Introduction to asynchronous communication

In this lesson we will learn about the Asynchronous Communication style and how it can help us enable much more resilient communication between our services.

The problem with synchronous communication

Imagine a scenario where after receiving a request from a client our microservice needs to reach out to two other microservices. Also, each of these services need to get in touch with other services which in turn might need to further reach to more services. There may also be services that depend on the services our service depends on.

We originally fine tuned our service so that it never takes more than 300 milliseconds to respond to our client requests. However, when we started making synchronous calls to one of our dependent microservices we had to add to our time the 200 milliseconds that service can take to respond. That was still ok in the beginning. But unfortunately, when that dependency eventually started calling another dependency and that one to yet another one, our overall average latency bumped to 1450 milliseconds, which is bad.

On top of that, if one of our deeper indirect dependencies starts failing, even if temporarily, it can cause all of the services that depend on it to also start failing and the effect keeps cascading until potentially most of our system becomes unavailable.

One important concept that you should keep in mind as you embrace microservices is the Service Level Agreement or SLA. The SLA is basically a commitment between you as the service provider and a client. For instance, as part of our SLA we could have defined that our service can be expected to be up and running 99.9% of the time, so when our clients choose to use our service they should expect it to be down approximately 44 minutes during the month, which the owners of the client will have to decide if it's acceptable.

However, when using synchronous communication calculating such numbers for our service SLA gets more complicated. Our deepest dependency is also advertising a 99.9% up time in their SLA, and let's say all our services initially advertise that value. However, in our scenario, when dependency 4 fails, it impacts the SLA of dependency 3, which now sees its SLA down to 99.8%. This in turn impacts the SLA of dependency 1 and ultimately impacts our own SLA. Now with an SLA of 99.6% up time our clients would be subject to about 175 minutes of down time across the month, which is way below of what we wanted to provide initially.

So, the synchronous communication style suffers from an increase of latency due to the chain of calls, it can significantly amplify the impact of partial failures and can potentially reduce the SLA of our service.

Asynchronous communication style

Now let's look at the asynchronous communication style. Here the client does not have to wait for a response in a timely manner. And, in fact, depending on how the communication has been setup, there might be no response at all.

To enable such communication there is usually an intermediary called the message broker, which is pretty dumb in nature, meaning that it has no business logic on it, and it is also highly available.

With the message broker in place the service client sends messages to the broker and the broker forwards the messages to receivers as soon as possible. The messages can be received by two types of receivers:

- A single receiver, in which case we think of the message as a command via which the client requests an action on the receiving service. For instance, when a purchase operation starts, our Trading service will send a command to our Inventory service asking it to grant items to the user's inventory and a command to our Identity service asking it to debit gil from the user.
- We could also have multiple receivers, where there are multiple services that subscribe to the events published by the client service. This would be the case if, for instance, our Catalog service would like to publish any updates to its catalog of items so that other services can be informed about the changes.

Microservices autonomy

One of the key benefits of using asynchronous communication is that it enforces the microservices autonomy. Let's see how this is enabled.

In a single receiver scenario, we would have again our client talking to our service but our service will not talk to it's dependent service. It will instead send an asynchronous message to a broker and it will immediately acknowledge to the client the successful reception of the request. The dependent service will consume this message from the message broker as soon as possible and will eventually provide a reply via the same broker. At that point the client can request the status of its initial request or our service can notify the client of the response.

All of the services in our system would follow the same message-based approach, including cases where a reply is not required from the called service.

The great thing about this kind of layout is that when one of the dependent services fail it would not cause any impact on the other services, since all of them have been decoupled from each other and only talk to the message broker. So the same SLA that one of the services presents can be honored across the entire system, assuming the message broker offers high availability.

So, with asynchronous communication partial failures are no longer propagated, each service has its own independent SLA and, best of all, the autonomy of all microservices is enforced given the lack of coupling between them.

Asynchronous propagation of data

One of the nice things that asynchronous communication provides is the ability to asynchronously propagate data across services.

In the synchronous communication scenario, when the client requests the user's inventory, and since the Inventory service only has the data that's relevant to it, Inventory has to first talk to the Catalog service in order to retrieve additional details about each item. How can asynchronous communication help here?

Well, thanks to the presence of the message broker, what we can do now is have the Catalog service publish an event each time a catalog item is created (or updated or deleted). Such event doesn't need to have all the details of each item, but only the ones of interest to client services. We can then have our Inventory service listen to these events and create its own collection of Catalog items in its own database. As long as we have a highly available message broker, the list of catalog items in the Inventory database should be eventually consistent with regards to what's in the Catalog service.

Now, when the client requests the user's inventory, Inventory has no need to go out into any dependent service to request item details. It has everything it needs in its own database and it can immediately provide a reply to the client with no additional latency.

So thanks to asynchronous communication we can enable eventual consistency across our system, the autonomy of our services is preserved and the previous inter-service latency is reduced or completely removed.

Implementing asynchronous communication

In the next lesson we will start implementing asynchronous communication between our two microservices. We will introduce RabbitMQ as our message broker of choice given that it supports the AMQP protocol, it is lightweight, easy to run locally and is very popular in the open source community.

RabbitMQ introduces the concept of exchanges, which you can compare to a mailbox. When a service like Catalog needs to publish a message it will send it to an exchange in RabbitMQ and, with the appropriate bindings in place, the exchange will distribute the message to any of the configured queues. From there, RabbitMQ will take care of delivering the message to any subscribed services like Inventory.

We could use RabbitMQ directly in our code, but we would first need to create the exchanges and queues and configure the bindings to ensure messages are properly routed. Also, our code would be tied to RabbitMQ and, if we ever wanted to move to another message broker, we would need to rewrite a good part of our services code. For this and a few other reasons, we will instead use MassTransit on top of RabbitMQ.

MassTransit is a popular open source distributed application framework for .NET. MassTransit can take care of doing most of the RabbitMQ configuration for us and is able to integrate with multiple other message brokers while allowing our services to stick with a higher layer of broker agnostic APIs. As part of this MassTransit introduces the concept of a publisher and a consumer, which is what our service code will focus on.

Let's start coding.