

# Phoenix Channels vs Rails Action Cable

By: Chris McCord • 2016年8月9日

Ruby

Elixir

At DockYard, we transitioned our backend development from Ruby and Rails to Elixir and Phoenix once it became clear that Phoenix better served our clients needs to take on the modern web. As we've seen, Phoenix is Not Rails, but we borrow some of their great ideas. We were also delighted to give back in the other direction when Rails announced that Rails 5.0 would be shipping with Action Cable – a feature that takes inspiration from Phoenix Channels.

Now that Rails 5.0 has been released, we have clients with existing Rails stacks asking if they should use Action Cable for their real-time features, or jump to Phoenix for more reliable and scalable applications. When planning architecture decisions, in any language or framework, you should always take measurements to prove out your assumptions and needs. To measure how Phoenix channels and Action Cable handle typical application work-flows, we created a chat application in both frameworks and stressed each through varying levels of workloads.

## How the tests were designed

For our measurements, we used the tsung benchmarking client to open WebSocket connections to each application. We added XML configuration to send specific Phoenix and Rails protocol messages to open channels on the established connection and publish messages. For hardware, we used two Digital Ocean 16GB, 8-core instances for the client and server.

The work-flows for each tsung client connection were as follows:

- open a WebSocket connection to the server
- create a single channel subscription on the connection, to a chat room chosen at random

- Periodically send a message to the chat room, randomly once every 10s-60s to simulate messaging across members in the room

On the server, the channel code for Rails and Phoenix is quite simple:

```
1 | defmodule Chat.RoomChannel do
2 |   use Chat.Web, :channel
3 |
4 |   def join("room:" <> _id, _params, socket) do
5 |     {:ok, socket}
6 |   end
7 |
8 |   def handle_in("publish_msg", %{ "body" => body, "user" => user }, socket) do
9 |     broadcast!(socket, "new_message", %{body: body, user: user})
10 |    {:reply, :ok, socket}
11 |   end
12 | end
```

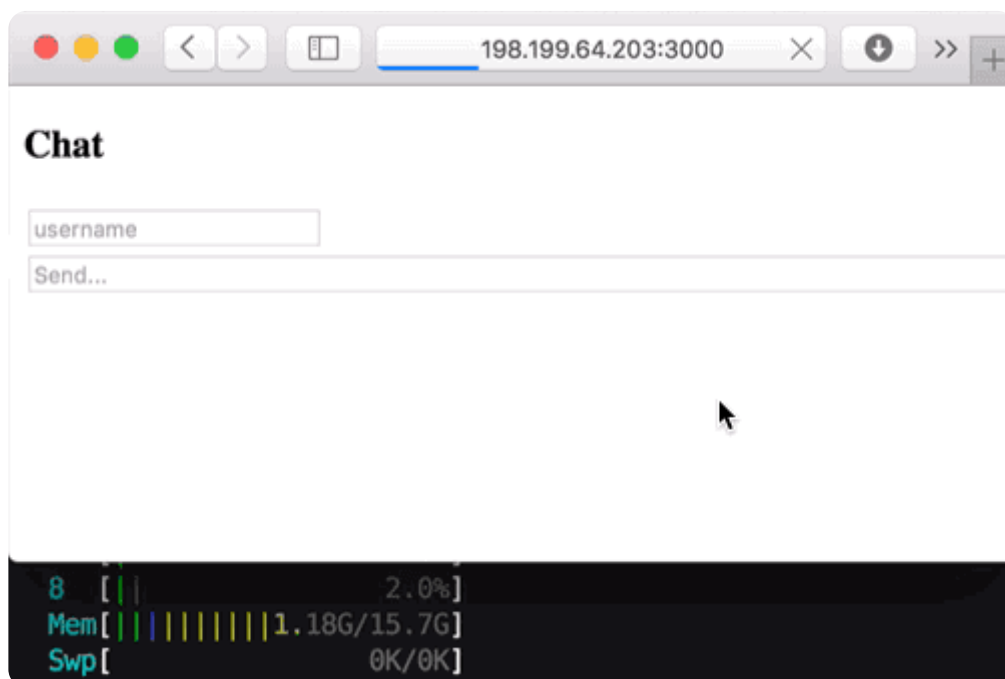
```
1 | class RoomsChannel < ApplicationCable::Channel
2 |   def subscribed
3 |     @topic = "rooms:#{rand(1..8)}"
4 |     stream_from @topic
5 |   end
6 |
7 |   def publish_msg(data)
8 |     ActionCable.server.broadcast(@topic,
9 |       body: data["body"],
10 |      username: data["username"],
11 |      started: data["started"]
12 |    )
13 |   end
14 | end
```

