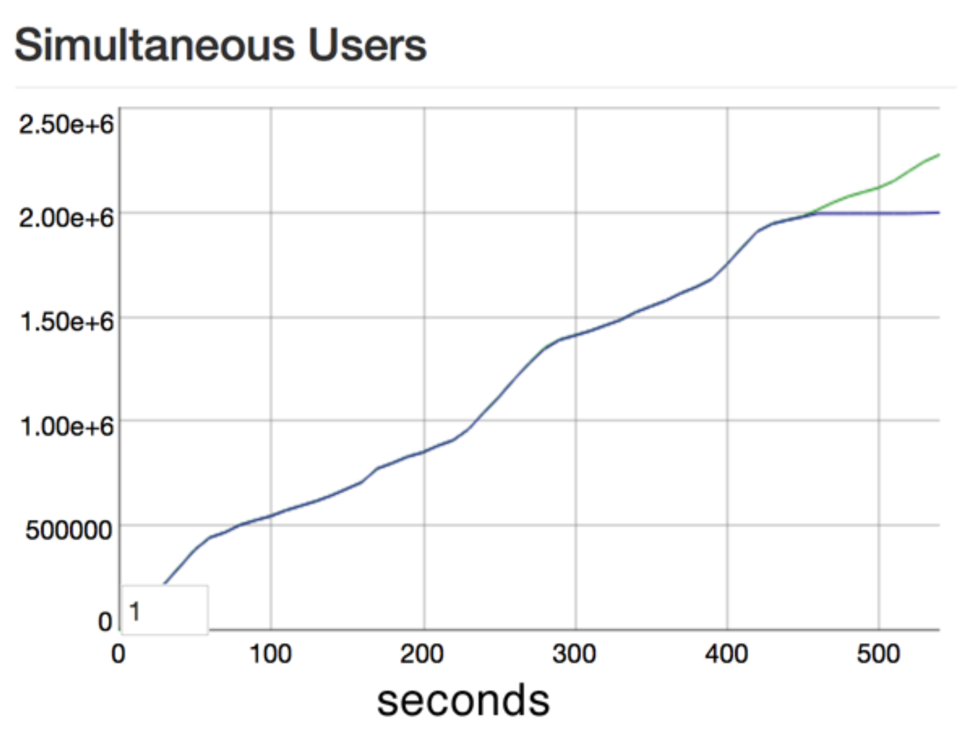
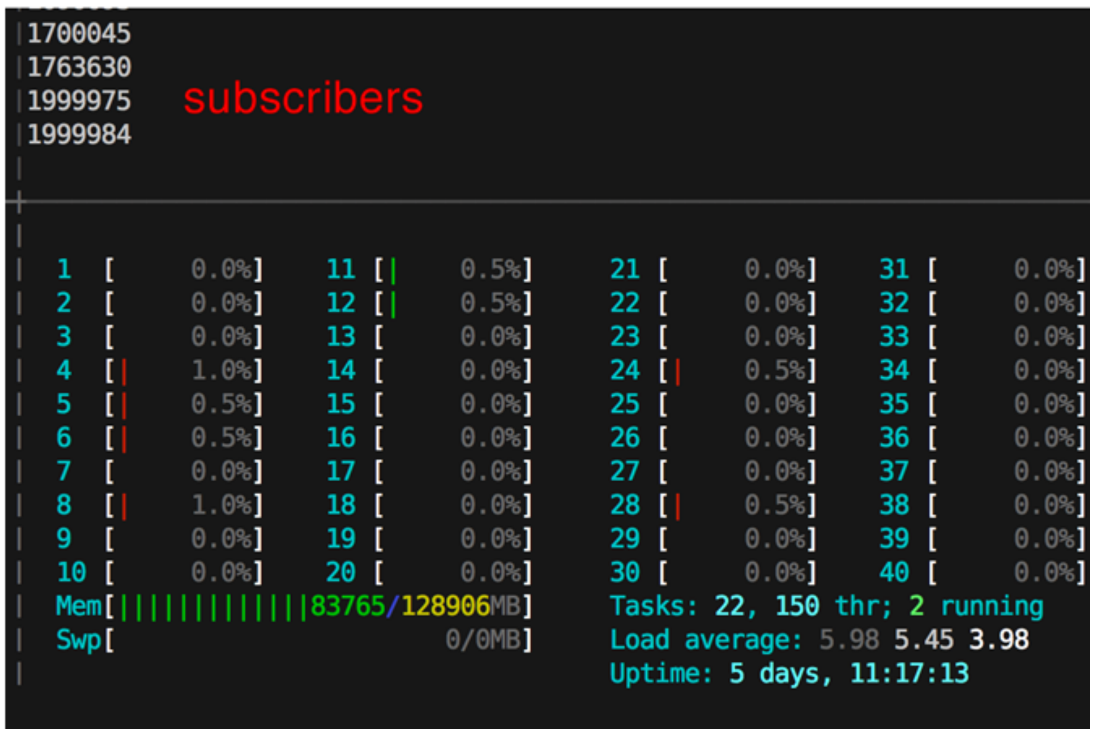
The Road to 2 Million Websocket Connections in Phoenix

Posted on November 3rd, 2015 by Gary Rennie





```
1700045
1763630
1999975   subscribers
1999984

 1  [      0.0%]   11 [|     0.5%]   21 [      0.0%]   31 [      0.0%]
 2  [      0.0%]   12 [|     0.5%]   22 [      0.0%]   32 [      0.0%]
 3  [      0.0%]   13 [      0.0%]   23 [      0.0%]   33 [      0.0%]
 4  [|     1.0%]   14 [      0.0%]   24 [|     0.5%]   34 [      0.0%]
 5  [|     0.5%]   15 [      0.0%]   25 [      0.0%]   35 [      0.0%]
 6  [|     0.5%]   16 [      0.0%]   26 [      0.0%]   36 [      0.0%]
 7  [      0.0%]   17 [      0.0%]   27 [      0.0%]   37 [      0.0%]
 8  [|     1.0%]   18 [      0.0%]   28 [|     0.5%]   38 [      0.0%]
 9  [      0.0%]   19 [      0.0%]   29 [      0.0%]   39 [      0.0%]
 10 [      0.0%]   20 [      0.0%]   30 [      0.0%]   40 [      0.0%]
 Mem[|||||||||||||83765/128906MB]   Tasks: 22, 150 thr; 2 running
 Swp[                    0/0MB]   Load average: 5.98 5.45 3.98
                                  Uptime: 5 days, 11:17:13
```

If you have been paying attention on Twitter recently, you have likely seen some increasing numbers regarding the number of simultaneous connections the Phoenix web framework can handle. This post documents some of the techniques used to perform the benchmarks.

