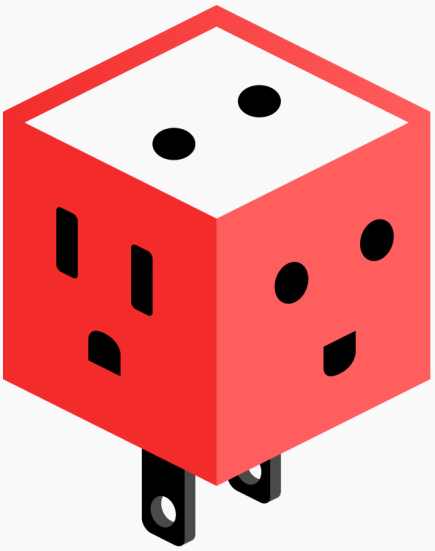


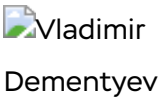
BACKEND

AnyCable:
Action Cable on steroids

December 20, 2016

Vladimir Dementyev

Backend Developer at Evil Martians



If you're interested in translating or adapting this post, please email us first.

Action Cable is one of the most attractive new features in Rails 5. WebSockets out of the box—sounds pretty cool, doesn't it?

> **New!** *The code examples in this article were updated in July 2020 to reflect the changes in AnyCable APIs and configuration. Check out the official documentation website for up-to-date information.*

Read also AnyCable 1.0: Four years of real-time web with Ruby and Go in our blog.

Action Cable shows plenty of good parts:

- Astonishingly easy configuration.
- Application logic access through channels—definitely the killer feature.
- JavaScript client that works with no additional effort.

Everything you need to build yet another Chat on Rails in minutes!

Now, to the troubling part. In my opinion, besides bugs, the only problem is Ruby itself—the language I'm fond of, but not considering it a technology fit for writing scalable concurrent applications. Not yet, at least (yeah, everyone heard about Ruby 3x3 and Matz's plans).

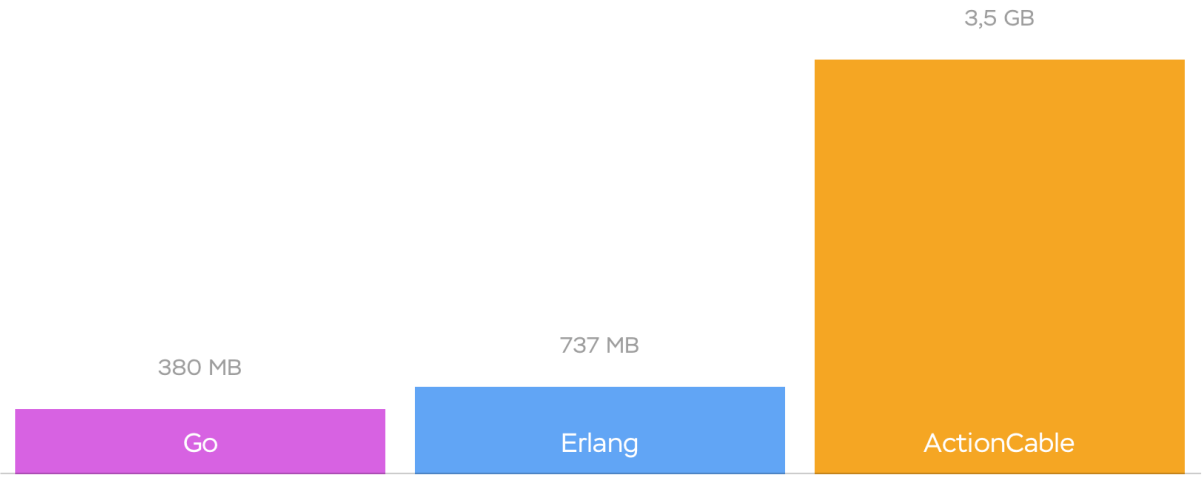
Benchmarks: Action Cable vs. Erlang vs. Golang

So what exactly is wrong with Action Cable? The answer is performance.

