the machine's networking stack.

```
sudo netstat -anptl
```

This will show you the details about all the connections on the machine. The details include

- local address
- remote address
- connection state
- process pid

We can also use this to see if a single process has established 28k connections to an outbound server which gives us insights into the port exhaustion problem.

```
sachinm@ip-192-168-0-122:~$ sudo netstat -anptl | grep 443 | head
tcp        0      0 192.168.0.122:6018      192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:37607     192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:5195      192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:20232     192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:7894      192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:40408     192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:6831      192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:54080     192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:46377     192.168.0.168:443       ESTABLISHED 9758/vegeta
tcp        0      0 192.168.0.122:63188     192.168.0.168:443       ESTABLISHED 9758/vegeta
```

For eg:- the above image shows that a process with pid 9758 has established multiple connections with the foreign machine with IP 192.168.0.168 and port 443. As we can clearly see, on the source side of things, there are numerous ports being used.

```
sachinm@ip-192-168-0-122:~$ sudo netstat -anptl | grep
'192.168.0.168:443' | cut -c69-79 | sort | uniq -c | sort -
rn
```

```
5670 ESTABLISHED
```

This modified command will show the status of the different connections established with 192.168.0.168 on port 443. Currently there are 5670 connections. If this limit were to reach 28k, then you should look at options to increase the ephemeral port range on the machine.

Let's look at another interesting command that you can issue at the server end or the proxy end to find out how many inbound connections have been established and by which IPs. So for example check out the result of the below command

```
ss -tan 'sport = :443' | awk '{print $(NF)" "$(NF-1)}' | sed
's/:[^ ]*//g' | sort | uniq -c
```

```
sachinm@ip-192-168-0-168:~$ ss -tan 'sport = :443' | awk '{print $(NF)" "$(NF-1)}' | sed 's/:[^ ]*//g' | sort | uniq -c
      1 * *
   2342 192.168.0.122 192.168.0.168
   2340 192.168.0.123 192.168.0.168
   2344 192.168.0.124 192.168.0.168
   2336 192.168.0.125 192.168.0.168
   2338 192.168.0.126 192.168.0.168
   1018 192.168.0.138 192.168.0.168
   2347 192.168.0.141 192.168.0.168
   2349 192.168.0.142 192.168.0.168
   2338 192.168.0.170 192.168.0.168
   2339 192.168.0.179 192.168.0.168
   2339 192.168.0.18 192.168.0.168
   2341 192.168.0.232 192.168.0.168
   2337 192.168.0.244 192.168.0.168
   2341 192.168.0.5 192.168.0.168
   2338 192.168.0.58 192.168.0.168
   2339 192.168.0.59 192.168.0.168
   2340 192.168.0.68 192.168.0.168
```

This shows that there are about 14 different machines who have established around 2300 connections each with 192.168.0.168 and if you look at the command closely, we have filtered out results only for port 443.

Enough with finding the problem already. Let's dive straight into finding the solution(s) to this problem.

What's the way out?

Well don't be afraid because sysctl just happens to be a friendly monster. There are many ways by which we can solve this problem.

Approach #1

One of the most practical approaches to solve this problem and one that you most likely will or rather should end up doing is to increase the

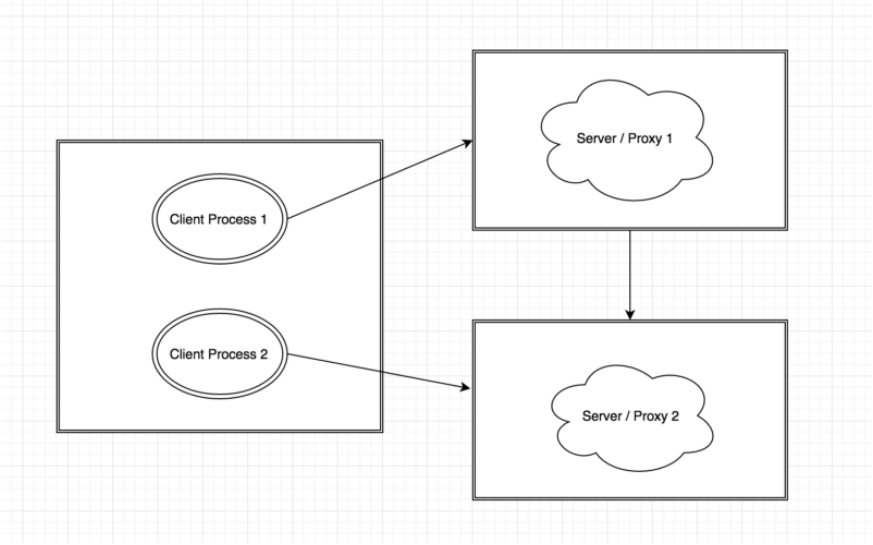
local ephemeral port range to the maximum possible value. As mentioned before, the default range is very small.

```
echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

This will increase the local port range to a bigger value. We cannot increase the range beyond this as there can only be a maximum of 65535 ports and the first 1024 are reserved for select services and purposes.

Note that you might still get bottleneck on this issue. However, instead of 28000 ports being used locally, it will be 64000 ports. Not a full proof solution but this is something that you can do to give you some breathing room.

Does this mean I can only get about 64k concurrent connections from a single client machine? The answer is NO.



In this scenario, a single client machine will be able to generate about 120k concurrent connections because both the processes are connecting to two different backend servers or proxies and hence different destination IPs.

Approach #2

Another simple solution is to enable a Linux TCP option called ***tcp_tw_reuse.*** This option enables the Linux kernel to reclaim a connection slot from a connection in TIME_WAIT state and reallocate it to a new connection.

```
--> vim /etc/sysctl.conf
```

```
--> Add the following line in the end
# Allow reuse of sockets in TIME_WAIT state for new
connections
# only when it is safe from the network stack's perspective.
net.ipv4.tcp_tw_reuse = 1
```

```
--> Reload sysctl settings
sysctl -p
```

Approach #3

Use more server ports. Till now we have talked about port exhaustion problems arising because in the quadruplet logic discussed before, the destination Ip, destination port and source Ip remain constant. The only thing that changes is the client ports.

However, if the server listens on two different ports instead of one, then we have twice the number of ephemeral ports available instead of one. This clubbed with the first approach gives you about 120k concurrent connections on a single machine.

You have to however take care that running the server on two ports—which essentially means running two servers on the same machine—does not have a huge impact on the hardware.

Approach #4

In a real production scenario, you may have millions of concurrent users simultaneously hitting the system. But in a load testing scenario, these users are to be artificially generated by a client running on a machine.

Here again the 65k port limit comes to bite on the client side. The only way to overcome this from the client's perspective is to increase the number of client machines that are generating the load. As you will read the next part in this series you will find that we had to use about 14 different machines to generate the kind of load we wanted to test HAProxy.

Putting it all together

There isn't one single configuration that will solve all your woes and work like a charm. It is always the combination of multiple things that work out in the end.

For us as a prerequisite to load testing HAProxy, we followed approach #1 and approach #2 and eventually approach #3 to generate a huge… huge load of **2 million concurrent connections** on a single HAProxy machine.

Here's the final part of this series, where I'll put together all the components that went into generating this kind of load, the tunings we did and the learnings that came out of it.

Do let me know how this blog post helped you and stay tuned for the final part in this series of posts. Also, please recommend (❤) this post if you think this may be useful for someone.