How It Started

A couple of weeks ago I was trying to benchmark the number of connections and managed to get 1k connections on my local machine. I wasn't convinced by the number so I posted in IRC to see if anyone had benchmarked Phoenix channels. It turned out they had not, but some members of the core team found the 1k number I provided suspiciously low. This was the beginning of the journey.

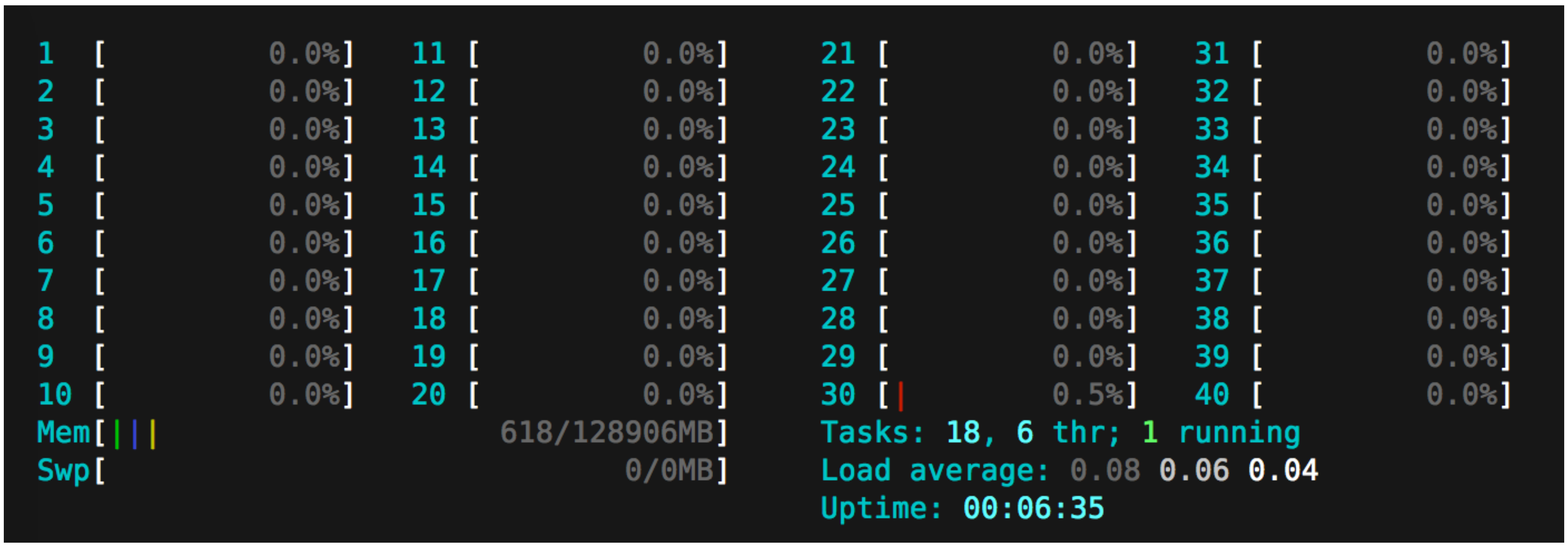
How To Run The Benchmarks

The Server

To benchmark the number of simultaneous web sockets that can be open at a time, the first thing required is a Phoenix application to accept the sockets. For these tests, we used a slightly modified version of the [chrismccord/phoenix_chat_application](https://github.com/chrismccord/phoenix_chat_application) available at [Gazler/phoenix_chat_example](https://github.com/Gazler/phoenix_chat_example) - the key difference is:

The `after_join` hook that broadcasts a user has joined a channel has been removed. When measuring concurrent connections, we want to limit the number of messages that are sent. There will be future benchmarks for that.

Most of these tests were performed on Rackspace 15 GB I/O v1 - these machines have 15GB RAM and 4 cores. Rackspace kindly let us use 3 of these servers for our benchmarks free of charge. They also let us use a OnMetal I/O which had 128GB RAM and showed 40 cores in htop.



```
 1  [          0.0%]   11 [          0.0%]   21 [          0.0%]   31 [          0.0%]
 2  [          0.0%]   12 [          0.0%]   22 [          0.0%]   32 [          0.0%]
 3  [          0.0%]   13 [          0.0%]   23 [          0.0%]   33 [          0.0%]
 4  [          0.0%]   14 [          0.0%]   24 [          0.0%]   34 [          0.0%]
 5  [          0.0%]   15 [          0.0%]   25 [          0.0%]   35 [          0.0%]
 6  [          0.0%]   16 [          0.0%]   26 [          0.0%]   36 [          0.0%]
 7  [          0.0%]   17 [          0.0%]   27 [          0.0%]   37 [          0.0%]
 8  [          0.0%]   18 [          0.0%]   28 [          0.0%]   38 [          0.0%]
 9  [          0.0%]   19 [          0.0%]   29 [          0.0%]   39 [          0.0%]
 10 [          0.0%]   20 [          0.0%]   30 [|         0.5%]   40 [          0.0%]
 Mem[|||                  618/128906MB]   Tasks: 18, 6 thr; 1 running
 Swp[                          0/0MB]   Load average: 0.08 0.06 0.04
                                        Uptime: 00:06:35
```

One additional change you may want to make is to remove `check_origin` in `conf/prod.exs` - this will mean that the application can be connected to regardless of the ip address/hostname used.

To start the server just `git clone` it and run:

```
MIX_ENV=prod mix deps.get
```

```
MIX_ENV=prod mix deps.compile
```

```
MIX_ENV=prod PORT=4000 mix phoenix.server
```

You can validate this is working by visiting `YOUR_IP_ADDRESS:4000`

The Client

For running the client, we used Tsung. Tsung is an open-source distributed load testing tool that makes it easy to stress test websockets (as well as many other protocols.)

The way Tsung works when distributing is by using host names. In our example, the first machine was called “phoenix1” which was assigned against the ip in `/etc/hosts`. The other machines “phoenix2” and “phoenix3” should also be in `/etc/hosts`.

It is important to run the clients on a different machine from the Phoenix application for the benchmarks. The results will not be a true representation if both are running on the same machine.

Tsung is configured using XML files. You can read about the particular values in the documentation. Here is the config file we used (however the numbers have been lowered to reflect the number of clients here, for our bigger tests we used 43 clients). It starts 1k connections a second up to a maximum of 100k connections. For each connection it opens a websocket, joins the “rooms:lobby” topic and then sleeps for 30000 seconds.

We used a large sleep time because we wanted to keep the connections open to see how responsive the application was after all the clients had connected. We would stop the test manually instead of closing the websockets in the config (Which you can do with `type="disconnect"`).

