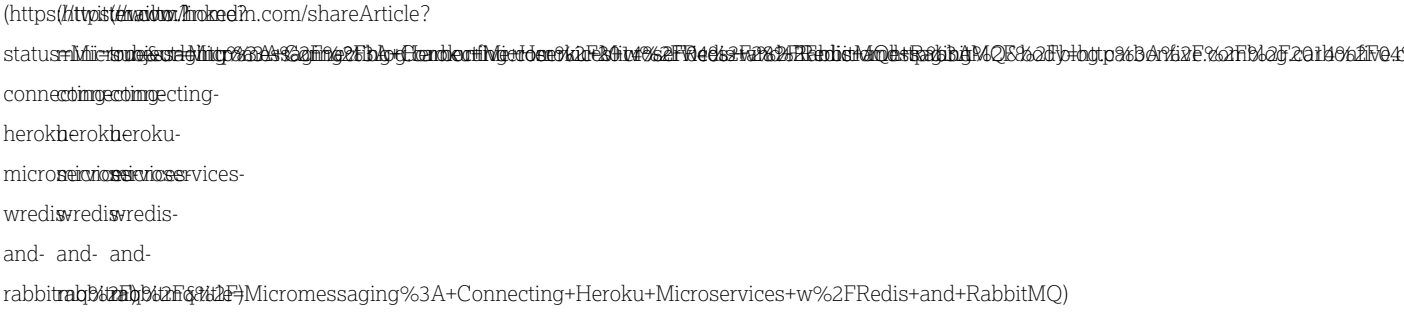
Micromessaging: Connecting Heroku Microservices w/Redis and RabbitMQ

(http://blog.carbonfive.com/author/erin/) Posted on 28th April 2014 *by Erin Swenson-Healey (http://blog.carbonfive.com/author/erin/) in Everything Else (http://blog.carbonfive.com/category/none/)*



While attempting to deploy a system of interconnected microservices to Heroku recently, I discovered that processes running in dynos in the same application cannot talk to each other via HTTP (https://devcenter.heroku.com/articles/dynos#isolation-and-security). I had originally planned on each microservice implementing a "REST" API – but this wasn't going to be an option if I wanted to stick with Heroku. Much head-scratching ensued.

The solution, it turns out, is to communicate between microservices through a centralized message broker – in my case, a Redis database (but I'll show you how do it with RabbitMQ as well, free of charge). The design of each microservice API has been decoupled from HTTP entirely: Client/server communication is achieved by enqueueing JSON-RPC 2.0-encoded messages in a list, with `BRPOP` and return-queues used to emulate HTTP request/response semantics. The Redis database serves as a load balancer of sorts, enabling easy horizontal scaling of each microservice (in Heroku dyno) on an as-needed basis. Redis will ensure that a message is dequeued by only a single consumer, so you can spin up a lot of dynos without worrying that they'll clobber each other's work. It's pretty sa-weet.

So how'd I do it, you ask? Read on!

The Microservice Core

To keep things simple, the server side of our application will consist of a service that calculates the sum or difference between two numbers:

```
1  class Calculator
2
3    def add(a, b)
4      return a+b
5    end
6
7    def subtract(a, b)
8      return a-b
9    end
10
11 end
```

calculator.rb (https://gist.github.com/laser/11303586#file-calculator-rb) hosted with ❤ by **GitHub (https://github.com)** **view raw (https://gist.github.com/laser/11303586/raw/c5aefb0b3facfe42fd6c71c864734fe2f259ec56/calculator.rb)**

Pretty straightforward. Nothing about transport here at all – just domain logic: adding, subtracting.

JSON-RPC 2.0

Communication between client and server is achieved by sending and receiving JSON-RPC 2.0-encoded messages. These messages include the information required to – you guessed it – affect a method call in a remote system. To give you an idea of what this looks like in the context of our calculator:

JSON-RPC 2.0 Request

```
{
  "jsonrpc": "2.0",
  "id": "Ke43jnGeQsk2z9GYvFmrax",
  "method": "add",
  "params": [1, 5.1]
}
```

request.json (https://gist.github.com/laser/11303604#file-request-json) hosted with ❤ by GitHub (https://github.com) view raw (https://gist.github.com/laser/11303604/raw/286f81517a813edd11e3e7da43457957c6065015/request.json)