After establishing N numbers of rooms, with varying numbers of users per room, we measured each application's responsiveness. We tested performance by joining a random room from the browser and timing the broadcasts from our local browser to all members in the room. As we increased the numbers of users per room, we measured the change in broadcast latency. We recorded short clips of each application's behavior under different loads.

This simulates a “chat app”, but the work-flow applies equally to a variety of applications; real-time updates to visitors reading articles, streaming feeds, collaborative editors, and so on. As we evaluate the results, we’ll explore how the numbers relate to different kinds of applications.

## Rails Memory Leaks

After creating the Rails chat application, setting up redis, and deploying the application to our instance, we immediately observed a memory leak in the application that was visible just by refreshing a browser tab and watching the memory grow; to never be freed. The following recording shows this in action (sped up 10x):



We searched recently reported bugs around this area, and found an issue related to Action Cable [failing to call socket.close when cleaning up connections](#). This patch has been applied to the `5-0-stable` branch, so we updated the app to the unreleased branch and re-ran the tests. The memory leak persisted.

We haven’t yet isolated the source of the leak, but given the simplicity of the channel code, it must be within the Action Cable stack. This leak is particularly concerning since Rails 5.0 has been released for some time now and the `5-0-stable` branch itself has unreleased memory leak patches going back greater than 30 days.

## Scalability

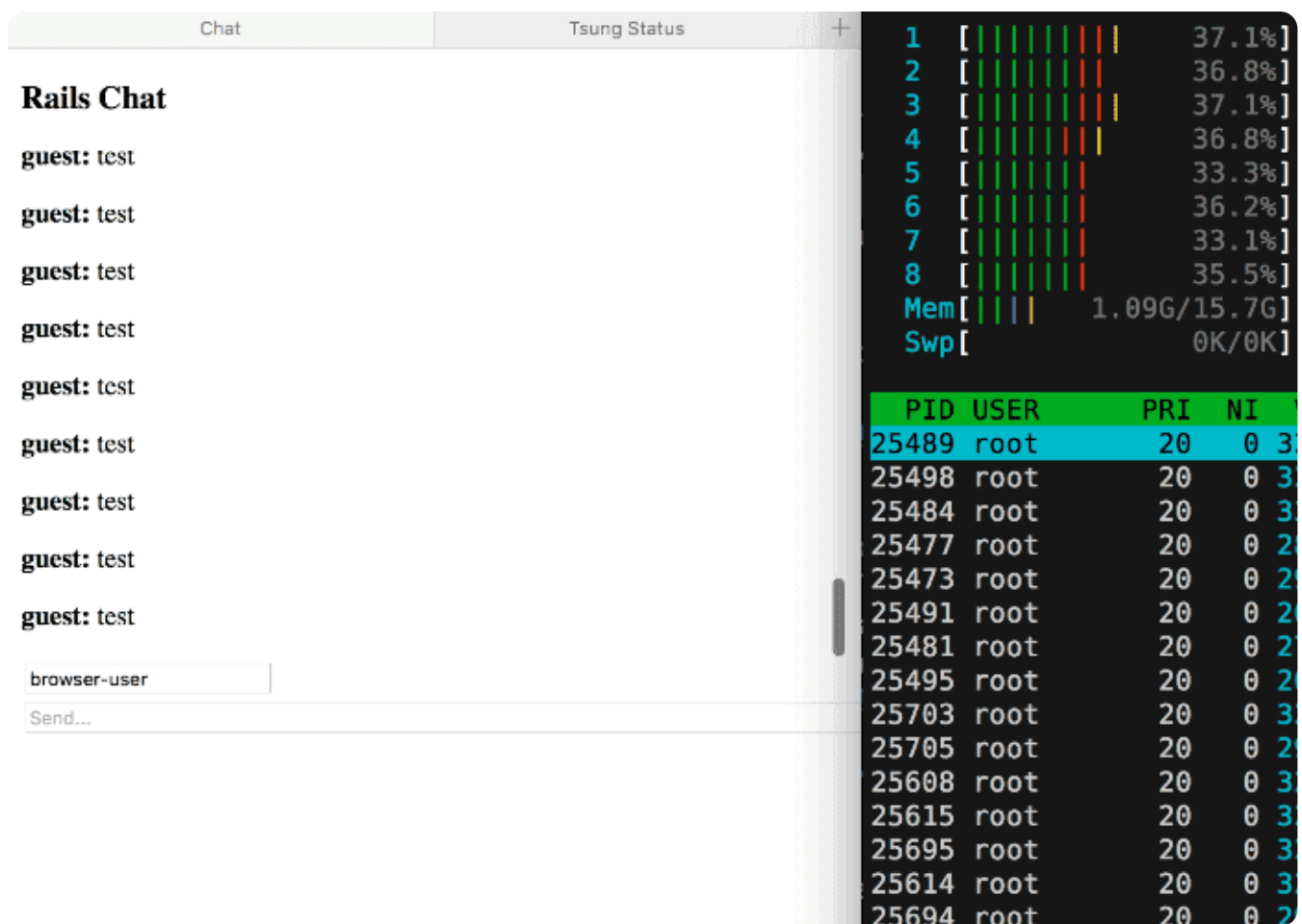
We set the memory leak issue aside and proceeded with our tests for the following scenarios:

- Max numbers of rooms supported by a single server, with 50 users per room
- Max numbers of rooms supported by a single server, with 200 users per room

*Note:* For Phoenix, for every scenario we maxed the *client server's* ability to open more WebSocket connections, giving us 55,000 users to simulate for our tests. Browser -> Server latency should also be considered when evaluating broadcast latency in these tests.

## 50 users per room

**Rails:** 50 rooms, 2500 users:



The screenshot displays a web application interface with two main panels. The left panel, titled 'Rails Chat', shows a chat window with multiple messages from 'guest: test'. The right panel, titled 'Tsung Status', displays system status metrics and a list of processes.

**System Status Metrics:**

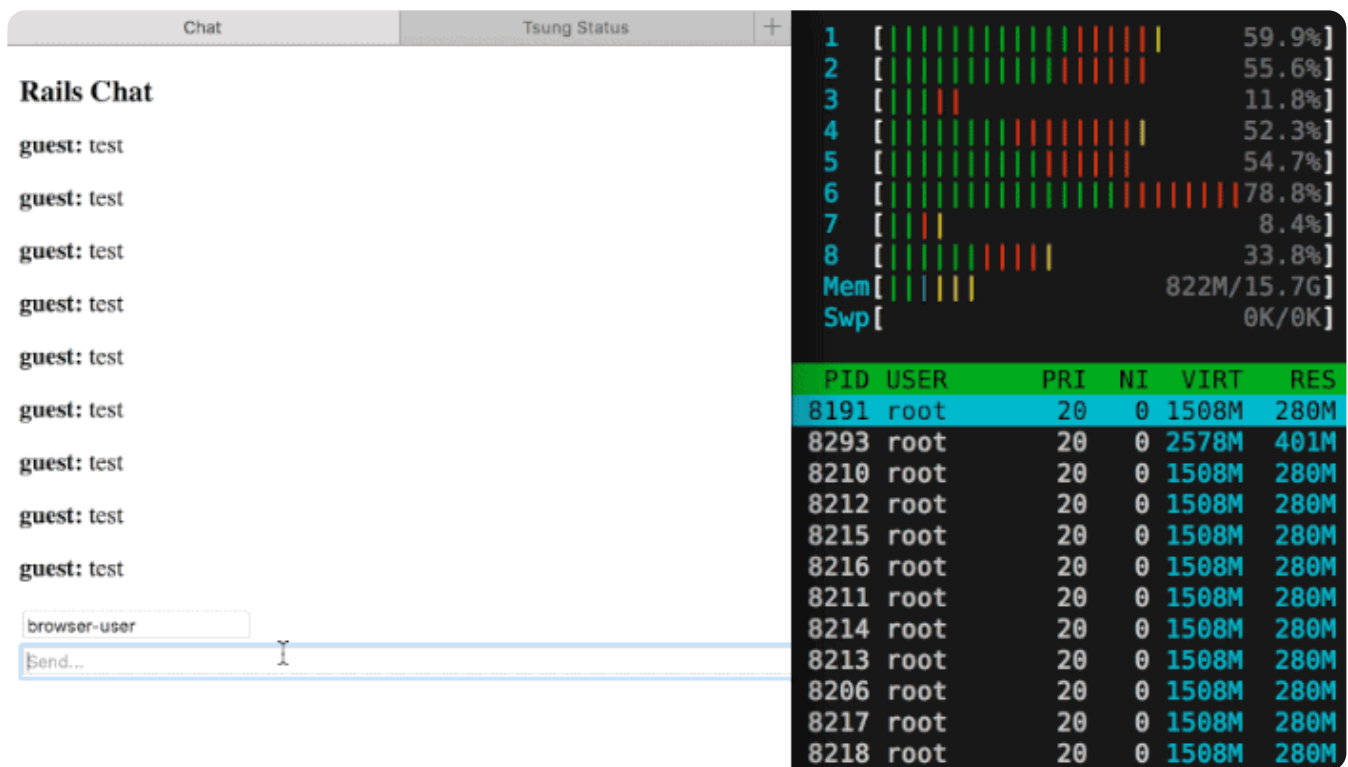
- 1 [|||||] 37.1%
- 2 [|||||] 36.8%
- 3 [|||||] 37.1%
- 4 [|||||] 36.8%
- 5 [|||||] 33.3%
- 6 [|||||] 36.2%
- 7 [|||||] 33.1%
- 8 [|||||] 35.5%
- Mem [|||||] 1.09G/15.7G
- Swp [|||||] 0K/0K

**Process List:**

| PID   | USER | PRI | NI |
|-------|------|-----|----|
| 25489 | root | 20  | 0  |
| 25498 | root | 20  | 0  |
| 25484 | root | 20  | 0  |
| 25477 | root | 20  | 0  |
| 25473 | root | 20  | 0  |
| 25491 | root | 20  | 0  |
| 25481 | root | 20  | 0  |
| 25495 | root | 20  | 0  |
| 25703 | root | 20  | 0  |
| 25705 | root | 20  | 0  |
| 25608 | root | 20  | 0  |
| 25615 | root | 20  | 0  |
| 25695 | root | 20  | 0  |
| 25614 | root | 20  | 0  |
| 25694 | root | 20  | 0  |