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/user/share/tsung/tsung-1.0.dtd">
<tsung loglevel="debug" version="1.0">
  <clients>
    <client host="phoenix1" cpu="4" use_controller_vm="false" maxusers="64000" />
    <client host="phoenix2" cpu="4" use_controller_vm="false" maxusers="64000" />
    <client host="phoenix3" cpu="4" use_controller_vm="false" maxusers="64000" />
  </clients>

  <servers>
    <server host="server_ip_address" port="4000" type="tcp" />
  </servers>

  <load>
    <arrivalphase phase="1" duration="100" unit="second">
      <users maxnumber="100000" arrivalrate="1000" unit="second" />
    </arrivalphase>
  </load>

  <options>
    <option name="ports_range" min="1025" max="65535"/>
  </options>

  <sessions>
    <session name="websocket" probability="100" type="ts_websocket">
      <request>
        <websocket type="connect" path="/socket/websocket"></websocket>
      </request>

      <request subst="true">
        <websocket type="message">{"topic":"rooms:lobby", "event":"phx_join", "payload":
{"user":"%%ts_user_server:get_unique_id%%"}, "ref":"1"}</websocket>
      </request>

      <for var="i" from="1" to="1000" incr="1">
        <thinktime value="30"/>
      </for>
    </session>
  </sessions>
</tsung>
```

The First 1k Connections

Tsung provides a web interface at port `8091` which can be used to monitor the status of the test. The only chart that we were really interested in for these tests was similtaneous users. So the first time I ran Tsung on my own was on my own machine with both Tsung and the Phoenix chat application running locally. When doing this Tsung would often crash - when this happens you can't see the web interface - which means there is no chart to show for this, but it was an unimpressive 1k connections.

The First 1k Connections Again!

I set up a machine remotely and attempted benchmarking again. This time I was getting 1k connections, but at least Tsung didn't crash. The reason for this was the system-wide resource limit was being reached. To verify this I ran `ulimit -n` which returned `1024` which would explain why I could only get 1k connections.

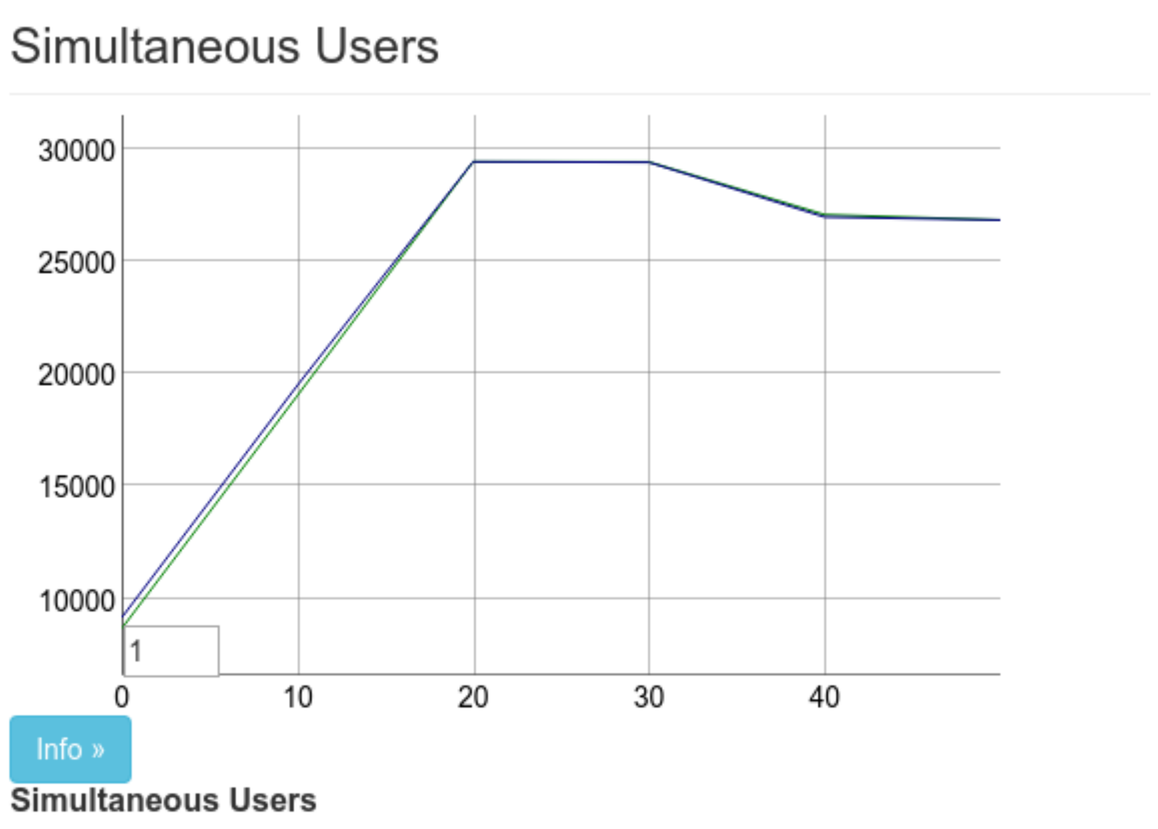
From this point onwards the following configuration was used. This configuration took us all the way to 2 million connections.