Let's consider several metrics: memory usage, CPU pressure, and broadcasting time. We built our benchmarks on top of the famous WebSocket Shootout by HashRocket.

We compared Action Cable with servers written in Erlang and Golang. The hardware we used to run the benchmarks was AWS EC2 `4.xlarge` (16 vCPU, 30Gib memory). Let's take a look at the results.

First, the RAM usage:



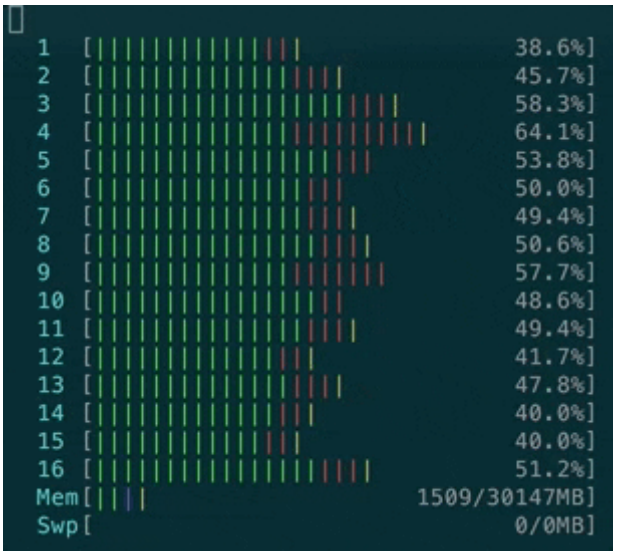
Benchmark: handle 20 thousand idle clients to the server (no subscriptions, no transmissions).

The results are pretty self-explaining: Action Cable requires much more memory. Because it's Ruby. And, of course, Rails.

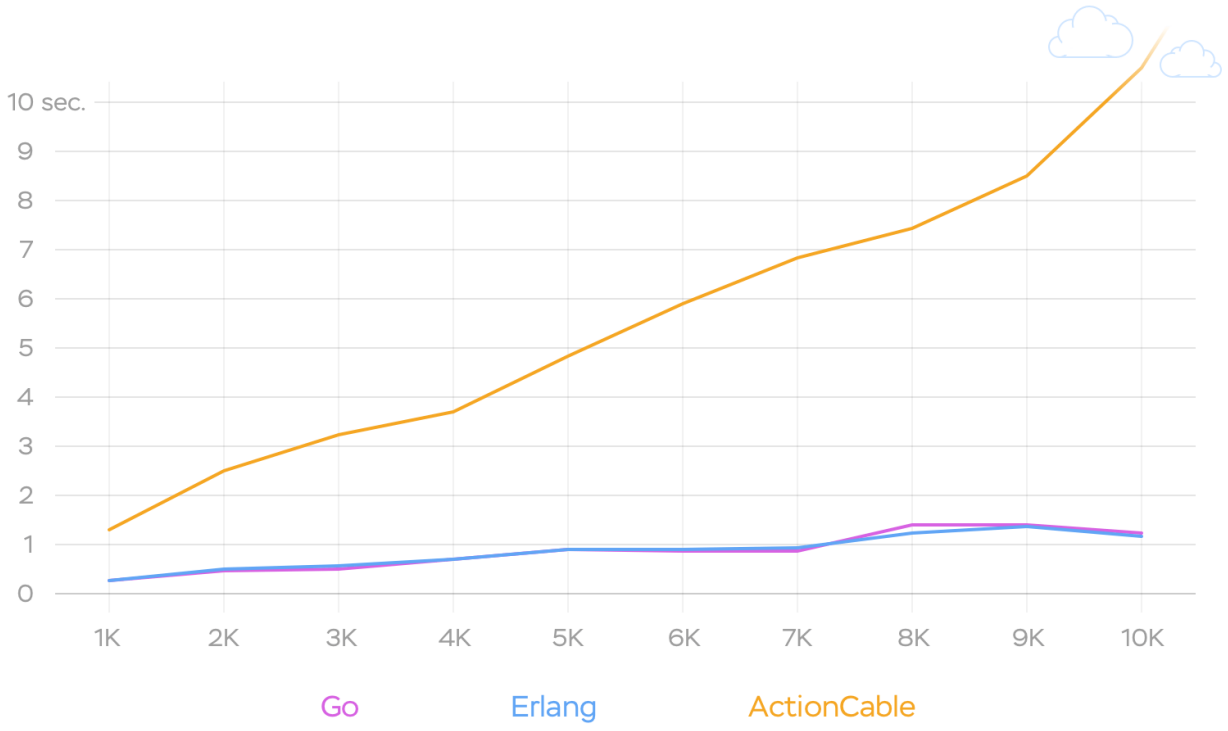
For the next two measurements we used *WebSocket Shootout* broadcasting benchmark which can be described as follows:

1000 clients connect, 40 of them send a message to a server, which is then re-transmitted to all clients. The sender-client measures the time it takes the message to make a round-trip (RTT).

Look how the CPU reacts during the benchmark:



And, last but not least, the broadcasting performance:



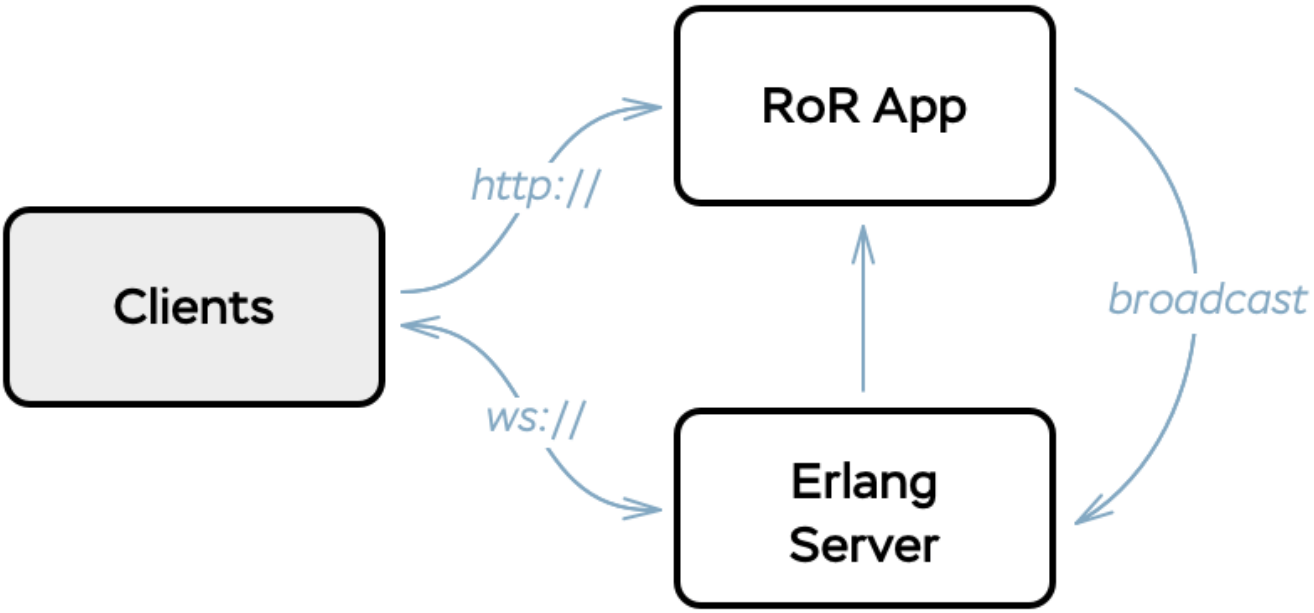
About a second for one thousand clients and more than ten seconds for ten thousand! Doesn't look *real-time*, does it?

But can we use the good parts of Action Cable with the power of Erlang/Golang/Whatever-you-like altogether? The answer is "Yes, we can". And here comes AnyCable.

The Idea

When @dhh announced Action Cable on April 2015, my initial thought was “Wow, this is Omakase I like!” The second thought was: “Oh, no, Ruby wasn’t meant for that.” And finally, the third thought: “I guess I know how to fix this!”