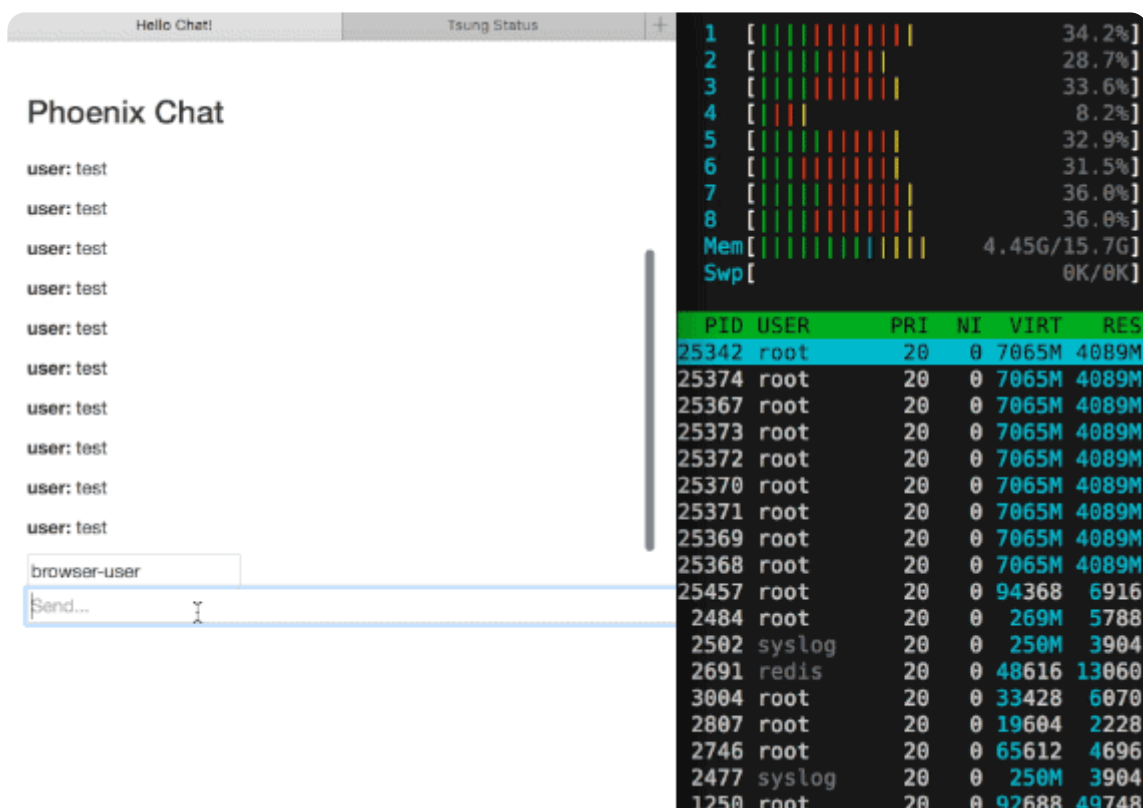
Responsiveness was speedy at 50 rooms, so we upped the room count to 75, giving us 3750 users.

**Rails:** 75 rooms, 3750 users:



Here, we can see Action Cable falling behind on availability when broadcasting, with messages taking an average of 8s to be broadcast to all 50 users for the given room. For most applications, this level of latency is not acceptable, so the level of