JSON-RPC 2.0 Response

```
{
  "jsonrpc": "2.0",
  "result": 6.1,
  "id": "Ke43jnGeQsk2z9GYvFmrax"
}
```

response.json (https://gist.github.com/laser/11303640#file-response-json) hosted with ❤ by GitHub (https://github.com) view raw (https://gist.github.com/laser/11303640/raw/a22a6be58d94f721b4df4f741253e6f92c53fc41/response.json)

Our request-messages have four properties: "jsonrpc", which is always "2.0" as per the spec (http://www.jsonrpc.org/specification#request_object); "id", which is a unique identifier created by the client; "method", which is the name of the method we intend to call on the server; and "params," which include values that we intend to pass to the method call. The response-message includes "jsonrpc" and "id" properties in addition to "result" (if the method call was successful) or "error" (if it was not).

Note that the message is **transport-agnostic**. Seeing where I'm going here? Using JSON-RPC allows us to communicate between components in our system – even when HTTP isn't an option.

Redis as a Message Broker

As you've seen, nothing about our JSON-RPC messages say anything about how they're transported between client and server; we can send our message however we want (HTTP, SMTP, ABC, BBD, TLC, OPP, etc.). In this case, we'll implement "sending" a message to the server as an `LPUSH` on to a Redis list. Let's bang out a quick client and I'll explain the interesting parts:

```
#!/usr/bin/env ruby
require 'redis'
require 'securerandom'
require 'json'

# connect to Redis
redis_client = Redis.connect(url: 'redis://localhost:6379')

# request id will be used as the name of the return queue
request = {
  'id' => SecureRandom.hex,
  'jsonrpc' => '2.0',
  'method' => 'add',
  'params' => [1, 5.1]
}

# insert our request at the head of the list
redis_client.lpush('calc', JSON.generate(request))

# pop last element off our list in a blocking fashion
channel, response = redis_client.brpop(request['id'], timeout=30)

# print it out to the console
puts "1+5.1=%.1f" % JSON.parse(response)['result'] # 1+5.1=6.1
```

redis_client.rb (https://gist.github.com/laser/11303717#file-redis_client-rb) hosted with ❤ by GitHub (https://github.com) view raw (https://gist.github.com/laser/11303717/raw/3d753daf44d284e6919769f3c9036234156 30e99/redis_client.rb)

First, we instantiate a Redis client. We use the Redis client to `LPUSH` our messages to a list that both the API client and server know about (" `calc` "). After enqueueing a request-message we can use `BRPOP` to block on receiving a response-message that will be enqueued into a separate return list. This return list's name will be equal to the id on the JSON-RPC request-message. Once we get a result (encoded as JSON), we simply parse it and write to the console.

Next, let's build out the server:

```
#!/usr/bin/env ruby
require 'redis'
require 'json'
```

```
3
4  require './calculator'
5
6  calculator = Calculator.new
7
8  # connect to Redis
9  redis_client = Redis.connect(url: 'redis://localhost:6379')
10
11 while true
12   # pop last element off our list in a blocking fashion
13   channel, request = redis_client.brpop('calc')
14
15   request = JSON.parse request
16
17   # call a method on the calculator instance, passing along params
18   args   = request['params'].unshift(request['method'])
19   result = calculator.send *args
20
21   reply = {
22     'id' => request['id'],
23     'result' => result,
24     'jsonrpc' => '2.0'
25   }
26
27   # 'respond' by inserting our reply at the tail of a 'reply'-list
28   redis_client.rpush(request['id'], JSON.generate(reply))
29
30   # set an expire time for our list to make sure we don't leak
31   redis_client.expire(req_message['id'], 30)
32 end
```

redis_server.rb (https://gist.github.com/laser/11303750#file-redis_server-rb) hosted with ❤ by **GitHub (https://github.com)** **view raw (https://gist.github.com/laser/11303750/raw/06a99a205c3d1ee53030f27e1fccbcfe96f6b0a7/redis_server.rb)**

The server implementation reduces to a loop in which we block on the receival of an inbound JSON-RPC message in our `calc` list. When a message is received, we parse it and pass its arguments to the appropriate method on our calculator instance. Using the id on the request, we enqueue our response-message in a return list and resume polling.

The cool thing about this approach is that we can spin up as many Heroku dynos as we want and Redis will do the load balancing for us. Web scale!

