Handling Distributed Transactions in the Microservice world

Sohan Ganapathy Following
May 10, 2019 · 7 min read ★

Everyone today is thinking about and building Microservices — me included. Microservices, from its core principles and in its true context, is a distributed system.

What is a distributed transaction?

Transactions that span over multiple physical systems or computers over the network, are simply termed Distributed Transactions. In the world of microservices a transaction is now distributed to multiple services that are called in a sequence to complete the entire transaction.

Here is a monolithic e-commerce system using transactions:

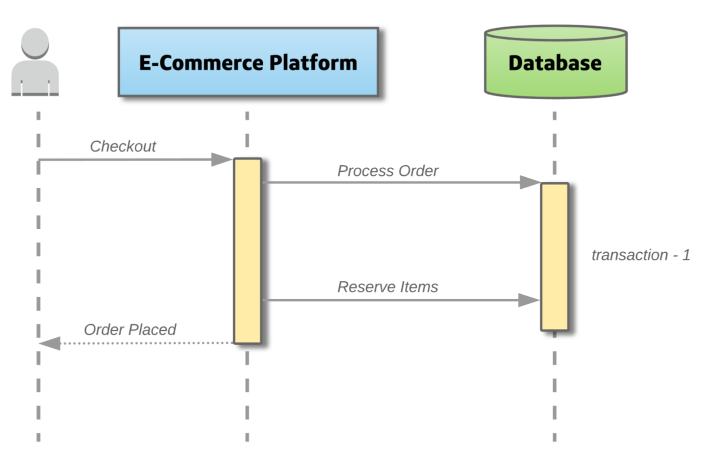

image 1: Transaction in a monolith

In the system above, if a user sends a **Checkout** request to the platform, the platform will create a local database transaction that works over multiple database tables, to **Process** the order and **Reserve** items from the inventory. If any step fails, the transaction can **roll back**, both the order and items reserved. This is known as ACID (Atomicity, Consistency, Isolation, Durability), which is guaranteed by the database system.

Here is the e-commerce system decomposed as microservices:

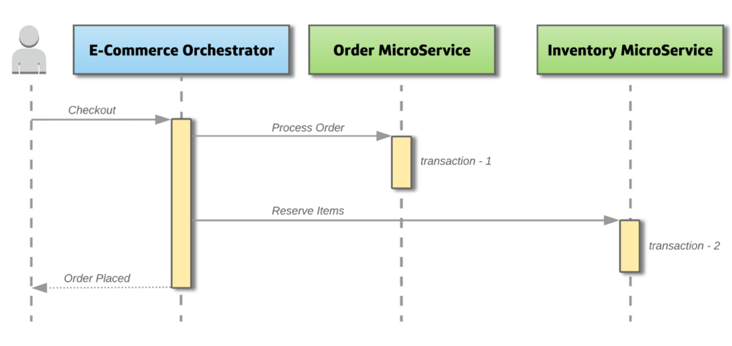

image 2: Transactions in a microservice

When we decompose this system, we created the microservices `OrderMicroservice` and `InventoryMicroservice`, which have separate databases. When a **Checkout** request comes from the user, both these microservices will be invoked to apply changes into their own database. Because the transaction is now across multiple databases via multiple systems, it is now considered a **distributed transaction**.

What's the problem with distributed transactions in microservices?

With the advent of microservice architecture we are losing the ACID nature of databases. Transactions may now span multiple microservices and therefore databases. The key problems we would face are:

How do we keep the transaction atomic?

Atomicity means that in a transaction either all steps are completed or no step is completed**.** In the example above, if the 'reserve items' in the `InventoryMicroservice` method fails, how do we roll back the 'process order' changes that were applied by the `OrderMicroservice`?

How do we handle concurrent requests?

If an object from any one of the microservice is being persisted to the database and at the same time, another request reads the same object. Should the service return the old data or new ? In the example above, once `OrderMicroservice` is complete and the `InventoryMicroservice` is now performing its update, should requests for number of orders placed by the customer include the current order?

Today systems are designed for failures and some of the main problems faced is handling distributed transactions, to quote Pat Helland.