```
sysctl -w fs.file-max=12000500
sysctl -w fs.nr_open=20000500
ulimit -n 20000000
sysctl -w net.ipv4.tcp_mem='10000000 10000000 10000000'
sysctl -w net.ipv4.tcp_rmem='1024 4096 16384'
sysctl -w net.ipv4.tcp_wmem='1024 4096 16384'
sysctl -w net.core.rmem_max=16384
sysctl -w net.core.wmem_max=16384
```

The First Real Benchmark

I had been talking about Tsung in IRC when Chris McCord (creator of Phoenix) contacted me to let me know RackSpace had set up some instances for us to use for the benchmarks. We got to work setting up the 3 servers with the following config file: https://gist.github.com/Gazler/c539b7ef443a6ea5a182

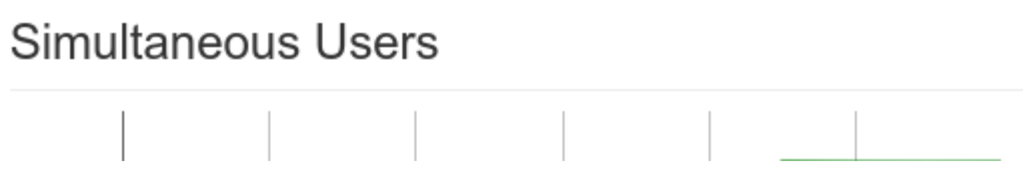
After we were up and running we dedicated one machine to Phoenix and two for running Tsung. Our first real benchmark ended up with about 27k connections.

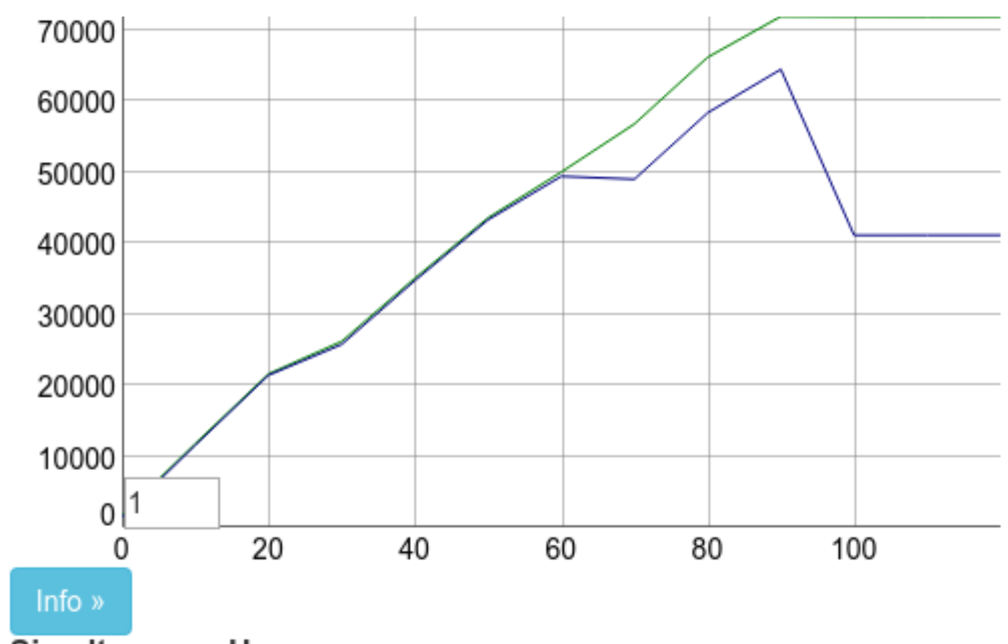


In the above image there are two lines on the chart, the line on the top is labeled "users" and the line on the bottom labeled "connected". The users increases based on arrival rate. For most of these tests we used an arrival rate of 1000 users per second.

As soon as the results were in, José Valim was on the case with this commit

This was our first improvement and it was a big one. From this we got up to about 50k connections.





Simultaneous Users

Observing the Changes

After our first improvement we realized that we were going in blind. If only there was some way we could observe what was happening. Luckily for use Erlang ships with observer and it can be used remotely. We used the following technique from https://gist.github.com/pnc/9e957e17d4f9c6c81294 to open a remote observer.

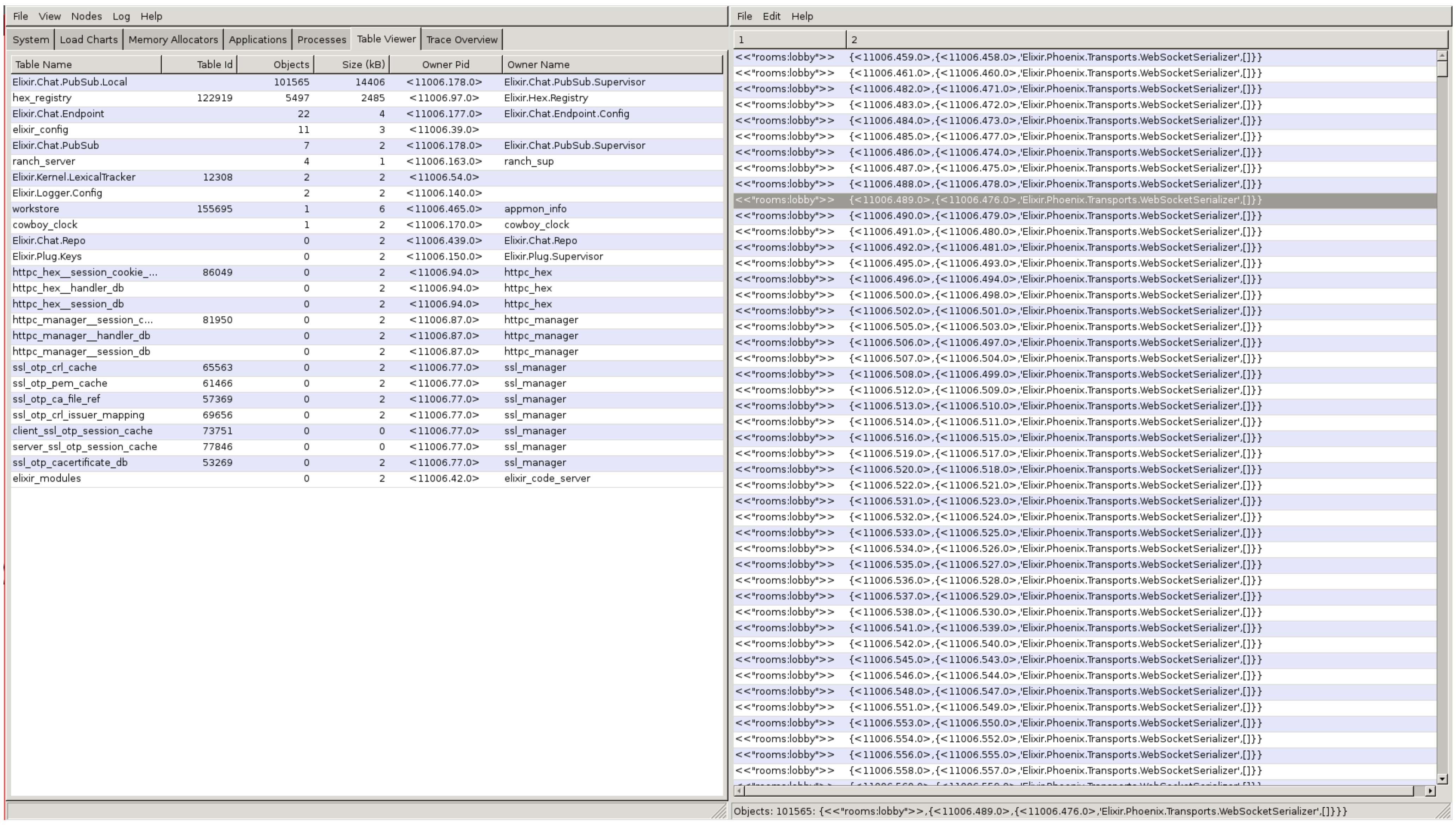

File View Nodes Log Help
System Load Charts Memory Allocators Applications Processes Table Viewer Trace Overview
File Edit Help

Table Name	Table Id	Objects	Size (kB)	Owner Pid	Owner Name
Elixir.Chat.PubSub.Local		101565	14406	<11006.178.0>	Elixir.Chat.PubSub.Supervisor
hex_registry	122919	5497	2485	<11006.97.0>	Elixir.Hex.Registry
Elixir.Chat.Endpoint		22	4	<11006.177.0>	Elixir.Chat.Endpoint.Config
elixir_config		11	3	<11006.39.0>	
Elixir.Chat.PubSub		7	2	<11006.178.0>	Elixir.Chat.PubSub.Supervisor
ranch_server		4	1	<11006.163.0>	ranch_sup
Elixir.Kernel.LexicalTracker	12308	2	2	<11006.54.0>	
Elixir.Logger.Config		2	2	<11006.140.0>	
workstore	155695	1	6	<11006.465.0>	appmon_info
