HLD: Microservices 2

Summary of Last Class

In the last class, we discussed the following:

- Why are Microservices needed?
- Microservices vs Monolith
- Microservices vs SOA.
- Cons of Microservices which were primarily around data inconsistencies, observability and isolation.
- And finally, observability in Microservices.

Today, we will spend more time on how to ensure consistency and make distributed transaction work. We will also focus on how to isolate microservices from other microservices being down.

Consistency in Microservices

- In a monolith , imagine I have to transfer 10 rupees from person A to person B.
    - Let's say the database here is mysql, so it will support ACID operations.
    - I will just run one transaction and in that entire transaction I will check that:
        - If A's balance is greater than 10
        - Then subtract 10 from it
        - Add 10 to B's balance
    - You can write a sql query for that.

    - If there is one database where I have things in tables, as sql databases support ACID, because of isolation this would work.
    - It happens atomically, so there is no chance of an inconsistency.
    - If you can recall from your sql classes, that if everything is in one single sql database then we can guarantee atomicity and consistency.
- Now the problem is when this A to B transfer is happening when the accounts are in different banks.
    - Let's say A has an account in Axis bank while B has SBI, A wants to transfer 1000 Rs. to B.
    - Here you have two different systems / services.
    - Both banks have separate systems, however you are required to do a transaction across these two systems.
    - *How do we do it so that it's consistent, that is the problem statement to solve.*
    - Here we have taken examples of two different companies altogether, however even within the same company this problem can occur.
        - Let's take the example of flipkart, it has multiple microservices like Orders, Inventory, Payments etc.
        - It is possible that when you place an order, we need to add the order in the Orders service but we also need to subtract it from the available Inventory.
        - Then we have to make the payment and after that there is Shipping microservice.
        - Here also we are doing a transaction across multiple services and every service has its own codebase, database.
        - *So how do we make sure that the transaction happens reliably across multiple services?* That's the problem statement we need to solve.

Two Phase Commit

- Imagine I have two services:
    - Order service
    - Inventory service
- Now let's say somebody has ordered an iphone 15, now I have to create an entry for that with address and a bunch of other things.

    - At the same time I have to also update here that for the iphone 15 inventory, if you had 100 items before then reduce it to 99.
    - So we have to do these two operations at the same time, that's basically a transaction.
    - However these are two different systems, they have their own database.
- Two phase commit is a pattern where you say that I will try to execute a distributed transaction, basically I will try to simulate a transaction over the network.
- How do we do that ?
    - Step 1: *Are you ready?* First step is to check with both systems, if they are ready for the transaction on the entry corresponding to iphone15. In other words, you're asking both services to take a lock on the corresponding row/entry, so that all other reads and writes are blocked for iphone15.
    - Step 2: After taking the lock on it, now *execute* the update operation on both systems.
    - Step 3: Once both services have confirmed step 2 is complete, then release the lock. Basically, ask both services to release the lock.
- This is called a two phase commit. However there are some downsides to it:
    - If you are having a long living transaction, then you will have the lock for a very long time.
        - This eventually makes everything slower. Every read and write query for the same entry will be stuck behind a lock for a long time, thereby making the request slow.
        - This will degrade the user experience.
- *Hence microservices consider 2 phase commit as an anti pattern.*

SAGA Pattern

- If a two phase commit is anti pattern then how do I do transactions in microservices ?
- SAGA pattern is one of the principles that microservices uses to keep doing distributed transactions.

SAGA Steps:

- Step 1: Break down the entire transaction into steps. Each step should be doable inside a single microservice. The second step should happen in another microservice.
- Step 2: Execute steps. On success, ensure all steps were executed. On failure, either retry or rollback reliably all earlier steps.

Example: Let's see through the banking example.