performance for maximum rooms on this server would be somewhere between 50 and 75 rooms, given 50 users per room.

**Phoenix:** 1100 rooms, 55,000 users (maxed 55,000 client connections)



We can see that Phoenix responds on average in 0.25s, and only is maxed at 1100 rooms because of the 55,000 client limit on the tsung box.

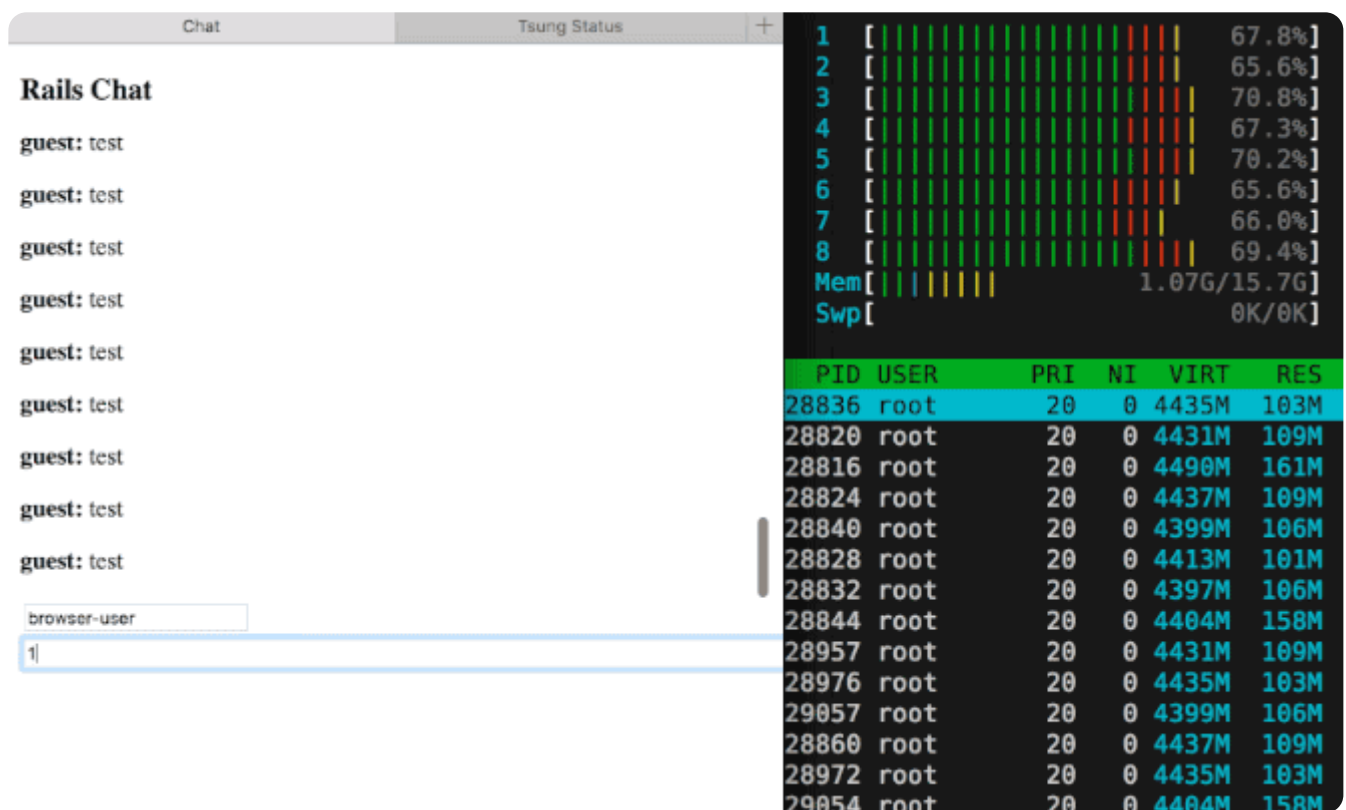
## Making sense of the results, 50 users per room