cowboy_clock		1	2	<11006.170.0>	cowboy_clock
Elixir.Chat.Repo		0	2	<11006.439.0>	Elixir.Chat.Repo
Elixir.Plug.Keys		0	2	<11006.150.0>	Elixir.Plug.Supervisor
httpc_hex__session_cookie_...	86049	0	2	<11006.94.0>	httpc_hex
httpc_hex__handler_db		0	2	<11006.94.0>	httpc_hex
httpc_hex__session_db		0	2	<11006.94.0>	httpc_hex
httpc_manager__session_c...	81950	0	2	<11006.87.0>	httpc_manager
httpc_manager__handler_db		0	2	<11006.87.0>	httpc_manager
httpc_manager__session_db		0	2	<11006.87.0>	httpc_manager
ssl_otp_crl_cache	65563	0	2	<11006.77.0>	ssl_manager
ssl_otp_pem_cache	61466	0	2	<11006.77.0>	ssl_manager
ssl_otp_ca_file_ref	57369	0	2	<11006.77.0>	ssl_manager
ssl_otp_crl_issuer_mapping	69656	0	2	<11006.77.0>	ssl_manager
client_ssl_otp_session_cache	73751	0	0	<11006.77.0>	ssl_manager
server_ssl_otp_session_cache	77846	0	0	<11006.77.0>	ssl_manager
ssl_otp_cacertificate_db	53269	0	2	<11006.77.0>	ssl_manager
elixir_modules		0	2	<11006.42.0>	elixir_code_server

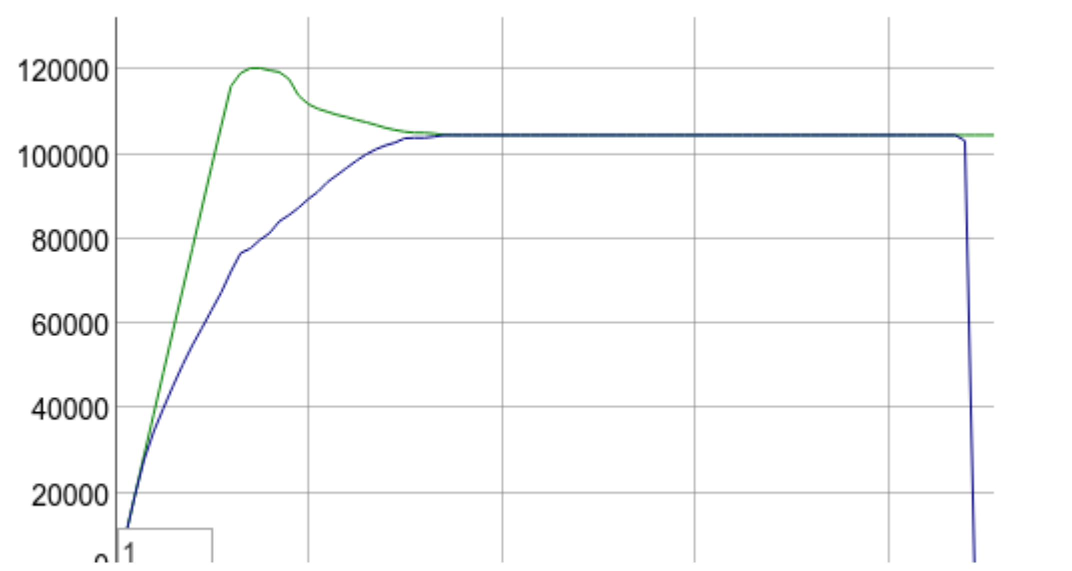
1	2
<<"rooms:lobby">>	{<11006.459.0>,{<11006.458.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.461.0>,{<11006.460.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.482.0>,{<11006.471.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.483.0>,{<11006.472.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.484.0>,{<11006.473.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.485.0>,{<11006.477.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.486.0>,{<11006.474.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.487.0>,{<11006.475.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.488.0>,{<11006.478.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.489.0>,{<11006.476.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.490.0>,{<11006.479.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.491.0>,{<11006.480.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.492.0>,{<11006.481.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.495.0>,{<11006.493.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.496.0>,{<11006.494.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.500.0>,{<11006.498.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.502.0>,{<11006.501.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.505.0>,{<11006.503.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.506.0>,{<11006.497.