- Imagine person A from Axis bank wants to transfer money to person B who is from SBI bank.
    - Step 1: Firstly we would check if the balance of A is greater than or equal to 1000 Rs. If yes, subtract 1000 Rs from A's balance.
        - If it is not then we show the error.
    - Step 2: Second step would be making a call to SBI to increase the balance of B by 1000 Rs.
        - If it is successful then great.
        - If it fails then we have to rollback all earlier steps.
        - Here every step is atomic, and if any step fails then we have a reliable way to rollback.
- How do we execute reliably?
    - For this we have two patterns which work for SAGA.

Orchestration pattern

- Basically you make one person (or service) the owner. One machine or one service is the owner.
- Imagine in a bank we have the transaction service, it is the responsibility of the transaction service to make sure all steps are executed.
- Hence the transaction service becomes our orchestrator.
- It is the responsibility of it to make sure that every step is called by the right microservice, and if a step fails then it would retry or rollback the other steps.

Choreography pattern

- It is based on event driven architecture.
- It means there is no single orchestrator.

Rather, every service cares about doing the write atomically within the service. Once, done, it publishes an event to a persistent queue like Kafka. All other services that care about that write would subscribe to the event. As soon as they get notified, they start their own corresponding write.

Let's take an example. Imagine, someone places an order on Flipkart. So, the request goes to the orderService. OrderService stores all the details. Post that, it publishes an event to Kafka "new_order_created" along with details of the order (user_id, name, payment details, item_details, price of items, etc.).

Now, all services which depend on a new order being created should subscribe to the event "new_order_created". For example, InvoiceGenerationService, ProductService (to reduce available inventory), NotificationService.

They all can read in parallel, and execute their own transaction. They all in turn again can publish their own event when done (for example, "new_invoice_created", OR "inventory_updated" OR "user_notified"). If those events cause more writes, then those services should subscribe to those events.

How is failure handled? Just like an event is published on success, very similarly a different event is published when there is permanent failure. For example, "invoice_generation_failed". Microservices affected by that failure should subscribe to the failure event and rollback their transaction.

How would retry happen on failure Good thing with persistent queue is that it's easy to manipulate for retries. Kafka maintains an offset for every topic+partition. As long as you do not send acknowledgement back, offset is not increased. Only send ack once the event is successfully processed.

In a way, the transaction is executed as a coordinated set of events, which is exactly what SAGA means.

Choreography vs Orchestration

- Pros and Cons of both, it seems that choreography pattern is better and is more reliable,like there cannot be a machine failure which leads to any form of inconsistency which is true for most part.
    - However one of the things to keep in mind is that any form of event driven architecture end to end is going to be slow.
    - It is because when I complete a job and publish an event, now this event getting to the consumer is not instant.
        - For example consider two cases, in the first we have machine A which makes a direct call to machine B, while in other case machine A publishes an event to kafka and machine B subscribes to this event.
        - The second one is going to be slower, because there is an additional hop.
    - *In terms of speed, the orchestrator might be faster in comparison to the choreographer.*
    - *However, in reliability the choreographer is better because the orchestrator machine may die and then rollback becomes a challenge.*
    - *In the case of a central orchestrator, the observability is easier because at time of failure we can check logs on the orchestrator.*
    - It is also easier for us to check which service is slow and which one is faster.
    - However in case of a choreographer, we will be required to have a proper observability framework.

Kafka for Event driven architecture

Quick revision quiz. How do you scale the speed of publishing or consuming within a topic in Kafka?

Create more partitions. Each partition could be on a different Kafka machine (also called broker). Within a partition, ordering is guaranteed but not across partitions. So, choose partition_key wisely.

For example, for messages, recipient_id can be partition_key. So, all messages to a recipient X go on a single partition and there is ordering guaranteed in those messages. I do not care about ordering with someone else's messages.

Also, instead of a single consumer, if I use a consumer group (collection of machines), Kafka distributes partitions between each machine in the consumer group. This way, even the consumer gets parallelised and the speed of consumption increases.

