Data management patterns in microservices architecture

Giovanni Jiayi Hu
Department of Mathematics
University of Padua, Italy I-35121
Email: giovannijiayi.hu@studenti.unipd.it

***Abstract*—The microservices architecture is an approach to developing an application as a set of small independent services, which have their own lightweight stack and communicate with other similar services via well-defined interfaces. A viable strategy to achieve loose coupling is for each service to have its own data store. However, this kind of architecture introduces a new range of obstacles. We must, indeed, implement transactions that work across multiple distributed services. An operation that spans services must use what's known as a saga, a message-driven sequence of local transactions, to maintain data consistency. Unfortunately, sagas lack the isolation feature of traditional ACID transactions. As a result, an application must use countermeasures to prevent or reduce the impact of concurrency anomalies caused by the lack of isolation. On top of that, writing queries in a microservices architecture reveals to be likewise challenging. Implementing queries in an existing monolithic application is relatively straightforward because it has a single database, but in a microservice architecture, queries often need to retrieve data that are scattered among the databases owned by multiple services. This paper provides an overview of microservices patterns which emerge when adopting independent data stores, along with some suggested practical implementations.**

I. Introduction

A microservice is as a small self-contained application that has a single responsibility, a lightweight stack, and can be deployed, scaled and tested independently [1]. Services run in their own remote processes and communicate with each other using a lightweight mechanism such as message-queuing, which is discussed later.

The central aspect of the microservice definition is independence, the ability for a service to change without affecting any other. As such, services are independently deployable and scalable in addition to offering better fault isolation than the monolithic architecture. For example, a memory leak in one service only affects that service, while the other ones continue to handle the requests as usual.

The database architecture must then respect the following requirements to achieve this praised independence:

- Services must be loosely coupled so that they can be developed, deployed and scaled independently;
- Some business transactions must enforce invariants that span across multiple services. For instance, before placing an order, you must verify that a new order will not exceed the customer's credit limit;
- Some business transactions need to query data that are owned by multiple services. For example, displaying the orders require querying the Orders service to gather the cost and the Restaurant service to know if an order has been accepted;
- Different services might have different data storage requirements. For an Orders service, a relational database could be the best choice, but other services might need a SQL database with extensions for non-relational features, such as geospatial datatypes, or even a NoSQL database. A service which provides the nearest restaurants requires geospatial queries probably.
- Databases must support replication and sharding to scale. Replication improves the application availability and performance whereas sharding allows the application to handle increasing data volumes;
- Databases must contain no business logic, such as stored procedures. In doing so, databases for different services can more easily be combined and separated with minimal code changes [2].

A standard solution to these requirements is to keep each microservice's persistent data private to that service and accessible only via its API. The transactions of a service involve only its database. Using a database per service allows the services to be loosely coupled and guarantees higher resilience. Changes to one service's database do not impact any other services. Besides, each service can use the type of database that is best suited for its needs. For example, a service that handles text searches could use ElasticSearch [3].

However, using a database per service introduces severe drawbacks which must be carefully evaluated: implementing business transactions that span across multiple services is complex, and implementing queries which join data from several databases is challenging.

II. Queries

Three problems are commonly encountered when implementing queries in a microservice architecture:

1) To perform a single task, a client may have to make multiple calls to various services, which result in high latency and poor user experience. This chattiness between a client and different services can adversely impact the performance and scale of the application;
2) Retrieving data scattered across multiple services can result in expensive, inefficient in-memory joins without a proper solution;

3) The service can store the information in a database that doesn't efficiently support the required query, for instance, traditional relational databases are not efficient for geospatial queries.

A. API composition pattern

The most straightforward approach is the API composition pattern [4], which implements a query operation by invoking the services that own the data and combining the results. As an example, a query operation to show the details of an order might invoke three services - `Order Service`, `Delivery Service and Payment Service` - and return aggregate the results.