I’ve already had experience writing WebSocket-based applications in Erlang, and I knew that Erlang is one of the best players in this game, so I wrote down a simple architecture sketch:



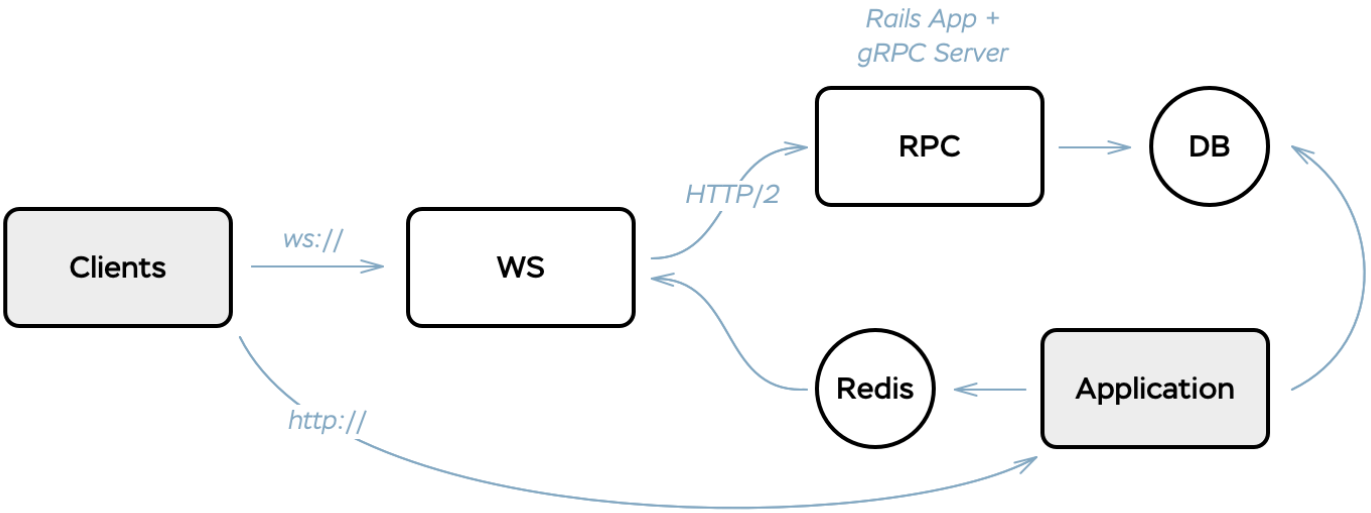
Looks pretty obvious except one thing: the communication between an Erlang server and a Rails app (Action Cable channels). The straightforward solution is using HTTP and REST. But don’t forget that we want to improve performance, so that’s not an option.

But aren’t we able to develop our own RPC protocol with raw TCP transport and binary encoding, maybe, Protocol Buffers? We sure are!

Luckily, I found a better option—gRPC. The puzzle was solved and AnyCable was born.

Introducing AnyCable

Take a look at the figure below:



AnyCable architecture.

AnyCable WebSocket server is responsible for handling clients, or *sockets*. That includes:

- Low-level connections (sockets) management.
- Subscriptions management.
- Broadcasting messages to clients.

The server doesn't know anything about your business logic. It's a kind of a **logicless proxy server**, just like NGINX for HTTP traffic.

WebSocket server should include **gRPC client** built from AnyCable `rpc.proto`, that describes the following RPC service:

```
service RPC {
  rpc Connect (ConnectionRequest) returns (ConnectionResponse) {}
  rpc Command (CommandMessage) returns (CommandResponse) {}
  rpc Disconnect (DisconnectRequest) returns (DisconnectResponse) {}
}
```

RPC server is a connector between Rails application and WebSocket server. It's an instance of your application with a gRPC endpoint which implements `rpc.proto`.

This server is a part of the `anycable` gem. Another gem, `anycable-rails`, makes it possible to use all Action Cable goodies (channels, streams) without any tweaks (see below for more information).