Connecting Through RabbitMQ

In case Redis is not for you, we can achieve the same goals by using RabbitMQ as a message broker. The implementation is straightforward and has a similar feel to what we've done with Redis:

```
1  #!/usr/bin/env ruby
2  require 'bunny'
3  require 'json'
4  require './calculator'
5
6  calculator = Calculator.new
7
8  # connect to RabbitMQ
9  conn = Bunny.new
10 conn.start
11
12 # create a channel and exchange that both client and server know about
13 ch = conn.create_channel
14 q  = ch.queue('calc')
15 x  = ch.default_exchange
16
17 q.subscribe(block: true) do |delivery_info, properties, payload|
18   req_message = JSON.parse payload
19
20   # calculate a result
21   result = calculator.send *(req_message['params'].unshift(req_message['method']))
22
23   reply = {
24     'id' => req_message['id'],
25     'result' => result,
26     'jsonrpc' => '2.0'
27   }
28
29   # enqueue our reply in the return queue
30   x.publish(JSON.generate(reply), {
```

```
    routing_key: properties.reply_to,
    correlation_id: properties.correlation_id
  })
end
```

rabbitmq_server.rb (https://gist.github.com/laser/11303869#file-rabbitmq_server-rb) hosted with ❤ by GitHub (https://github.com) view raw (https://gist.github.com/laser/11303869/raw/743f784439f2f776d2c08e4cd8f024b23d17f990/rabbitmq_server.rb)

The only major difference here is that instead of a never-ending while loop, the call to subscribe is passed a block: true. This will cause the calling thread to block and will prevent the program from exiting until we interrupt it.

Now, for our client:

```
#!/usr/bin/env ruby
require 'bunny'
require 'securerandom'
require 'json'

conn = Bunny.new
conn.start

ch = conn.create_channel
q  = ch.queue('calc', auto_delete: false)
x  = ch.default_exchange

# request id will be used as the name of the return queue
req_message = {
  'id' => SecureRandom.hex,
  'jsonrpc' => '2.0',
  'method' => 'add',
  'params' => [1, 5.1]
}

# we'll set this in the block passed to subscribe below
response = nil

# create a return queue for this client
reply_q = ch.queue('', exclusive: true)

# send out our request, serialized as JSON
x.publish(JSON.generate(req_message), {
  correlation_id: req_message['id'],
  reply_to: reply_q.name,
  routing_key: q.name
})

# subscribe to the return queue in a blocking fashion
reply_q.subscribe(block: true) do |delivery_info, properties, payload|
  if properties[:correlation_id] == req_message['id']
    response = payload            # visible via closure
    delivery_info.consumer.cancel # unblock the consumer
  end
end

# print it out to the console
JSON.parse(response)['message']
puts "1+5.1=%.1f" % JSON.parse(response)['result'] # 1+5.1=6.1
```

rabbitmq_client.rb (https://gist.github.com/laser/11303888#file-rabbitmq_client-rb) hosted with ❤ by GitHub (https://github.com) view raw (https://gist.github.com/laser/11303888/raw/882ab7be0996721cc1f7926364a8e5c6cfba5c95/rabbitmq_client.rb)

If You Come Crawling Back to HTTP…

If you decide to migrate to a new platform that allows you to use HTTP, you can do so with low impact to your codebase. We'll use Sinatra to handle HTTP requests, parsing the request body and marshalling the necessary bits to our calculator:

```
#!/usr/bin/env ruby
require 'sinatra'
require './calculator'

calculator = Calculator.new

post '/calc' do
  req = JSON.parse(request.body.read)

```

```ruby
  # call a method on the calculator instance, passing along params
  args   = req['params'].unshift(req['method'])
  result = calculator.send *args

  # HTTP stuff
  status 200
  headers 'Content-Type' => 'application/json'

  # make sure to encode response as JSON
  JSON.generate({
    'id' => req['id'],
    'result' => result,
    'jsonrpc' => '2.0'
  })
end
```

http_server.rb (https://gist.github.com/laser/11303914#file-http_server-rb) hosted with ❤ by **GitHub (https://github.com)** **view raw (https://gist.github.com/laser/11303914/raw/03c388ab576bac071da7a8d35889170e5a18dd38/http_server.rb)**