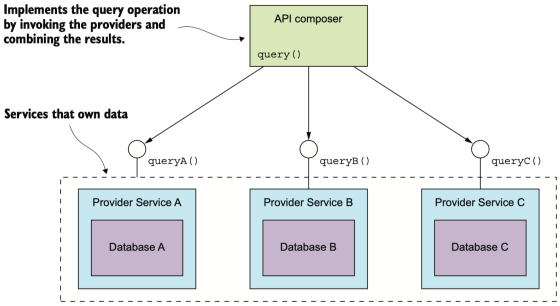

Fig. 1. The API composition pattern implements an operation by querying the service providers and combining the results [4].

The role of API composer is usually taken by a service such as an API gateway, in which case the pattern is also called Gateway Aggregation [5]. The latter approach requires developing your own API gateway with a framework or library, instead of just configuring static files using an off-the-shelf API gateway such as Ambassador for Kubernetes[1].

The API composition pattern can reduce the number of requests made to the services and improve application performance over high-latency networks, especially when caching techniques are applied.

However, the pattern has several drawbacks. It can result in a very inefficient in-memory join of large datasets and it requires more computing and network resources, increasing the cost of running the application. Moreover, there is a risk that a query operation returns inconsistent data since it executes queries against multiple databases.

Lastly, many queries can't be implemented using this pattern. Image an operation which retrieves a consumer's order history. In contrast to replying with the details of a single order, it returns multiple orders with details from different services. Requesting orders from the services by ID is only practical if those services have a bulk fetch API, which is however unlikely. Individually requesting orders will, on the other hand, be inefficient because of excessive network traffic.

On the implementation side, an API composer should call provider services in parallel to minimise the response time for a query operation. Sometimes, though, an API composer needs the result of one service to invoke another one. The logic to efficiently execute a mixture of sequential and parallel service invocations can, therefore, be complex.

For an API composer to be maintainable as well as performant, it could use a reactive design based on Reactive streams (explained later) or some other equivalent abstraction, such as Futures or Promises depending on the programming language [6]. Reactive streams and futures/promises make it possible to compose asynchronous operations or define a pipeline of operations to be invoked upon completion of the computation. This approach is in contrast to callback-heavy or more imperative direct blocking approaches.

Ultimately, whether you can use the API composition pattern to implement a particular query operation depends also on how the data is partitioned and the capabilities of the APIs exposed by the services that own the data.

B. CQRS pattern

In contrast to the API composition pattern, the Command Query Responsibility Segregation (CQRS) pattern [7] is more powerful as we shall explain, but it is also more complex. It maintains one or more view databases whose sole purpose is to support queries.

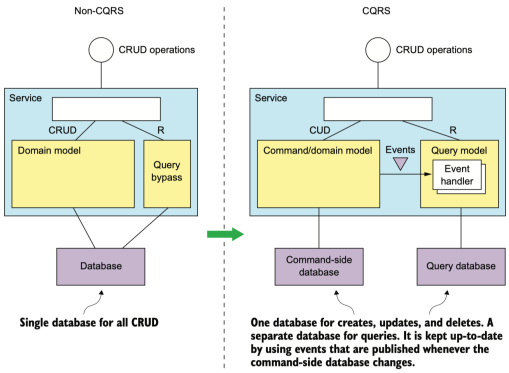

Fig. 2. On the left is the not-CQRS version of the service, and on the right is the CQRS version [9].

Command Query Responsibility Segregation, as the name suggests, is all about segregation, or the separation of concerns. As figure 2 shows, in a not-CQRS service, the create, read, update and delete (CRUD) operations are typically implemented by a domain model that's mapped to a database. An Object-Relational Mapping (ORM) [8] is a common technique in these cases, although, a few queries might bypass the domain model and access the database directly for performance reasons. On the other hand, a CQRS service splits a persistent data model and the modules that use it into two parts: the command side and the query side.

- The write side (or command side) implements create, update, and delete operations (CUD) managing its own database and ensuring business rules. It may also handle simple queries, such as non-join, primary key-based

[1]https://www.getambassador.io/

queries. The command side publishes events whenever its data changes;
- The read side (or query side) is separate to the write side, which means different object models, probably running in different processes, perhaps on separate hardware. The query side handles only read operations. It subscribes to events produced by the write side and uses them to build and maintain a model that is suitable for answering client queries. Consequently, it uses whatever kind of database makes sense for the queries that it must support. The read side can be seen as an implementation of the Reporting Database pattern [10].