By default, we use Redis pub/sub to send messages from the application to the WS server, which then broadcasts the messages to clients. For test and development you can use

an alternative HTTP pub/sub implementation. Other adapters might get included in the future versions, if there is a demand from the community.

How AnyCable works?

After a brief look at the AnyCable architecture let's go deeper and see what's going on—from both a Rails application and a client point of view.

Consider a simple example—yet another Action Cable chat application. We want to implement the following features:

- Only authenticated users can join rooms.
- A user can join the *specific* chat room (using ID).
- A user can send a message to everyone in the room.

Each feature requires client-server communication. Let's discuss them one by one.

Authentication

So how does authentication happen in Action Cable? The same way we usually handle it in our controllers: with cookies.

Every time a client connects to Action Cable the application instantiates an instance of the `ApplicationCable::Connection` class and invokes the `#connected` method on it. The instance is added to the list of all connections and stays in memory until the client disconnects.

The connection allows you to access the underlying request object and, of course, its headers, including cookies:

```
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connected
      self.current_user = User.find_by(id: cookies[:user_id])
      reject_unauthorized_connection unless current_user
    end
  end
end
```

What about AnyCable? The connection itself occurs outside of the Rails application, on our WebSocket server. How can we authenticate it, or, more precisely, how to invoke our `#connected` method?

This is how AnyCable WebSocket server handles a client connection:

- Accepts a connection.
- Invokes the `Connect` RPC method and pass the request headers (some of them, e.g. `Cookie`) to it.
- Receives a response from an RPC server describing connection identifiers (JSON encoded) and status (whether it should be disconnected or not).
- Disconnects the client if necessary.
- Transmits messages to the client if the response object contains any.

And that is what happens within the `Connect` method realization on our RPC server:

- Create instance of the slightly patched `ApplicationCable::Connection` class.
- Invoke `#handle_open` on this connection.
- Reply with the connection information (identifiers) and status.

Note that AnyCable doesn't store any objects (e.g., connections) in memory. They're all disposable.

Subscriptions

Ok, now we're connected to the server. To make this connection useful we have to subscribe to a channel.

To do so, the client has to send a message in a specific form – `{"command": "subscribe", "identifier": "..."}`.

Consider our example chat channel code:

```
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:id]}"
  end
end
```

The client sends the message (JS code):

```
App.cable.subscriptions.create({channel: 'ChatChannel', id: 1}, ... )
```

AnyCable server transforms this message into a `Command` RPC call and then:

- Adds the client (socket) to the `chat_1` *broadcasting group*.
- Sends a subscription confirmation to the client.

What's a *broadcasting group*? Implementations may vary and depend on a language or platform you choose to build your WebSocket server. But to one degree or another it *quacks* like a hash that maps streams to connections.

Performing Actions

We're almost done. The only question remaining is, how do we send a message to the clients?

Let's add one more method to our channel:

```
class ChatChannel < ApplicationCable::Channel
  # ...

  def speak(data)
    ActionCable.server.broadcast(
      "chat_#{params[:id]}",
      text: data['text'], user_id: current_user.id
    )
  end
end
```

To perform an action the client should send another command – `{"command": "message", "identifier": "...}`. We can handle it the same way we do with the "subscribe" command.

The interesting thing here is the `#broadcast` call. Why? Because it's the weakest part of Action Cable (see broadcasting benchmark above).

In a simplified form, broadcasting can be implemented like this:

```
subscribers_map[stream_id].each do |channel|
  channel.transmit message
end
```

What's wrong with it? First, Rails itself. The second suspect is Ruby.

AnyCable moves this functionality out of Ruby and Rails and leverages the power of other technologies.