Clients can now communicate with our calculator by issuing HTTP POST requests to our "/calc" endpoint:

```ruby
#!/usr/bin/env ruby
require 'net/http'
require 'securerandom'
require 'json'

uri = URI('http://localhost:4567/calc')
req = Net::HTTP::Post.new(uri.path)
req.body = {
  'id' => SecureRandom.hex,
  'jsonrpc' => '2.0',
  'method' => 'add',
  'params' => [1, 5.1]
}.to_json

res = Net::HTTP.start(uri.hostname, uri.port) do |http|
  http.request(req)
end

puts "1+5.1=%.1f" % JSON.parse(res.body)['result'] # 1+5.1=6.1
```

http_client.rb (https://gist.github.com/laser/11303935#file-http_client-rb) hosted with ❤ by **GitHub (https://github.com)** **view raw (https://gist.github.com/laser/11303935/raw/10396567c90c81e3bf54d7b0f8ca833d31a30e2a/http_client.rb)**

Cleaning Up the Cruft

Our client code is easy to understand, but a bit verbose. Let's reduce some of the boilerplate by introducing a new class, using method_missing to remote calls to your API.

```ruby
#!/usr/bin/env ruby
require 'redis'
require 'securerandom'
require 'json'

class RedisRpcClient

  def initialize(redis_url, list_name)
    @redis_client = Redis.connect(url: redis_url)
    @list_name    = list_name
  end

  def method_missing(name, *args)
    request = {
      'id' => SecureRandom.hex,
      'jsonrpc' => '2.0',
      'method' => name,
      'params' => args
    }

    # insert our request at the head of the list
    @redis_client.lpush(@list_name, JSON.generate(request))

    # pop last element off our list in a blocking fashion
    channel, response = @redis_client.brpop(request['id'], timeout=30)

    parsed = JSON.parse(response)
    parsed['result']
  end
```

```
30
31  end
32
33  # create the client and connect to Redis
34  client = RedisRpcClient.new('redis://localhost:6379')
35
36  # remote a call to 'add'
37  sum = client.add(1, 5.1)
38
39  # print it out to the console
40  puts "1+5.1=%.1f" % sum # 1+5.1=6.1
```

fancy_redis_client.rb (https://gist.github.com/laser/11304574#file-fancy_redis_client-rb) hosted with ❤ by **GitHub (https://github.com)** **view raw (https://gist.github.com/laser/11304574/raw/08eed2c90fcb8552ed3f7807609d542b79104352/fancy_redis_client.rb)**

Now we have a reusable Redis-enabled API client whose interface hides the details of serializing hashes to JSON and other boring stuff. Something similar could be done on the server side, deserializing JSON to a method name and `args` to pass to the calculator instance.

In Summary

Occasionally, platforms prevent us from using transport technologies that we're familiar with – HTTP, in this case – and we're stuck investigating new ways of linking things together. In this tutorial I've shown you a few ways to connect the pieces of your system through a centralized message broker. By decoupling our API design from any one particular transport, we've achieved a flexibility unattainable by traditional "REST" APIs, unlocking the ability to horizontally scale our microservices across Heroku dynos with ease.

Notes

1. I'm referring to a "Level Two" implementation as per the Richardson Maturity Model (http://martinfowler.com/articles/richardsonMaturityModel.html).

Feedback

Comments: 24

Mr Rogers
April 28, 2014 at 8:46 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5021)

Nice stuff. I wonder if you looked into speed/load testing this stuff? I recall, from back in my Java days, building an xml-rpc service to query a in-house search engine and we found that we were able to serve far more requests than we could using the engine's built in REST interface.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5021#respond)

Erin Swenson-Healey
April 28, 2014 at 9:40 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5022)

Hey Mr Rogers,

I did a quick comparison of HTTP vs. Redis vs. RabbitMQ using Benchmark and the results were pretty interesting:

Going through HTTP (including Sinatra), I was able to complete 1000 RPC request/response cycles in ~2.43 seconds. With Redis, I was able to complete 1000 RPC request/response cycles in ~0.42 seconds (a 16x speed improvement). RabbitMQ came in at 1000 RPC cycles in 1.60 seconds. Link to Benchmark output here:

https://gist.github.com/laser/11377229 (https://gist.github.com/laser/11377229)

…and code run for comparison here:

https://gist.github.com/11377327.git (https://gist.github.com/11377327.git)

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5022#respond)

Carl Hörberg
May 1, 2014 at 8:15 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5027)

The reason for the lower RabbitMQ result is that you're canceling the consumer and then opens a new subscribe again. In a normal use case this is probably neglectable time, but in a tight benchmark loop it looks bad.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5027#respond)

Erin Swenson-Healey
May 1, 2014 at 8:39 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5028)

Hey Carl,

Thanks for the tip. What would be a better way to write the RabbitMQ portion of the benchmarking code such that I was doing an apples-to-apples comparison with Redis?