CQRS enables the efficient implementation of queries that retrieve data owned by multiple services, especially compared to the API composition pattern, which sometimes results in expensive, inefficient in-memory joins.

Furthermore, separation of the read and write sides allows each to be scaled appropriately to match the workload. For example, query sides typically encounter a much higher demand than the write part.

CQRS also allows an application or service to implement a diverse set of queries efficiently. Attempting to support all queries using a single persistent data model is also often challenging and in some cases, impossible. Even when a relational database has extensions to support a particular kind of query, using a specialised database is often more efficient. The CQRS pattern avoids the limitations of a single database by defining one or more read sides, each of which efficiently implements specific queries.

Another benefit is that CQRS defines separate code modules and database schemas for the command and query sides of a service. By separating concerns, the command side and query side are independent of each other and likely to be simpler and easier to maintain. The write side database can be refactored without needing to change to read databases.

Even though CQRS has several benefits, there is a significant additional operational complexity of managing and operating the extra data stores, which might even be of different types. Besides, as you might expect, there's a delay between when the command side publishes an event and when the query side processes that event and updates its read model. A client application that updates a model using a write operation and then immediately queries a view may see the previous version of the data. It is likely though that the service has been designed with eventual consistency, discussed in the next section about distributed transactions.

On the implementation side, applying the CQRS pattern does not mandate that you use different data stores for write and read sides, or that you use any particular persistence technology such as a relational database, NoSQL store, or event store. Nevertheless, a NoSQL database is often a proper choice for a CQRS read side, which can leverage its strengths like the ease of scaling and ignore its weaknesses. A CQRS view is unaffected by the limitations of a NoSQL database because it only uses simple transactions and executes a fixed set of queries without the need for relational joins. Actually, you don't need to normalise the database, because it is read-only and the records can duplicate data if needed to make queries easier.

Although CQRS does not require messaging, it is common to use messages to process commands and publish update events. Messages also allow the CQRS write part to handle commands asynchronously from a queue, rather than being processed synchronously, making the service itself asynchronous.

In that case, the application must handle message failures or duplicate messages. A command or event handler should be idempotent whenever possible, that is duplicate events result in the same correct outcome. For instance, an event which cancels an order can be repeated multiple times with the same result. In other cases, events can be structured differently to be idempotent. Assume the following structure for a `ThumbsUp` event.

```
{
  "likingUser": 2,
  "restaurantId": 1
}
```

A service could be subscribed to `ThumbsUp` event and increment a thumbs-up counter for a restaurant, but if the same message arrives twice, then the counter is incremented twice as well. The event structure can be changed as follows to solve this problem:

```
{
  "likingUser": 2,
  "restaurantId": 1,
  "totalLikes": 320
}
```

Thanks to that, no matter how many times the event arrives, the resulting state is the same as only one copy arrived.

Unfortunately, idempotent events make your system vulnerable to simultaneous events. Two `ThumbsUpEvent` could be published at the same time by two different users; however, they carry the same `totalLikes`. One of the two events will be overridden. Out-of-order events are likewise troublesome: two `ThumbsUpEvent` events could be published sequentially with `totalLikes: 320` and `totalLikes: 321`, but if they arrive in the wrong order, then the system remains inconsistent.

As demonstrated, an idempotent event is not feasible most of the time. A non-idempotent event handler must instead detect and discard duplicate events by recording the ids of events that it has already processed. One viable solution is to keep the history of IDs of all events that they have already seen. Theoretically, a CQRS read side should keep a full history of events, but in practice, the last couple of hours should be sufficient.

Ultimately, CQRS provides many benefits when it comes to the scalability of data queries. Unfortunately, it doesn't come for free, and the attached costs may be severely underestimated, causing a significant decrease in productivity.