Tip: Go over the Kafka class and notes in detail.

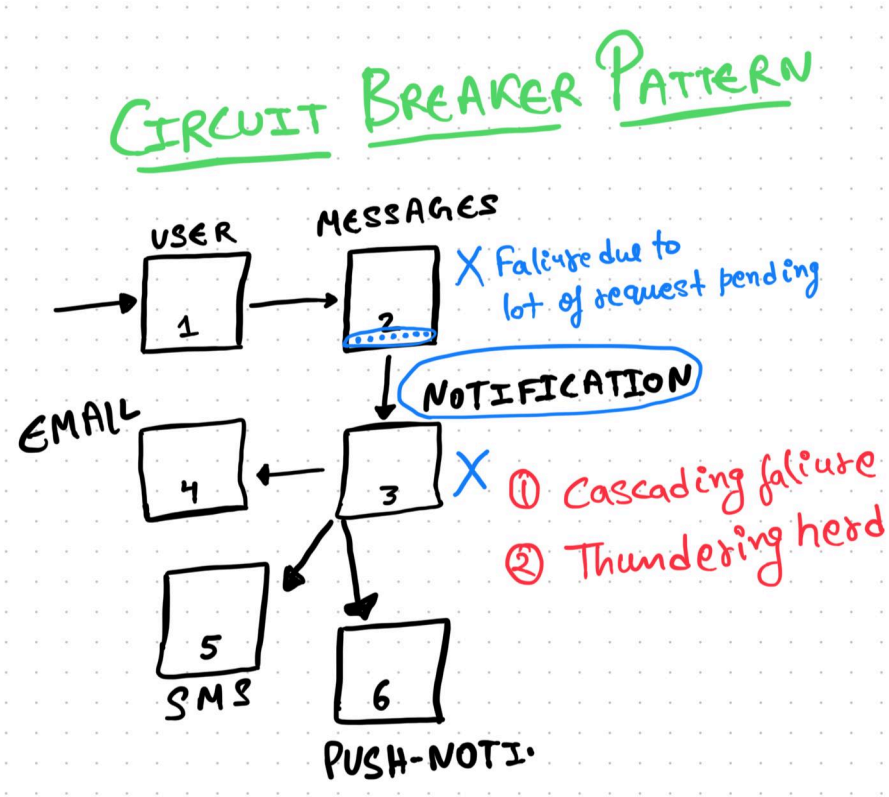
CQRS Pattern

- CQRS translates to command-query-request-segregation.
- Let me first tell you, two problem statements and the solution to both problems statements is going to be CQRS.
- First, let's say there are two microservices:
    - Messages - all my messages comes here.
    - Notifications - all my notifications come here.
- Imagine this is for facebook, it has both user and pages.
    - Just like users, pages can also get notifications and messages.
    - You can message a page / business.
    - Pages are very small portion of total userbase.
    - For pages you wanted to know:
        - How many notification does a page get on average ?
        - How many messages does a page get on average ?
        - Which are the pages which are getting more messages but less notifications ?
    - There are use cases for which you want to do a join on a small portion of data. Like here for pages you want to do a join across messages and notifications.

    - For microservices there is no way of doing join across 2 microservices. Service 1 cannot go and directly query the database of service 2 [Anti-pattern]. That is absolutely not allowed.
    - If you do so, then it is a anti-pattern. As it is violation of seperation of concern.
- *Problem statement 1: How do I do the join across two different microservices ?*
- CQRS pattern solves this exact problem, it says that whenever there is this kind of situation may be you look at command which is all of your update information:
    - Create
    - Update
    - Delete
- So all of your write operations, you consider them separately, all of you read operations (query), consider them separately.
    - Basically segregate those two.
    - Like for updates for each of the services let them come separately to corresponding services. This is to say addMessage goes to Messages Microservice, and addNotification goes to Notification Microservice.
    - However let's create another database which is outside of these two microservices, maybe it is a third microservice.
    - For the small portion of information that you want to do join across. We start publishing those to this microservice.
        - How do you reliably publish it to another microservice? May be you can publish it to kafka and this Microservice subscribes and consumes from Kafka?
        - Or you can write some database hooks or some scripts.
    - So now this db has information about both messages and notifications for just pages.
    - None of the writes are directly coming to this db, they are coming to their original microservices but they are being relayed to this db.
    - And now for all read operation, I only talk to this new third Microservice. It already has combined information, so getting joined information is easy. Those read can easily handled by this microservice. .
- This is CQRS pattern, it is nothing but just separate out the read operations from the write operation for a specific use case.