> In general, application developers simply do not implement large scalable applications assuming distributed transactions. — Pat Helland

Possible Solutions

The above two problems are pretty crucial while designing and building microservice based applications. To address them the following list of approaches have been described:

- Two-Phase Commit
- Eventual Consistency and Compensation / SAGA

1. Two-Phase Commit

As its name suggests, this way of handling transactions has two stages, a *prepare* phase and a *commit* phase. One important participant is the **Transaction Coordinator** which maintains the lifecycle of the transaction.

How it works:

In the prepare phase, all microservices involved prepare for commit and notify the coordinator that they are ready to complete the transaction. Then in the commit phase, either a commit or a rollback command is issued by the transaction coordinator to all microservices.

Lets take the e-commerce system as an example:

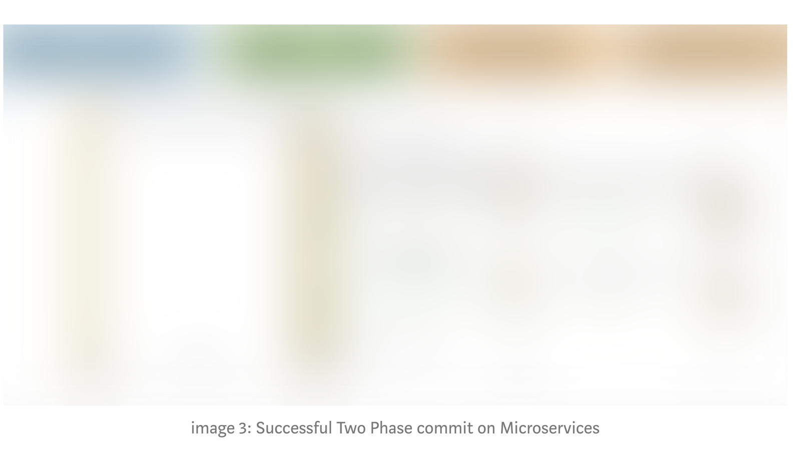

image 3: Successful Two Phase commit on Microservices

In the example above (image 3), when a user sends a checkout request the `TransactionCoordinator` will first begin a global transaction with all the context information. First it will send out a *prepare* command to the `OrderMicroservice`, to create an order. Then it will send out a *prepare* command to the `InventoryMicroservice`, to reserve the items. When both the services are OK to perform the change, they lock down the objects from further changes and notify the `TransactionCoordinator`. Once the `TransactionCoordinator` has confirmed that all microservices are ready to apply their changes, it will then ask them to persist their changes by requesting a commit with the transaction. At this point, all objects will be unlocked.

image 4: Failed Two Phase commit on Microservices

In a failure scenario (image 4) - if at any point a single microservice fails to prepare, the `TransactionCoordinator` will abort the transaction and begin the rollback process. In the diagram, the `OrderMicroservice` failed to create an order for some reason, but the `InventoryMicroservice` has replied that it is prepared to create the order. The `TransactionCoordinator` will request an abort on the `InventoryMicroservice` and the service will then roll back any changes made and unlock the database objects.

Advantages

- The approach guarantees that the transaction is atomic. The transaction will end with either all microservices being successful or all microservices have nothing changed.
- Secondly, it allows read-write isolation, the changes on objects are not visible until the transaction coordinator commits the changes.
- The approach is a synchronous call, where the client would be notified of success or failure.

Dis-Advantages

- Everything isn't perfect, two phase commits are quite slow compared to the time for operation of a single microservice. They are highly dependent on the transaction coordinator, which can really slow down the system during high load.
- The other main drawback is the locking of database rows. The lock could become a performance bottleneck and it is possible to have a **Deadlock,** where two transactions mutually lock each other.

Top highlight

2. Eventual Consistency and Compensation / SAGA

One of the best definitions of eventual consistency, is described on microservices.io: *Each service publishes an event whenever it updates its data. Other service subscribe to events. When an event is received, a service updates its data.*

In this approach, the distributed transaction is fulfilled by asynchronous local transactions on related microservices. The microservices communicate with each other through an event bus.