E

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5028#respond)

Carl Hörberg
May 1, 2014 at 7:03 pm (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5030)

https://gist.github.com/carlhoerberg/2c1fde3e5992092b2d81
(https://gist.github.com/carlhoerberg/2c1fde3e5992092b2d81)

Keep a subscriber open, and use an internal ruby queue to block-pop for messages (this also solves a race condition bug in your code, where messages with "wrong correlation id" would be discarded).

Erin Swenson-Healey
May 2, 2014 at 10:00 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5035)

Hey Carl,

The intermediate Ruby-queue solution you've provided is great. I'll update the article accordingly. Thanks for the gist!

Brendon Murphy
April 29, 2014 at 9:34 pm (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5025)

Dig this article, shared it with some co-workers.

A nice robustness enhancement to the redis implementation could be the brpoplpush command (http://redis.io/commands/brpoplpush (http://redis.io/commands/brpoplpush)). There's a simple queue implementation Ost (https://github.com/soveran/ost/blob/master/lib/ost.rb#L31 (https://github.com/soveran/ost/blob/master/lib/ost.rb#L31)) that uses it in a clever fashion for failures.

Basically you put requests on a temp list per worker. If disaster strikes between the time the worker pops the request and has a chance to reply, you still have the request hanging around in the 'backup' list for that worker for inspection, recovery, etc.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5025#respond)

Erin Swenson-Healey
May 2, 2014 at 9:35 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5033)

Hey Brendon Murphy,

Thanks for the tip. These implementations are not bombproof; rather, I'm trying to communicate (in as little code as possible) the flexibility that a developer affords herself when building APIs whose design has been decoupled from any single transport mechanism. HTTP, SMTP, Redis, IronMQ – whatever. They're just implementation details.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5033#respond)

stephennguyen
May 1, 2014 at 12:23 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5026)

Have you had a chance to benchmark IronMQ as apart of this test of microservice messaging? The backend servers were built for speed, efficiency, reliability, and has some great features using golang as our foundation.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5026#respond)

Erin Swenson-Healey
May 2, 2014 at 9:21 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5032)

Hey stephennguyen,

I have not tried this with IronMQ. Conceptually, I presume that an implementation using IronMQ would look similar to the Redis and RabbitMQ examples. If you were to provide an example, I'd include it in the article.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5032#respond)

stephennguyen
May 2, 2014 at 10:49 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5037)

Very curious I'll get it done and send it over to you. great post very informative!

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5037#respond)

findchris
May 1, 2014 at 11:00 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5029)

Check out: https://github.com/rack-amqp/jackalope (https://github.com/rack-amqp/jackalope)
I saw it presented at RailsConf, and basically does what you want, and more.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5029#respond)

Erin Swenson-Healey
May 2, 2014 at 9:41 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5034)

Hey findchris,

I'm having a difficult time understanding how jackelope relates to the article. Perhaps you could elaborate?

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5034#respond)

findchris
May 2, 2014 at 11:15 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5039)

It seems like a kindred project.

– "The solution, it turns out, is to communicate between microservices through a centralized message" – check
– "emulate HTTP request/response semantics" – check
– "The design of each microservice API has been decoupled from HTTP entirely" – check

I feel like your motivations align with that of the rack-amqp/jackalope projects.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5039#respond)

Erin Swenson-Healey
May 2, 2014 at 11:26 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5040)

Gotcha, gotcha. I did some digging into the jackelope source; makes sense to me now what they're trying to do. Excellent slides (that you linked to), btw.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5040#respond)

Jeff Dickey
May 2, 2014 at 10:22 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5036)

@foundchris:disqus it looks like this code basically allows you to swap amqp in as a replacement for http. Just curious, have you integrated this into a system? I'm curious why you would want to implement it

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5036#respond)

findchris
May 2, 2014 at 11:13 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5038)

That's about right. I have not implemented it, but found it to be an interesting experiment. Here are some slides on the project: https://speakerdeck.com/joshsz/rack-amqp-ditch-http-inside-soa (https://speakerdeck.com/joshsz/rack-amqp-ditch-http-inside-soa)

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5038#respond)

Christopher Dell
May 3, 2014 at 6:37 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5041)

Great article, and very timely for me. Much <3<3<3<3.