Circuit Breaker Pattern

Problem Statement



- Problem statement, We have the several microservice: user, message, notification, sms, email, push-notification.
    - Imagine some problem happens in the notification service. If it goes down, it poses some risks:

        - Requests come to user service -> message service -> notification service, but the third step keeps timing out.
        - The number of request waiting for notification service to respond keep increasing, ultimately increasing the load to message service. Lots of queued up IO requests.
    - It would be also lead to failure of my message service, and if my message service is down then similarly my user service comes at the risk. So first risk is cascading failure.

Another risk is as follows. Imagine, notificationService is trying to recover. Whatever was causing issues on the machine has been fixed. However, these machines have a large backlog of requests and they keep getting bombarded with new requests. In a way, recovery of NotificationService becomes really slow because it is constantly being bombarded by new requests. There is called as thundering herd.

- So both problem happen if a complete microservice is degraded.
    - Reason could be you pushed a bad code.
- *What can you do so that when microservices are not doing well, when they are unhealthy, can you built some logic to help them recover and also make sure that other microservice may not die ?*

The Solution

To tackle this in microservices, we have circuit breaker pattern. Following is broadly how it works.

- Step 1: Break down all possible errors into Regular Failures and Non-transient Failures. A system or a microservice being down should cause errors like timed out / connection reset / server error. However, a user not found OR request not properly formatted error does not mean the microservice is down.
    - *Regular failures:* If my application is running properly then what are the failures that I do expect like user not found, etc.
    - *Non-transient failures:* like timeout, connection reset, internal server error, are not regular failures, these occur when machine are in degraded state or there is a serious bug.

- Step 2: Figure out when a microservice is down vs recovered.
    - How do we do that though? I could setup a failure threshold. For example, if the number of failures from the microservice in the last 1 min exceeded 100 (threshold), then the microservice is down.
    - Ok, but where do I track the number of failures?
        - Maybe, in the global cache of the parent microservice (the microservice calling the affected microservice).

We are broadly there. The circuit breaker pattern maintain three states of a microservice:

- *Closed:* It means service is healthy
- *Open:* It is completely unhealthy, do not send even a single request. Give it some time to recover.
- *Half open:* Let's send a small percentage of requests and measure success rate. If it exceeds a threshold, change the state to "closed".

So by default, a microservice starts in a closed state. If we suddenly see that the number of failures are more than the threshold, then we change the state to the open state.

However every T seconds, the state is changed to half open. This is so that I can send some requests and see what's the success rate there. In a way, test if the service might have recovered.

- If the success rate is good, then we shift to closed and if not then we again go back to open state.
    - The above process will keep continuing.

Think of the above as the following pseudocode.

```
curr_state = redis.get("___")
if curr_state == "open":
    # do nothing / call the fallback function / fail
```

```
else if curr_state == "closed":
    try:
        response = notifications.get_cluster(___).get...

    except (exception):
        cnt = redis.get("curr_failure_rate")
        if cnt >= threshold:
            redis.set("___","open")
            redis.update("count", cnt)
else
    # curr_state = "half-open"
    if rand() % 100 == 0: # for sample 1% request
        try:
            response...

        except(exception):
            ...
    else:
        # do nothing / call the fallback function / fail
```