How it works:

Again, lets take the e-commerce system as an example:

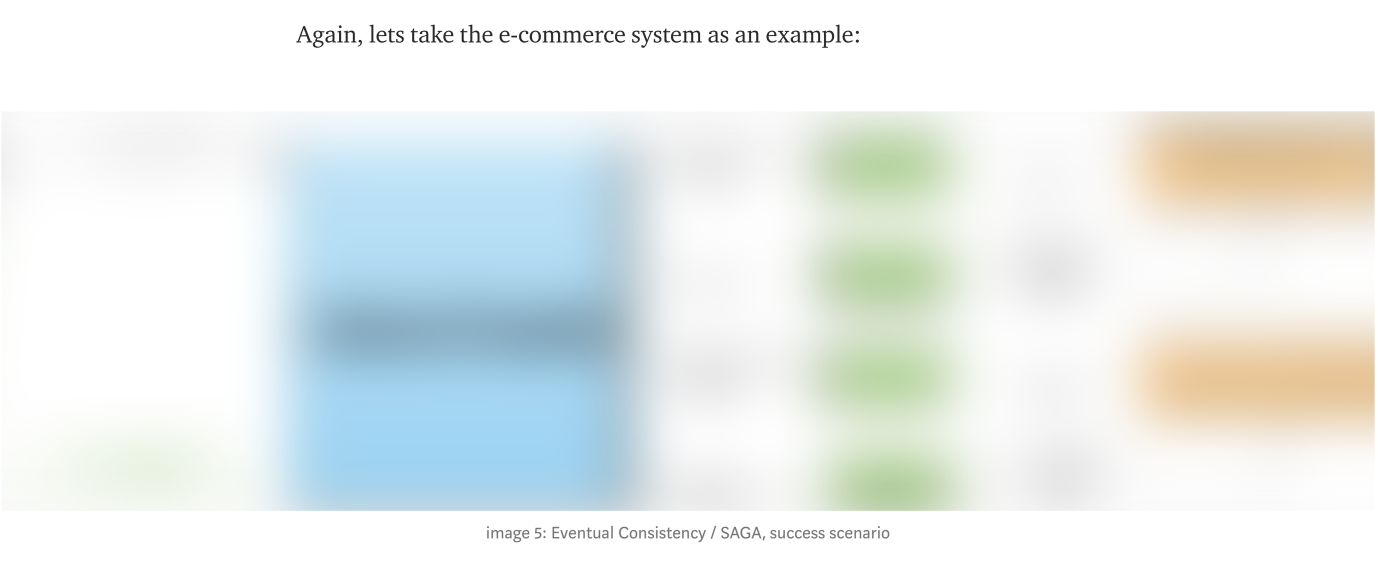

image 5: Eventual Consistency / SAGA, success scenario

In the example above (image 5), the client requests the system to *Process The Order.* On this request the `Choreographer` emits an event *Create Order*, marking the start of the transaction. The `OrderMicroservice` listens to this event and creates an order, if it was successful it emits an *Order Created* event. The `Choreographer` listens for this event and proceeds to reserve the items, by emitting the *Reserve Items* event. The `InventoryMicroservice` listens for this event and reserve's the items, if it was successful it emits an *Items Reserved* event. Which in this example means the end of the transaction.

All the event based communication between microservices happen via the Event Bus and is Choreographed by another system to address the complexity issue.