C. Reliable events

It's essential for the database update and the publishing of the event to happen atomically. Consequently, to communicate reliably, the CQRS services must use transactional messaging. Otherwise, a service might update the database and then crash, for example, before sending the message. If the service doesn't perform these two operations atomically, a failure could leave the system in an inconsistent state.

A straightforward way to reliably publish messages is to apply the Transactional outbox pattern. This pattern uses a database table as a temporary message queue, where the service inserts messages. Atomicity is guaranteed because this is a local ACID transaction. Then, an effortless way to publish the messages is to poll the table for unpublished messages although frequently polling the database can be expensive. Also, whether you can use this approach with a NoSQL database, which has no tables by definition, depends on its atomic update capabilities. For instance, MongoDB write operations are atomic only on the level of a single document [11].

A sophisticated solution is to tail the database transaction log (also called the commit log). Every committed update made by an application is represented as an entry in the database's transaction log. A transaction subscriber can read the transaction log and publish each change as a message to the message broker [12].

The challenge lies in writing low-level code that calls database-specific APIs. It's usually preferable using an open-source framework such as LinkedIn's Databus[2], a distributed source-agnostic change data capture system that publishes changes made by a MySQL, Postgres or MongoDB database to a message broker.

Typical Databus deployments consist of a cluster of log miners called Relay servers that pull change streams from the data sources [13]. The relays are set up in such a way that the change stream from every data source is available in multiple relays, for fault-tolerance and scaling. There are two configurations of the relays that are typically deployed.

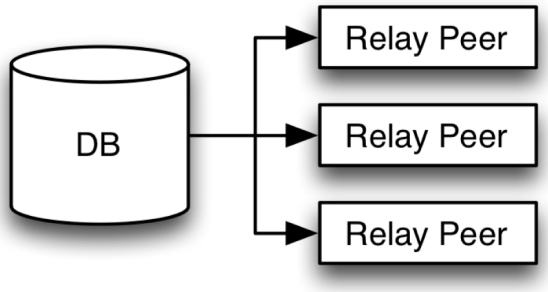

Fig. 3. Databus deployment with independent Relay servers.

In one deployment model, as shown in Figure 3, all the Relay servers hosting a stream connect to the stream's data source directly. Each server is assigned a subset of all the streams. When one of them fails, the surviving Relays continue pulling the change streams independent of the failed one, providing 100% availability of the streams at very low latency. This model, however, comes as the cost of increased load on the data source server.

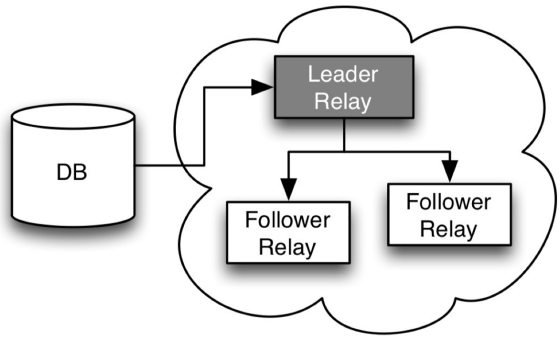

Fig. 4. Databus deployment with Leader-Follower model.

To reduce the load on the source data source server, an alternative deployment model for Relays, as shown in Figure 4, is the Leader-Follower model. In this model, for every data source, one server is designated to be the leader while the other ones are designated to be followers. The leader connects to the data source to pull the change stream while the followers pull the change stream from the leader. The clients can connect to any of the Relays, either leader or follower. If the leader Relay fails, one of the surviving followers is elected to be the new leader. This deployment drastically reduces the load on the data source server, but when the leader fails, there is a small delay while a new leader is elected. During this window, the latest changes in the change stream are not available to the consumers.

III. Distributed transactions

ACID (Atomicity, Consistency, Isolation, Durability) transactions greatly simplify the job of the developer by providing the illusion that each transaction has exclusive access to the data. In a microservice architecture, a single service can still use ACID transactions for local operations. The challenge, however, lies in implementing transactions for operations that update data owned by multiple services, each of which has its own database, without violating consistency.