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.507.0>,{<11006.504.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.508.0>,{<11006.499.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.512.0>,{<11006.509.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.513.0>,{<11006.510.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.514.0>,{<11006.511.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.516.0>,{<11006.515.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.519.0>,{<11006.517.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.520.0>,{<11006.518.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.522.0>,{<11006.521.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.531.0>,{<11006.523.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.532.0>,{<11006.524.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.533.0>,{<11006.525.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.534.0>,{<11006.526.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.535.0>,{<11006.527.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.536.0>,{<11006.528.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.537.0>,{<11006.529.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.538.0>,{<11006.530.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.541.0>,{<11006.539.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.542.0>,{<11006.540.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.545.0>,{<11006.543.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.546.0>,{<11006.544.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.548.0>,{<11006.547.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.551.0>,{<11006.549.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.553.0>,{<11006.550.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.554.0>,{<11006.552.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.556.0>,{<11006.555.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}
<<"rooms:lobby">>	{<11006.558.0>,{<11006.557.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}

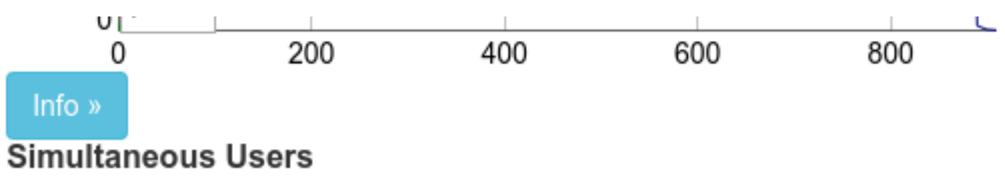
```
Objects: 101565: {<<"rooms:lobby">>,{<11006.489.0>,{<11006.476.0>,'Elixir.Phoenix.Transports.WebSocketSerializer',[]}}}
```

Chris was able to use the observer to order the processes by the size of their mailbox. The `:timer` process had about 40k messages in its mailbox. This is due to Phoenix doing a heartbeat every 30 seconds to ensure the client is still connected.

