

GIANT ROBOTS SMASHING INTO OTHER GIANT ROBOTS

Redis Pub/Sub...how does it work?

June 8, 2011
Nick Quaranto

Redis is a [key/value store](#), but it's jam-packed with a ton of other little utilities that make it a joy to explore and implement. Two of these are the PUBLISH and SUBSCRIBE commands, which enable you to do quick messaging and communication between processes. Granted, there's plenty of other messaging systems out there ([AMQP](#) and [ØMQ](#) come to mind), but Redis is worth a look too.

The way it works is simple:

- [SUBSCRIBE](#) will listen to a channel
- [PUBLISH](#) allows you to push a message into a channel

Those two commands are all you need to build a messaging system with Redis. But, what should you build?

Demo

I tend to build quick little games to learn new ideas, frameworks, languages, etc. For Redis pub/sub I chose to emulate IRC, since “channels” are essentially the same concept for an IRC server. A user connects, talks into a channel, and if others are there, they get the message. This is the basic concept, we're not going to re-implement the IRC protocol here.

I scraped together two tiny little Ruby scripts for this. The repo is [on GitHub](#), if you want to play with it. Make sure you're running Redis locally first! Install Redis via [redis.io](#) if not.

PUBLISH

First up, `pub.rb` publishes messages to a channel. We're going to bring in the [Redis](#) gem to use for making a client, and [JSON](#) in order to have an easy transport format. We could have used Ruby's Marshal class instead to serialize, but this works fine and is human readable.

```
# usage:
# ruby pub.rb channel username

require 'rubygems'
require 'redis'
require 'json'

$redis = Redis.new

data = {"user" => ARGV[1]}

loop do
```

```

msg = STDIN.gets
$redis.publish ARGV[0], data.merge('msg' => msg.strip).to_json
end

```

This script will run interactively, once provided a channel and username from the command line arguments. It then fires off the `PUBLISH` command every time the user types a message and hits Enter. It publishes the message to a channel (in `ARGV[0]`, the first command line argument).

So if we were to run:

```

% ruby pub.rb rubyonrails qrush
Hello world

```

Our Redis client would then send a `PUBLISH` command down the “rubyonrails” channel with the given message. The message itself is JSON and looks like:

```

{
  "msg": "Hello world",
  "user": "qrush"
}

```

We can actually verify this with the [MONITOR](#) command, which will spit out all commands the Redis server has processed. If we had `MONITOR` running before sending the above hello world snippet, it shows:

```

% redis-cli
redis> MONITOR
OK
1306462616.036890 "MONITOR"
1306462620.482914 "publish" "rubyonrails" "{\"user\":\"qrush\",\"msg\":\"Hello world\"}"

```

Currently this simple script doesn’t support publishing under more than one channel. Opening up more than one `pub.rb` process will let you do that.

SUBSCRIBE

Woot! Now that we have messages being sent we have to listen to them. Enter `sub.rb`:

```

require 'rubygems'
require 'redis'
require 'json'

$redis = Redis.new(:timeout => 0)

$redis.subscribe('rubyonrails', 'ruby-lang') do |on|
  on.message do |channel, msg|
    data = JSON.parse(msg)
    puts "#{channel} - [{data['user']}] : #{data['msg']}"
  end
end

```

Once again we’ll need Redis and JSON to connect and parse messages. The initialization process for Redis is different this time: it’s using a new `:timeout` option. This will force the Redis client to never timeout when waiting a response, so we’ll wait forever for messages to come in. Perfect!

This script subscribes to two different channels: `rubyonrails` and `ruby-lang`. Basically, once the interpreter reaches the subscribe block, it will never exit and continue to wait for messages.

When a message comes in, the `message` block is fired, yielding two arguments: the channel the message was on, and the actual data sent down the pipe. Parsing that JSON chunk then allows us to spit out who said it, where they said it, and what was actually said. Here's what it looks like if some other clients are publishing messages after we run our `sub.rb` file. (The published messages arrive in the order they are sent, but that's hard to display in text):

```
% ruby pub.rb rubyonrails qrush
Whoa!
`rake routes` right?

% ruby pub.rb rubyonrails turbage
How do I list routes?
Oh, duh. thanks bro.

% ruby pub.rb ruby-lang qrush
I think it's Array#include? you really want.

% ruby sub.rb
#rubyonrails - [qrush]: Whoa!
#rubyonrails - [turbage]: How do I list routes?
#ruby-lang - [qrush]: I think it's Array#include? you really want.
#rubyonrails - [qrush]: `rake routes` right?
#rubyonrails - [turbage]: Oh, duh. thanks bro.
```

Whoa! How does it work!?! Under the hood in the `redis-rb` client, the `subscribe` block is [actually stuck](#) in a Ruby `loop` (called from [Redis::SubscribedClient#subscription](#)). The client is going to continually attempt to read from the socket for messages, until there's an error of some kind (but not a Timeout!). The redis-server then [keeps a list of channels and patterns](#) for each connected client, and [publishes messages to them](#) when a PUBLISH command is sent.

Usage

Although this is a simple example of how Redis pub/sub works, it's pretty cool to see what others have done with it. Some examples include [Convore](#), which is used as a pretty central part of their infrastructure, and [Realie](#), a real-time code editor like Etherpad. Another good link to check out is [Salvatore Sanfilippo's](#) recent interview on [The Changelog \(around ~24:00-27:00\)](#) where he discusses that developers are switching from other MQs to Redis due to its simplicity and performance.

If you're using Redis' Pub/Sub within your infrastructure, we'd love to hear your feedback on how [Radish](#) can provide more visibility to what your messaging system is up to.

Like



and 4 others liked this.

Add New Comment

[Login](#)

Showing 3 comments

Sort by oldest first ▾



Martijn van Brandevoort

I love Redis (, but I wonder how robust the pub/sub implementation is in terms of transactional safety. If I look at the linked source code (thanks for those!) it all looks pretty simple, which makes me think an extra layer might be required to match the robustness of more sophisticated MQ systems. Can anyone weigh in on this?



Alex Treppass

Redis is in-memory, so persistence / robustness might be an issue. It apparently has a persist-to-disk 'VM' mode, but even the author of redis advises against using it: <http://groups.google.com/group...>






Alex Treppass

Redis is in-memory

' [Subscribe by email](#) + [RSS](#)

2 notes

1.  [zapfmann](#) liked this
2.  [ckalima](#) liked this
3.  [thoughtbot-giantrobots](#) posted this

[blog comments powered by DISQUS](#)

GIANT ROBOTS SMASHING INTO OTHER GIANT ROBOTS is the blog of [thoughtbot](#), a web design and development agency in Boston. For updates, [follow us on Twitter](#) or [subscribe via RSS](#).

GIANT ROBOTS is the blog of [thoughtbot](#), a web design and development agency in Boston.

For updates, [follow us on Twitter](#) or [subscribe via RSS](#).



[Radish](#)

[Redis visibility](#)