I've wrapped the client/server implementation for RabbitMQ into a small gem call `burrow` that you can have a look at here https://github.com/tigrish/burrow (https://github.com/tigrish/burrow). This means not having to repeat connection info and so on for every app.

Again, thanks for the article!

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5041#respond)

Erin Swenson-Healey
May 7, 2014 at 10:03 pm (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5051)

Hey Christopher Dell,

No problemo. I'm glad you enjoyed the post. Thanks for the link to your burrow lib.

Erin

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5051#respond)

Tammer Saleh
May 5, 2014 at 10:54 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5042)

I'm a bit confused by the statement that you can't communicate between dynos. Certainly you can't communicate between distinct dyno processes, but I'm not sure why you'd want to do that. Wouldn't you put each microservice application behind its own domain name and communicate via http/rest on those end points?

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5042#respond)

Erin Swenson-Healey

May 6, 2014 at 9:47 pm (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5046)

Hey Tammer Saleh,

There are a few motivations for this approach:

1. Since I won't be exposing those services to the Internet (they're running as instances of non-'web' dynos), I won't have to think much about authentication. If I ever did want to expose them to the outside world, I could centralize all authentication in an HTTP gateway bound to $PORT, which could then bridge incoming HTTP requests to the Redis/RabbitMQ/whatever back end.

2. If possible, I'd like to avoid a world in which I'd have to incur the mental overhead of deploying the services as separate Heroku applications. If each API were to be available via HTTP, each service would need to be running as an instance of distinct Heroku application's 'web' dyno. I'd prefer to deploy the entire application in a single shot.

3. Even if HTTP were an option for sending RPC messages between the UI portion of my application (bound to $PORT) and each service – I'd still go with a message queue as an intermediary; the speed difference is huge.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5046#respond)

Tammer Saleh
May 7, 2014 at 2:28 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5047)

I get #1, and it'd be great if Heroku offered an "internal-only" app (or the equivalent of AWS security groups.

I personally disagree on #2. One of the benefits of a microservice architecture is that you can decouple the development and deployment of each bit of the overall application. It also facilitates scaling each microservice independently, where in this approach, you must run each microservice process inside all of your dynos, using up resources.

#3 is a great point, but I still feel that this architecture is overall pretty confusing when compared to just http communication.

But it's a great article – thanks for posting it!

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5047#respond)

Erin Swenson-Healey
May 7, 2014 at 7:52 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-5049)

Hey Tammer Saleh,

Regarding scaling: What I've been doing is running each service as, essentially, an instance of a worker dyno. That is to say, if I have a UI (the 'web' dyno) backed by three services: contact service, user service, reporting service – then I'd have a Procfile that'd look something like:

```
# Procfile
web: bundle exec rails s -p $PORT
user_svc: sh -c 'cd services/user; ./service_runner.rb'
contact_svc: sh -c 'cd services/contact; ./service_runner.rb'
reporting_svc: sh -c 'cd services/reporting; ./service_runner.rb'
```

This allows me to horizontally scale each service independently; it's just a simple matter of adding more dynos. Beefing up my persistence layer can be handled independently from the services themselves.

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=5049#respond)

lindsay
March 9, 2015 at 8:12 am (http://blog.carbonfive.com/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/#comment-237658)

wish someone would repeat this article for node 🙂

Reply (/2014/04/28/micromessaging-connecting-heroku-microservices-wredis-and-rabbitmq/?replytocom=237658#respond)

Your feedback

Your name (required)

Your name

Your email address (required, but will not be published)

Your email address

Your website if you have one (not required)

Your website url

Your comment

Your comment

Submit comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

(http://carbonfive.com)

About (http://carbonfive.com/about) Work (http://carbonfive.com/work) Events (http://carbonfive.com/events) Notes (/)

Careers (http://carbonfive.com/careers) Contact (http://carbonfive.com/contact)

Subscribe

(https://twitter.com/carbonfive) (https://github.com/carbonfive)
(https://www.facebook.com/carbon5) (https://www.linkedin.com/company/carbon-five)