These results show that if you are building an application for small to medium sized businesses, if 75 companies visit the app at the same time, their notifications will be delayed. Given the severe degradation in performance from 50 to 75 rooms, it may also be hard to pinpoint your server density when planning for load.

For load planning, horizontal scalability will be required with 50-75 companies per server, but the reliance on redis should be considered as a central bottleneck. Horizontal scalability was the obvious choice for Ruby when it came to stateless HTTP requests, but we can see why vertical scalability becomes increasingly important for stateful connections. If you require dozens or hundreds of servers, Action Cable will still rely on a central redis bottleneck for PubSub.

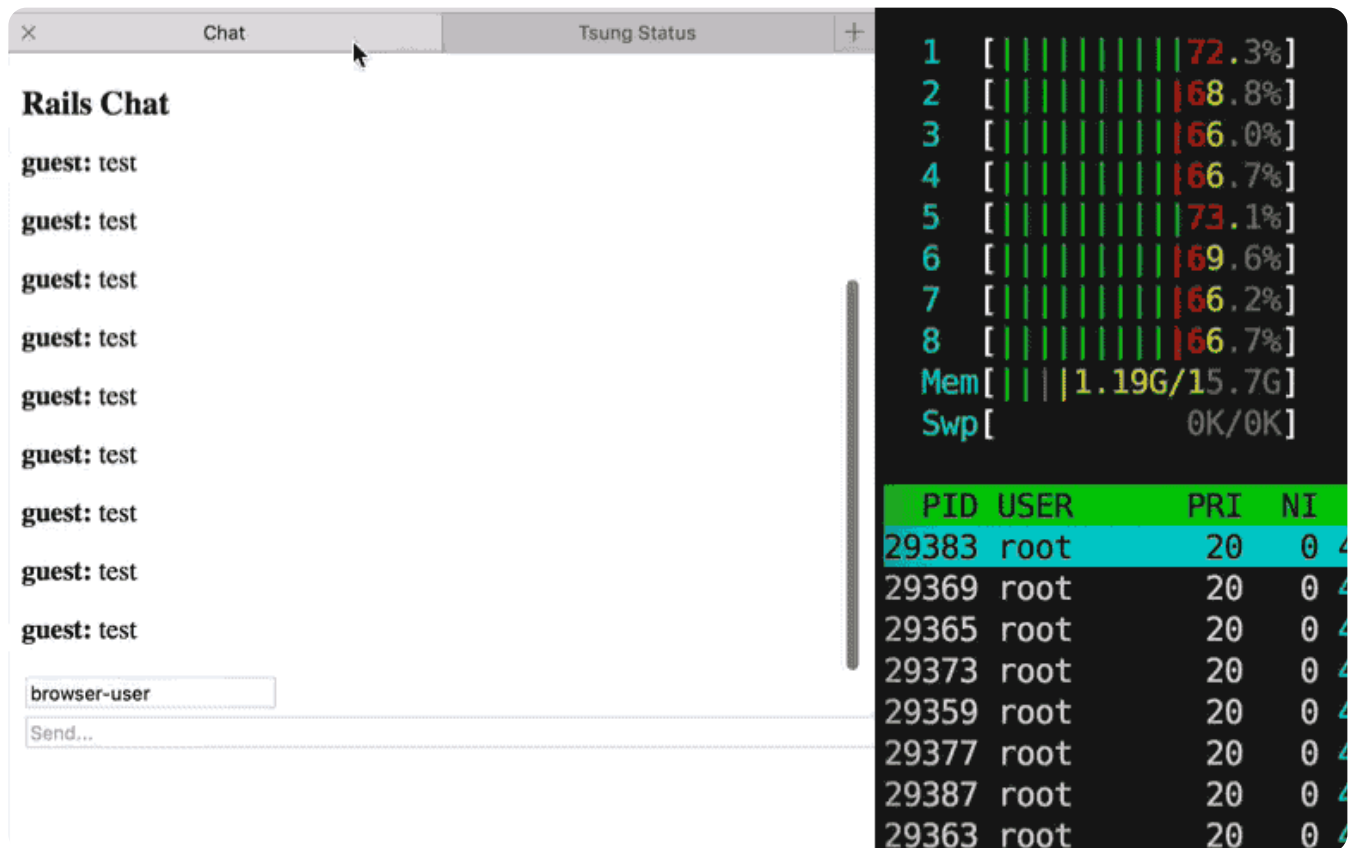
## 200 users per room

**Rails:** 8 rooms, 1600 users:



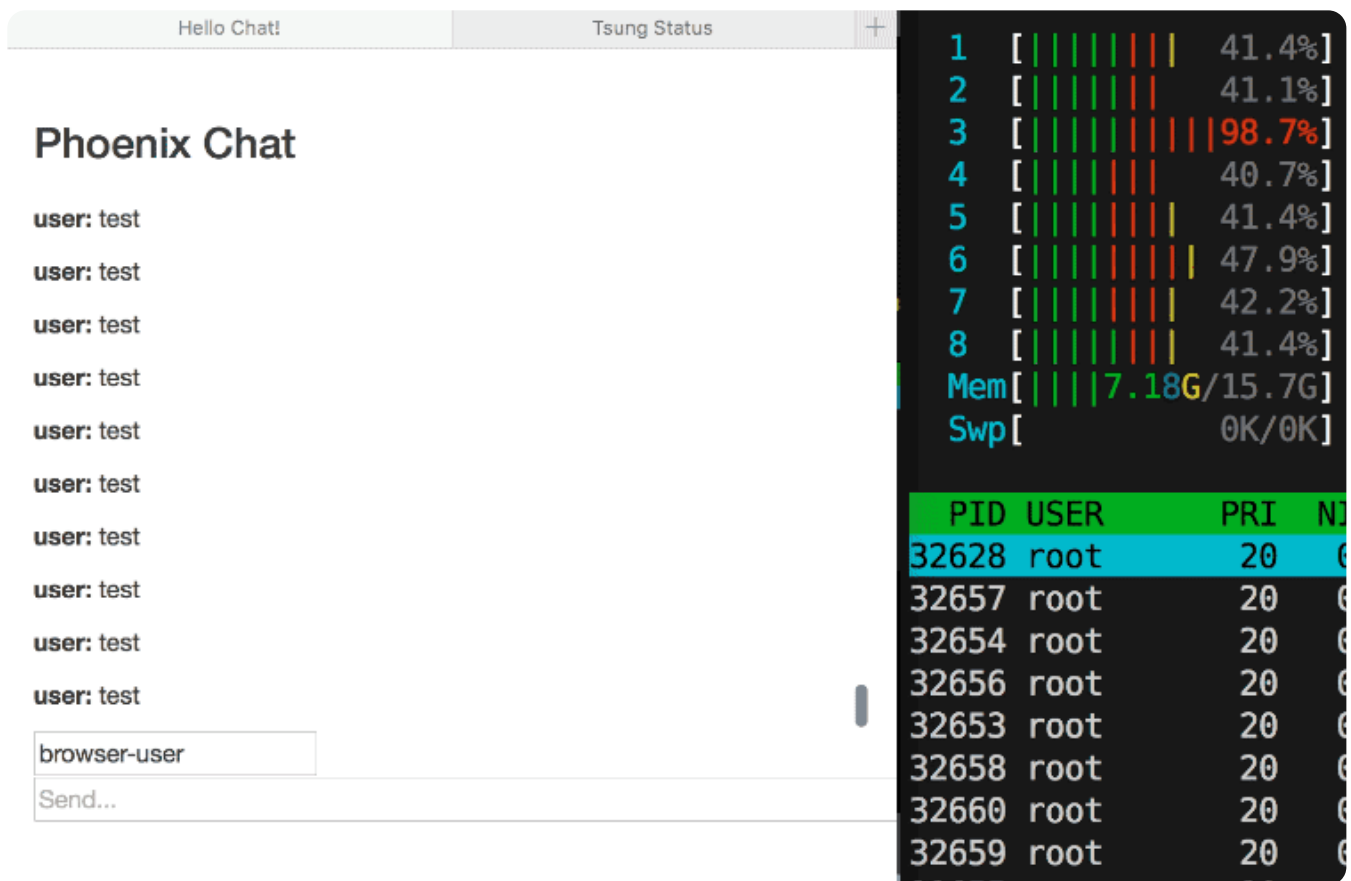
We can see Action Cable starts off with acceptable latency, but begins to quickly fall behind as broadcast latency grows with each message. Broadcast latency grows longer than 10s. Next, we upped the room count by 1, to see where the limit was.