Luckily, Cowboy already takes care of this, so after this commit the results looked like:

Simultaneous Users





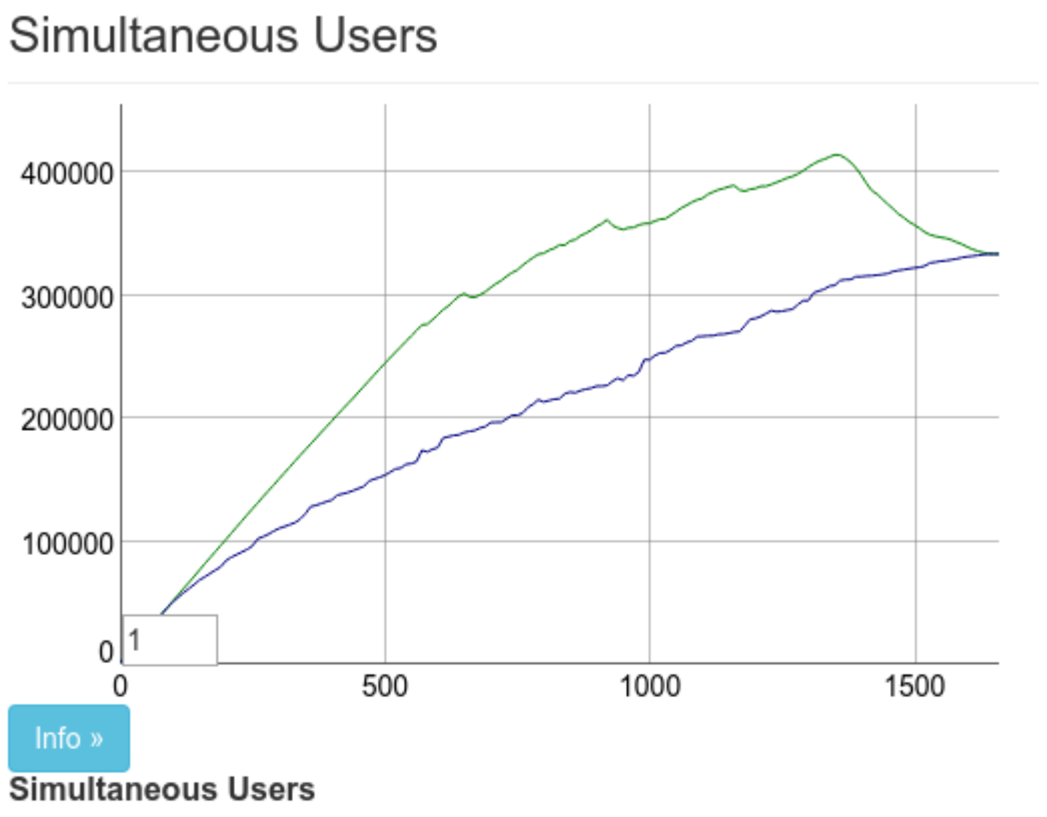
I actually killed the pubsub supervisor using observer in this image which explains the 100k drop at the end. This was the second 2x performance gain. The result was 100k concurrent connections using 2 Tsung machines.

We Need More Machines

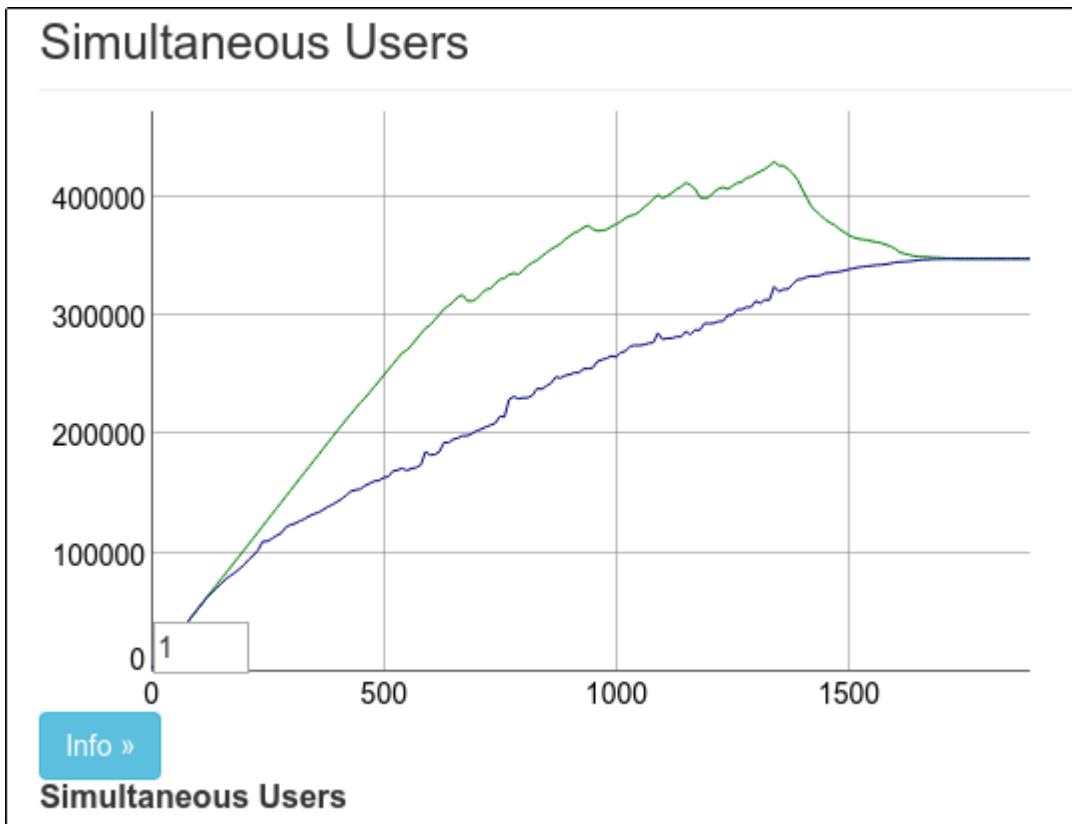
There are two problems with the image above. One is that we don't reach the full number of clients (about 15k were timing out) and two we can only actually generate between 40k and 60k connections per Tsung client (technically per IP address.) For Chris and I this wasn't good enough. We couldn't really see the limits unless we could generate more load.

At this stage RackSpace had given us the 128GB box, so we actually had another machine we could use, using such a powerful machine as a Tsung client limited to 60k connections may seem like a waste, but it's better than the machine idling! Chris and I set up another 5 boxes between us which is another 300k possible connections.

We ran the benchmarks again and we got about 330k connected clients.



The big problem is about 70k didn't actually connect to the machine. We couldn't work out why. Probably hardware issues. We decided to try running Phoenix on the 128GB machine instead. Surely there would be no issues reaching our connection limits, right?



Wrong. The results here are almost identical to those above. Chris and I thought 330k was pretty good. Chris tweeted out the results and we called it a night.

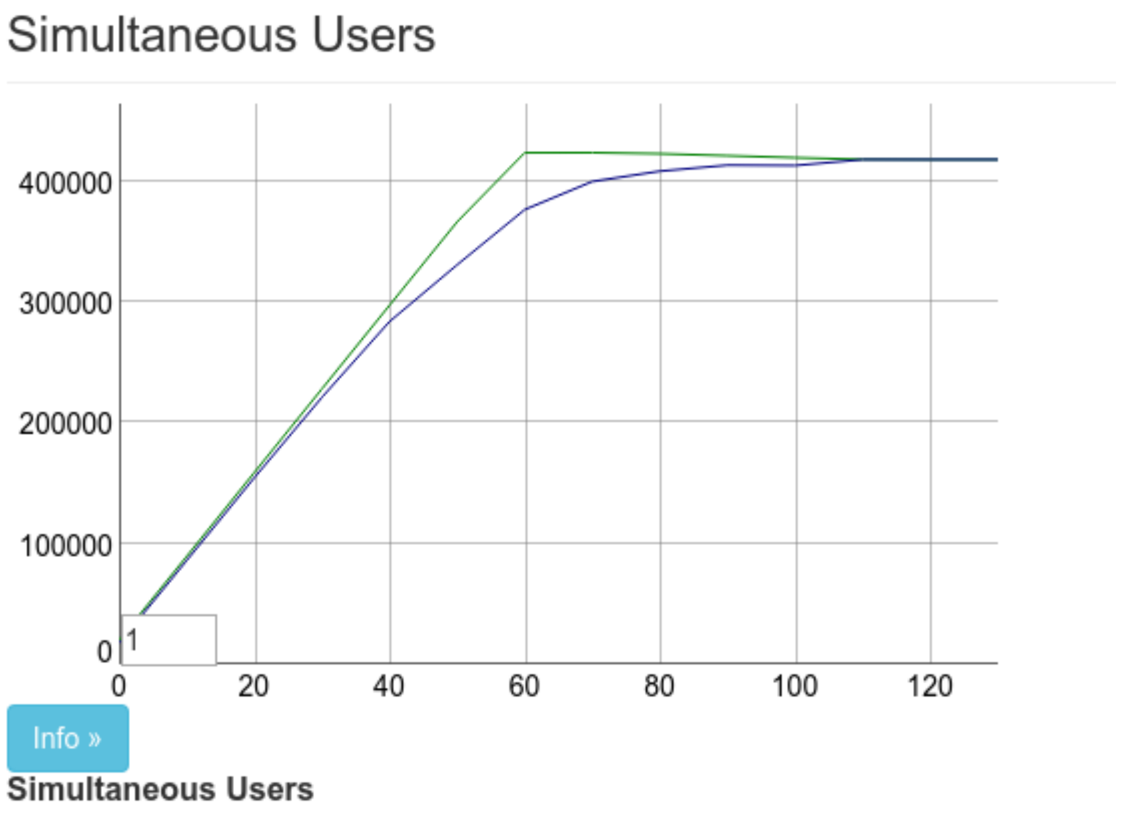
> Calling it quits trying to max Channels– at 333k clients. It took maxed ports on 8 servers to push that, 40% mem left. We're out of servers!
>
> — Chris McCord (@chris_mccord) October 24, 2015

Know Your ETS Types

After achieving 330k and having 2 fairly easy performance gains, we weren't sure there could be any more performance gains of the same magnitude. We were wrong. I wasn't aware of it at the time, but my colleague Gabi Zuniga ([@gabiz]) at [VoiceLayer] had been looking at the issue over the weekend. His commit gave us the best performance gain so far. You can see the diff on [the pull request]. I'll also provide it here for convenience:

```
-    ^local = :ets.new(local, [:bag, :named_table, :public,
+    ^local = :ets.new(local, [:duplicate_bag, :named_table, :public,
```

Those 10 additional characters made the chart look like this:



Not only did it increase the number of concurrent connections. It also allowed us to increase the arrival rate 10x too. Which made subsequent tests much faster.

The difference between `bag` and `duplicate_bag` is that `duplicate_bag` will allow multiple entries for the same key. Since each socket can only connect once and have one pid, using a duplicate bag didn't cause any issues for us.

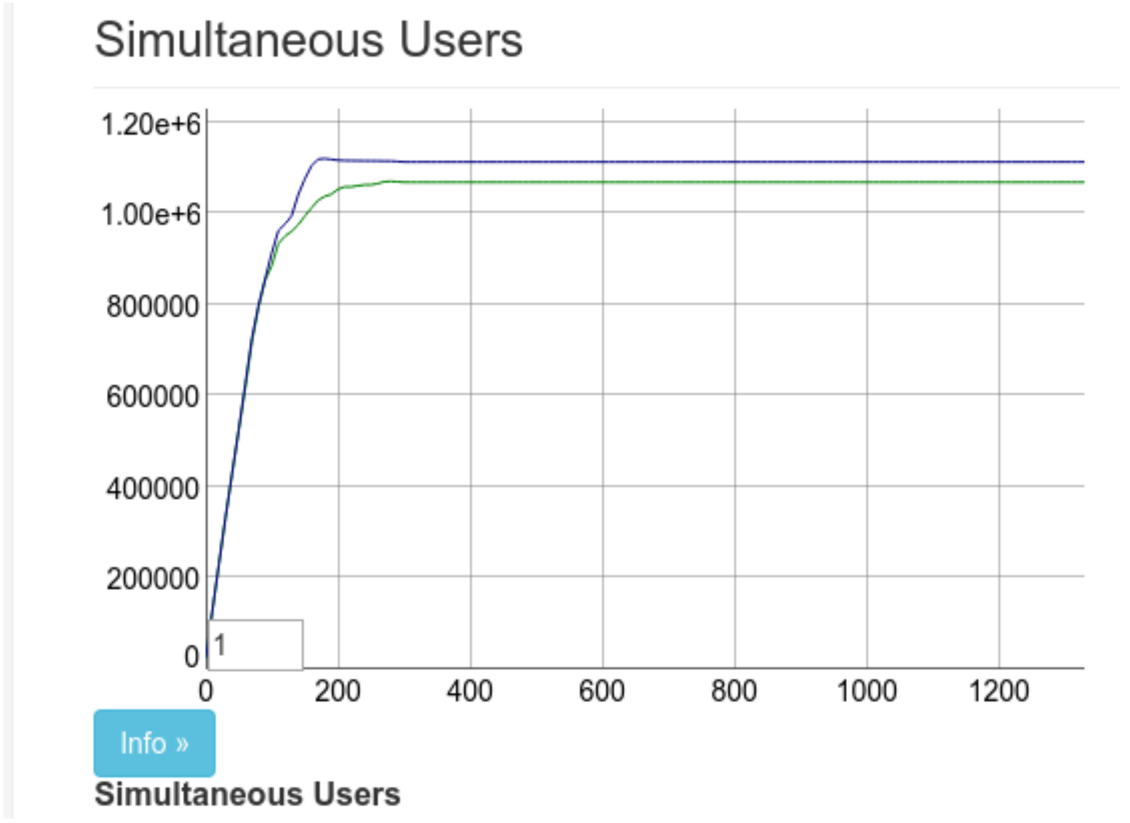
This maxed out at around 450k connections. At this point the 16GB box was out of memory. We were now ready to really test the larger box.

> I called it quits too early on the channel bench's, with an optimization by @gabiz , we have now maxed our 4core/15gb box @ 450k clients!
>
> — Chris McCord (@chris_mccord) October 25, 2015

We Need Even More Machines

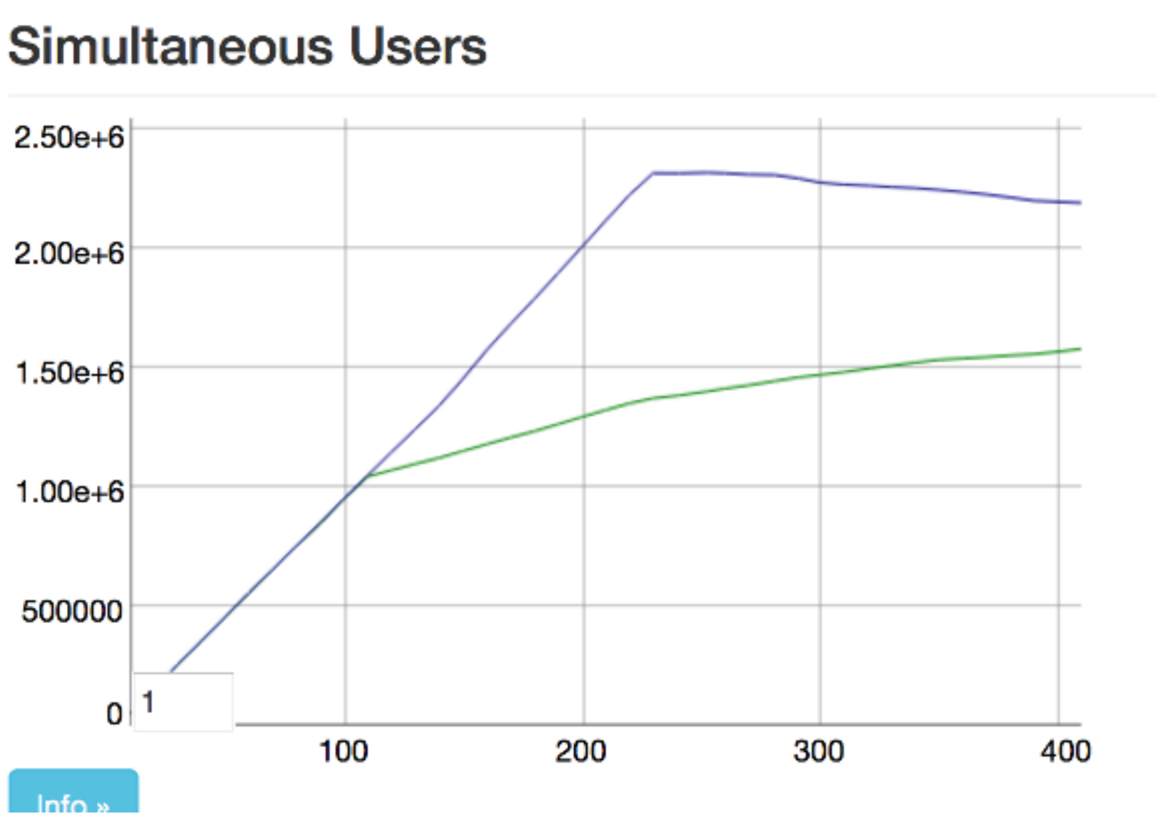
Justin Schneck ([@mobileoverlord]) informed us on IRC that he and his company [Live Help Now] would set up some additional servers on RackSpace for us to use. 45 additional servers to be precise.

We set up a few machines and set the threshold for Tsung to be 1 million connections. Which was a new milestone that was easily achieved by the 128GB machine:



> On a bigger @Rackspace box we just 1 million phoenix channel clients on a single server! Quick screencast in action: https://t.co/ONQcVWWdy1
>
> — Chris McCord (@chris_mccord) October 26, 2015

By the time Justin finished setting up all 45 boxes we were convinced 2 million connections was possible. Unfortunately that wasn't the case. There was a new bottleneck that only started appearing at 1.3 million connections!



That was it. 1.3M connections is good enough, right? Wrong. At the same time we hit 1.3M subscribers, we started getting regular timeouts when asking to subscribe to the single pubsub server. We also notice a large increase in broadcast time, taking over 5s to broadcast to all subscribers.

Justin is interested in Internet of (useful) Things, and wanted to see if we could optimize broadcasts for 1.3+M subscribers since he sees real use cases at these levels. He had the idea to shard broadcasts by chunking the subscribers and parellizing the broadcast work. We trialed this idea and it reduced the broadcast time back down to 1-2s. But, we still had those pesky subscribe timeouts. We were at the limits of a single pubsub server and single ets table. So Chris started work to pool the pubsub servers, and we realized we could combine Justin's broadcast sharding with a pool of pubsub servers and ets tables. So we sharded by subscriber pid into a pool of pubsub servers each managing their own ets table per shard. This let us to reach 2M subscribers without timeouts and maintain 1s broadcasts. The change is on this commit which is being refined before merging in to master.

> Final results from Phoenix channel benchmarks on 40core/128gb box. 2 million clients, limited by ulimit #elixirlang pic.twitter.com/6wRUIfFyKZ
>
> — Chris McCord (@chris_mccord) October 28, 2015

So there you have it. 2 million connections! Each time we thought there were no more optimizations to be made, another idea was pitched leading to a huge improvement in performance.

2 million is a figure we are pleased with. However, we did not quite max out the machine and we have not yet made any effort toward reducing the memory usage of each socket handler. In addition, there are more benchmarks we will be performing. This particular set of benchmarks was set exclusively around the number of simultaneous open sockets. A chat room with 2 million users is awesome, especially when the messages are broadcast so quickly. This is not a typical use case though. Here are some future benchmarking ideas:

One channel with x users sending y messages

X channels with 1000 users sending y messages

Running the Phoenix application across multiple nodes

A simulation sending a random number of messages with users arriving and leaving randomly to behave like a real chat room

The improvements discovered during this benchmark test will be made available in an upcoming release of Phoenix. Keep an eye out for information on future benchmark tests where Phoenix will continue to push the boundaries of the modern web.