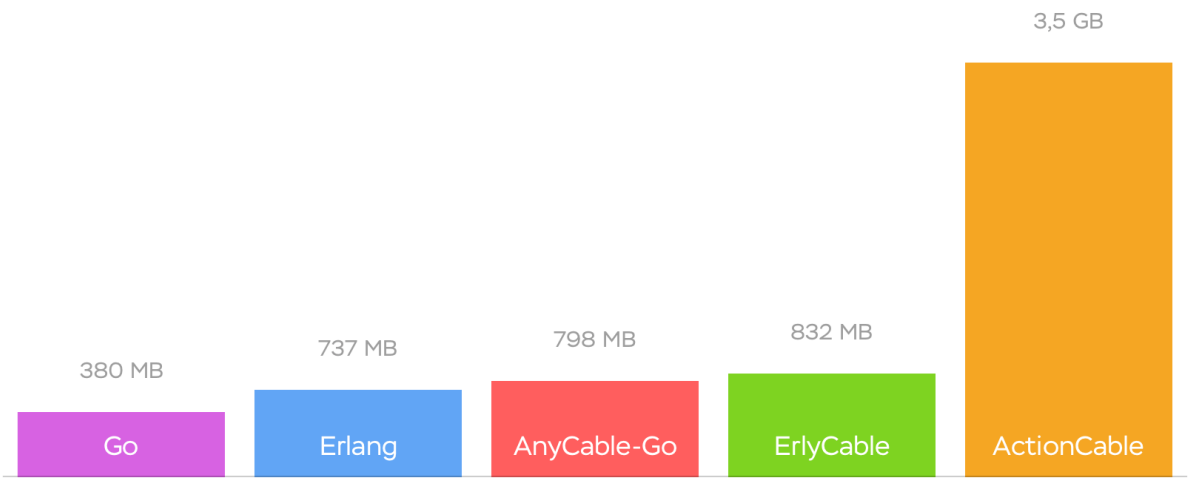
I think, it's time to repeat our benchmarks with the new kid on the block – AnyCable.

Benchmarks: AnyCable

> The latest benchmarks could be found in the main AnyCable repository on GitHub.

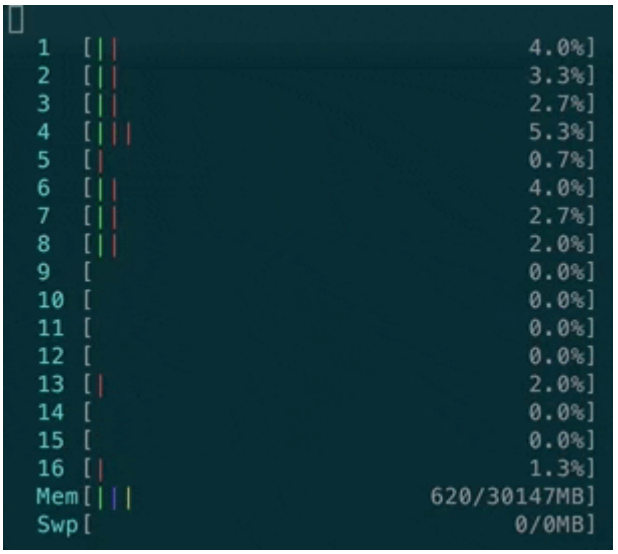
AnyCable shows almost the same resources usage as *raw* Erlang or Golang servers.

Memory usage is more than acceptable:

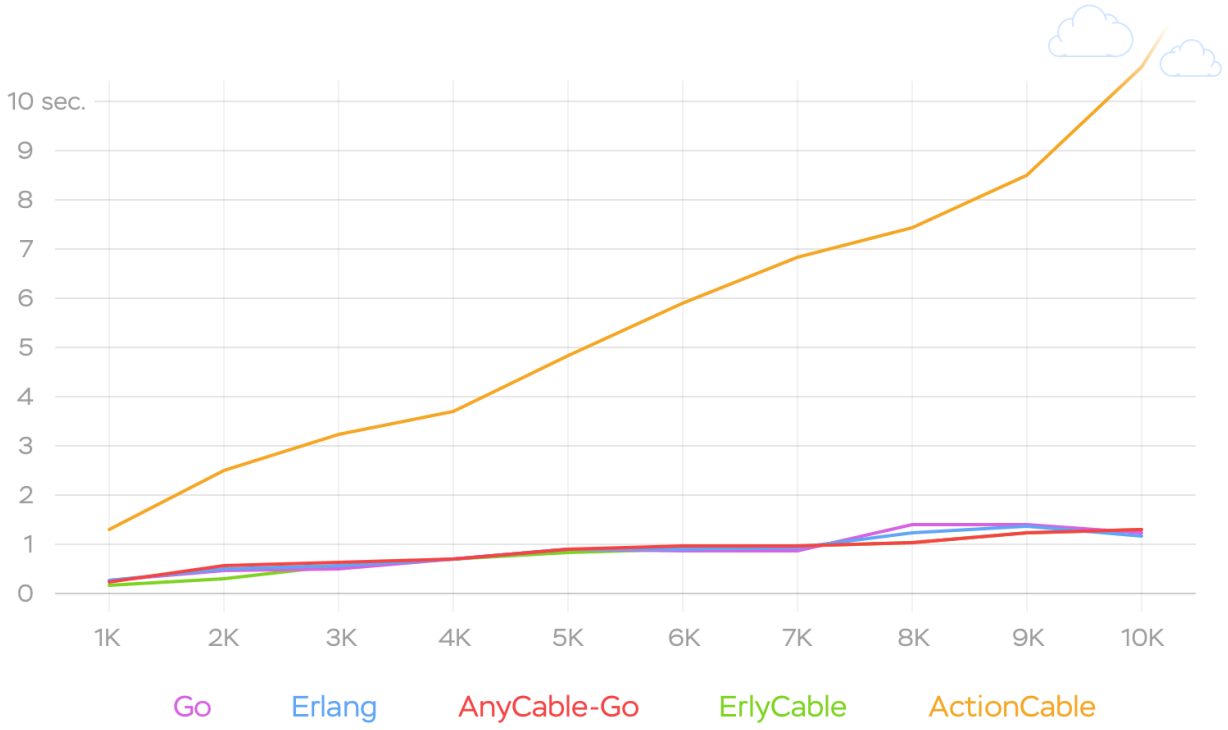


Benchmark: handle 20 thousand idle clients to the server (no subscriptions, no transmissions).

And CPU doesn't feel any pressure too:



Broadcasting time is perfect:



Compatibility

AnyCable aims to be as much compatible with Action Cable, as it possible to use *standard* Action Cable in development and test environments. But some features are rather hard to implement without loss of the performance gain.

See compatibility table for further information.

How to use?

Want to give AnyCable a try? It's pretty easy. Assuming you want to use AnyCable with a Rails app (if not—check out the docs for non-Rails usage):

- Add anycable-rails gem to your Gemfile:

```
gem "anycable-rails"
```

- Run an interactive setup generator:

```
rails g anycable:setup
```

- Install and run compatible WebSocket server, e.g., anycable-go.
- Run AnyCable along with the Rails web server.

  Here is an example `Procfile` for Hivemind or Overmind:

```
web: bundle exec rails s
rpc: bundle exec anycable
go:  anycable-go --port=8080
```

Do not have a Rails application with Action Cable yet? Feel free to play with our demo application!

What's next?

We've learned one aspect of AnyCable – the plug and play performance upgrade for Action Cable.

This approach – extracting low-level functionality from business logic or separation of concerns – can help us solve many problems:

- Support transport fallbacks (e.g. long-polling) or custom transports.
- Build shareable WebSocket servers (i.e., one WebSocket server for several different applications).
- ~~add enhanced analytics and monitoring to your WebSocket server~~—See AnyCable-Go instrumentation
- ~~And even use Action Cable (as a protocol) outside of Rails~~—See Non-Rails usage.

And, of course, you can easily extend your *cable* with custom features, that are not supported (and unlikely to be) by Action Cable.

* * *

AnyCable is a mature open source project, which has already hit the 1.0 release and started the 2.0 development cycle (see the AnyCable 1.0 announcement post).

Check out our official website and GitHub page for more information. Looking for help in building real-time features for your app? Evil Martians offer commercial support, feel free to drop us a line!