**Rails:** 9 rooms, 1800 users before availability was compromised when broadcasting.



At 200 users per room, Action Cable is unable to maintain the broadcast load and supported just 9 rooms before we experienced messages stop arriving or subscriptions failing to establish. At these levels, the only consistent performance we could get for 200 users per room, was limiting the server to 7 rooms, or 1400 users.

**Phoenix:** 275 rooms, 55,000 users (maxed 55,000 client connections)



We can see that Phoenix responds on average in 0.24s, and only is maxed at 275 rooms because of the 55,000 client limit on the tsung box. Additionally, it's important to note that Phoenix maintains the same responsiveness when broadcasting for both the 50 and 200 users per room tests.

## Making sense of the results, 200 users per room

You may be thinking, "but my application isn't a chat app, and only needs simple page updates". These tests apply equally to many scenarios. Imagine you have an information system where you want to publish periodic updates to pages. This could be for a news site where visitors see new comments, or a booking site where visitors see "20 other people are currently viewing this hotel".

This tests shows that if your app receives a sudden spike in visitors and more than 7 articles have 200 or more readers, the server won't be able to keep up with the notification demand for both the popular articles, *as well as the low traffic ones*.

For a booking site, imagine 7 hotels across the country release a discount rate and customers jump online for the deal. Suddenly, you need to spin up extra servers to maintain booking notifications, and the severity of the delays becomes worse if critical functionality of your application relies on these notifications.



## Conclusions

If memory leaks can be ruled out or addressed, the sweet-spot for Action Cable today is small workloads with few subscribers on any given topic. Going beyond this means engineering efforts and resources must be spent on managing multiple nodes and optimizing code out of channels. Higher workloads with broadcasts to more than a few dozen subscribers risks availability. With Phoenix, we've shown that channels performance remains consistent as notification demand increases, which is essential for handling traffic spikes and avoiding overload.

When I started my own real-time Rails features with Sync several years ago, memory leaks and consistent performance were my main fears that drove me to look elsewhere, find Elixir, and create Phoenix. The rails-core team has done a great job putting a real-time story in place, but wrangling Ruby's lack of concurrency features will continue to be a challenge.

The source code and instructions for running these tests on your own hardware can be found here.