Distributed Transactions

In this lesson you will learn about distributed transactions and the challenges presented when implemented via the two-phase commit algorithm.

What is a distributed transaction?

What is a distributed transaction? The following definition, taken from Wikipedia, describes it pretty well. A distributed transaction is a database transaction in which two or more network hosts are involved.

This usually means that, instead of performing the transaction in a single database hosted in a single server, you need to perform the transaction across multiple databases hosted in different computers or nodes on a network.

As you can expect, this is pretty challenging, but there is a common algorithm typically used to ensure the correct completion of distributed transactions, which is known as Two Phase commit.

Two-Phase Commits

To understand how the two-phase commit algorithm works, let's go back to our purchase scenario, now in the context of our microservices architecture.

Here we have again our client and our Inventory and Identity microservices, each of them with their own database. The two-phase commit algorithm requires a central coordinator which drives the whole distributed transaction. So, let's bring in a new Trading microservice to fulfil this role.

Once again, our user requests the purchase of 5 potions via our client app, who sends the corresponding request to the Trading service. Now, instead of sending a request to Inventory to just grant the 5 potions to the user, it asks a question: "can you grant 5 potions to this user?"

This is what we call the voting phase and to honor this request Inventory will likely need to lock the corresponding database record to make sure no other process can modify it until the distributed transaction is complete. Inventory would then confirm it can grant the 5 potions to the user.

After that, Trading asks a similar question to the Identity service: "can you debit 25 gil from this user"? Identity would likely confirm the user has enough gil available and then would also lock the user database record to make sure no other process can modify the amount of gil. Identity also confirms it can perform the requested operation.

At this point, if any of the participants state that it cannot fulfill the request, the entire transaction is aborted, and no change is applied.

However, if the coordinator has received a successful confirmation from all participants, then it moves to the commit phase, where it requests each participant to complete the requested operation. Inventory will then actually make the update in the InventoryItems table of its database, releases the lock and reports the success back. And the same way, Identity updates the Users table, releases the lock and reports the success.

Finally, the coordinator reports the successful completion of the transaction back to the client.

This solves the issue of making sure all changes happen across all participants or none of them happen. However, it does present a few issues too.

To start with, the fact that the coordinator has to wait for each participant to reply with a yes or no during the voting phase can inject huge amounts of latency into the system. Keep in mind that not only Trading, but also Inventory and Identity are all waiting at that point just to move to the next phase and in the meanwhile, there could be dozens or hundreds of other purchases waiting to get fulfilled.

The locking of resources in each service can also become a serious performance bottleneck. None of the participants can allow modifications during the voting phase, so we could again have lots of processes waiting for those locks to get released, slowing down the overall system.

Finally, there are a bunch of error scenarios that could complicate things further. For instance, think about what would happen if one of the services becomes unavailable during the commit phase? We have been designing our microservices to make sure the availability of any of them is never an issue for the others, but now an outage in a single service can put everything into a very problematic state.

So, in general, the best advice I can give you is to avoid distributed transactions when building microservices. It's just not worth it. And there's a much better approach to tackle transactions across microservices.

In the next lesson we will learn about Sagas and how they can coordinate things without the issues presented by two-phase commits.