RabbitMQ Examples – Microservices Monitoring

Why microservices?

Microservices – or microservices architecture – are applications that are arranged or structured as a collection of loosely coupled services. Some of the benefits of using a microservices architecture are:

* the ability to independently develop and deploy services
* speed and agility
* better code quality
* code created/organized around business functionality
* increased productivity
* easier to scale
* freedom to choose the implementation technology/language

How can RabbitMQ help us build such an application?

As usual, in the first place we need to think about what we need and to model our application. Let’s say that we want to be able to implement services that perform specific tasks and that can also communicate to each other, if they need to exchange information. And also, we anticipate that the application will grow a lot in terms of number of services, so we want to be able to monitor the health of each service.

We will not bother much with specifics. For now, we will have a single kind of service that does absolutely **nothing** – that’s exciting! – but which can be extended and given some functionality and meaning in the future.

However, we will focus on the monitoring part. For this, we will create a Monitor – or a **Big Brother**, if you want to call it like that – whose only job is to make sure that the services are still running and hopefully doing their job. And doing it **well**. The monitor will be permanently listening to the service’s **heartbeats** – not creepy at all – and will notify the supervisor if something goes bad with a certain service.

This is a relatively simple thing to implement. As mentioned, the monitor will receive **messages**, and messages contain some required information, such as the **originator**, the **receiver**, the **content**, which, to allow for flexibility, will be a list of key-value pairs, and a **type**. The type will be one of **CONNECT**, **HEARTBEAT**, **INFO**, **DISCONNECT** and **BITCONNEEEEEECT**.

Okay, maybe not the last one. Here’s the meaning of each of them:

* **CONNECT** tells the monitor that the service just started and it has connected to the monitor;
* **HEARTBEAT** tells the monitor that the service is healthy, family and friends are okay and the job is still going on well;
* **INFO** is a kind of “general-purpose” message – it can contain notifications, for example;
* **DISCONNECT** tells the monitor that the service has, well, disconnected from the monitor and is shutting down.

Now that we talked about the awesome monitor, it’s time we talk about the service. As previously described, it sends a **CONNECT** message when it starts up, a **DISCONNECT** message when it shuts down, and everything in between will be **HEARTBEAT** messages.

The thing is, like for humans, heartbeats are more or less regulated – remember seeing that cute person who you were too shy to talk to?

Anyway. The same goes for services – they should have a heartbeat at a fixed time interval, which is being specified to the monitor when the service connects to it – something like 60 seconds, which in Africa is also the equivalent of one minute. This way, if the monitor does not receive **any** heartbeat from the service after the specified time interval, the monitor can freak out and tell its supervisor that the service is down and the situation should be investigated and fixed as soon as possible, otherwise the clients will get angry and maybe cry. And they won’t pay you. And you will get angry too. And then you go home and make your wife angry. And then everyone becomes angry, and this is the butterfly effect.

And that’s pretty much it. The services speak, the monitor listens and reports any weird behaviour.

Are there any other ways to approach this?

Of course, there are many approaches to such a problem, but this was a simple example showing another use-case for a framework like RabbitMQ. After all, this is the topic we’re talking about. Whether you like it or not. I hope you do.

Another solution could be, for a web service for example, to have a /healthcheck endpoint which can be queried periodically instead of the service sending heartbeats itself, thus the responsibility of updating the state shifting to the monitor.

However, I consider that using a messaging framework offers more versatility, since additional functionalities can be added to the monitor, messages can be sent and maintained in the monitor for future interrogations – state changes, notifications, etc.



rabbitmqute

Bibliography and Resources:

1. <https://dzone.com/articles/why-microservices>
2. <https://solace.com/blog/microservices-advantages-and-disadvantages/>
3. <https://www.cloudamqp.com/blog/why-use-rabbitmq-in-a-microservice-architecture.html>
4. <https://tanzu.vmware.com/developer/blog/understanding-the-differences-between-rabbitmq-vs-kafka/>
5. <https://dzone.com/articles/rabbitmq-in-microservices>
6. <https://stackoverflow.com/questions/45208766/microservices-why-use-rabbitmq>
7. <https://medium.com/aspnetrun/microservices-event-driven-architecture-with-rabbitmq-and-docker-container-on-net-968d73052cbb>
8. <https://www.erlang-solutions.com/blog/using-rabbitmq-for-microservice-architecture-success-guest-blog-by-api-fortress-api/>
9. <https://en.wikipedia.org/wiki/Heartbeat_(computing)>
10. <https://en.wikipedia.org/wiki/Big_Brother_(Nineteen_Eighty-Four)>
11. <https://www.youtube.com/watch?v=temt8sLWLi8>
12. <https://www.youtube.com/watch?v=e5nyQmaq4k4>
13. <https://www.youtube.com/watch?v=dQw4w9WgXcQ>