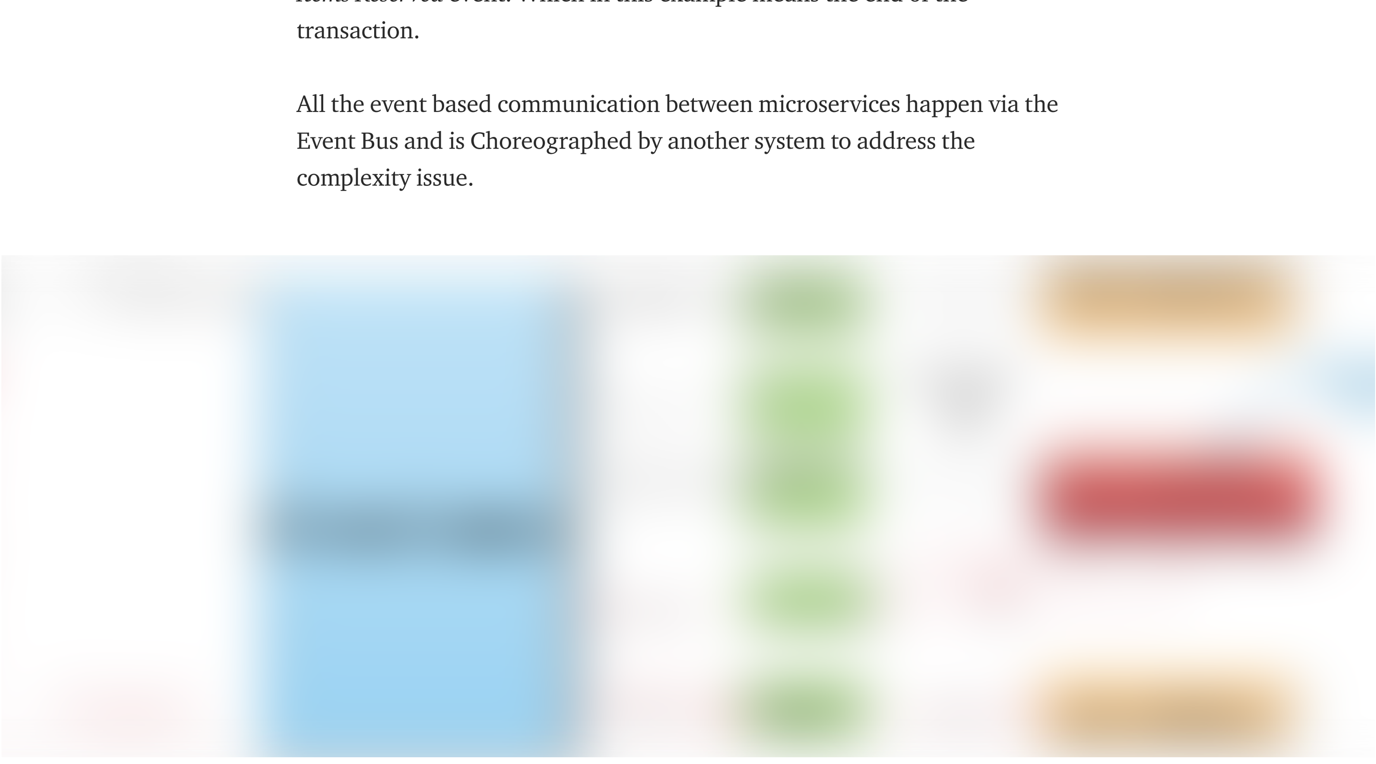

image 6: Eventual Consistency / SAGA, failure scenario

If for any reason the `InventoryMicroservice` failed to reserve the items (image 6), it emits a *Failed to Reserve Items* event. The `Choreographer` listens for this event and starts a **Compensating Transaction**, by emitting a *Delete Order* event. The `OrderMicroservice` listens to this event and deletes the order that was created.

Advantages

One big advantage of this approach is that each microservice focuses only on its own atomic transaction. Microservice's are not blocked if another service is taking a longer time. This also means that there is no database lock required. Using this approach makes the system highly scalable under heavy load, due to its asynchronous event based solution.

Dis-Advantages

The main disadvantage, is the approach does not have read isolation. Which means, in the above example the client could see the order was created, but in the next second, the order is removed due to a compensating transaction. Also, when the number of microservices increase it becomes harder to debug and maintain.

Conclusion

First alternative is to avoid needing distributed transactions. If it is a new application being built, start with a monolith as described in MonolithFirst by Martin Fowler. To quote a section, from the page.

> A more common approach is to start with a monolith and gradually peel off microservices at the edges. Such an approach can leave a substantial monolith at the heart of the microservices architecture, but with most new development occurring in the microservices while the monolith is relatively quiescent. — Martin Fowler

When there is a need to update data in two places as a result of one event, Eventual Consistency / SAGA approach is a preferable way of handling distributed transactions as compared to the two-phase commit. The main reason being two-phase commit does not scale in a distributed environment. The Eventual Consistency approach also introduces a new set of problems, such as how to atomically update the database and emit an event. Adoption of this approach requires a change in mindset for both development and testing teams.

Microservices Eventual Consistency Two Phase Commit Distributed Transaction

1.4K claps

WRITTEN BY
Sohan Ganapathy
Following
Software Architect | Full-Stack Engineer

The Startup
Follow
Medium's largest active publication, followed by +633K people. Follow to join our community.

See responses (13)

Discover Medium
Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours
Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Explore your membership
Thank you for being a member of Medium. You get unlimited access to insightful stories from amazing thinkers and storytellers. Browse