By consistency, we mean state consistency: a transaction preserves all the database rules like referential integrity or, for example, when transferring money from one account to another the total amount held in both accounts should not change. It should not be confused with the copy consistency in the CAP theorem [14], which refers to having all the copies of the data up-to-date and it's a subset of the former consistency.

The traditional approach to maintaining data consistency across multiple services or databases is to use distributed transactions. The de facto standard for distributed transaction management is the X/Open Distributed Transaction Processing (DTP) Model [15], which uses two-phase commit (2PC) to

[2]https://github.com/linkedin/databus

ensure that all participants in a transaction either commit or rollback.

Several problems arise, however, when a 2PC protocol is used in a system where network and hardware failures occur, or a server could just crash. For instance, the coordinator, as well as the participants, have states in which they block waiting for incoming messages [16]. For that reason, the main problem with traditional distributed transactions is that they reduce availability. For a distributed transaction to commit, all the participating services must be available.

The overall availability is the product of the availability of all of the participants in the transaction. If a distributed transaction involves four services that are 99.5% available, then the overall availability is 98%, which is significantly less. Each additional service involved in a distributed transaction further reduces availability.

On top of that, the famous CAP theorem provides another proof of the unsuitability of distributed transactions:

> It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: consistency, availability, partition tolerance.
>
> Eric Brewer

The original theorem refers to copy consistency, but ACID consistency also cannot usually be maintained across partitions.

Although the formulation was misleading and has been revisited [17], it remains true that the CAP theorem prohibits perfect availability and consistency in the presence of network partitions. But the three properties are continuous values other than binary. Availability is already expressed as a range from 0 to 100%, but there are also many levels of consistency, and even partitions usually affect only portions of a system, not globally.

The modern CAP goal tries to maximise combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans to be operative during a partition to maintain availability and for recovery afterwards, to keep consistency.

As a matter of fact, many systems nowadays drop perfect consistency in favour of eventual consistency [18], availability and partition tolerance. Temporary inconsistency can be tolerated for two reasons: for improving read and write performance under highly concurrent conditions and for handling partition cases where a majority model would render part of the system unavailable even though the nodes are up and running. Partition recovery will need to restore ACID consistency since maintaining invariants during partitions might be impossible.

It is clear at this point that to solve the complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that builds on top of the asynchronous CQRS services. CQRS services are already partition tolerant since events and commands are usually implemented via asynchronous message queues which persist unprocessed messages. Furthermore, commands and queries are handled without the need for external services, thus improving availability. What is missing then is a mechanism to keep the consistency and to plan for recovery if a partition occurs. This is where the saga pattern comes in.

IV. The saga pattern

Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions. You define a saga for each system command that needs to update or validate data in multiple services.

A saga is a sequence of local transactions. Each local transaction updates data within a single service using the familiar ACID transactions [19].

A saga is initiated by a command event and it reacts to subsequent command events, generating new ones and thus allowing command handlers to be kept independent. The completion of a local transaction triggers the execution of the next local transaction.

While handling a request, the service saga doesn't synchronously interact with any other service. Instead, it asynchronously sends messages to other services which are not required to be available at the same time. Eventually, any unavailable service comes back up, and process queued messages. In other words, it ensures that all the steps of a saga are executed, even if one or more of the saga's participants is temporarily unavailable.

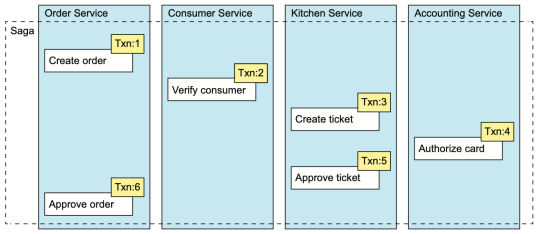

Fig. 5. Creating an Order using a saga [19].

The example saga used throughout this section is the `Create Order Saga`, which is shown in Figure 5. The saga's first local transaction is initiated by a command event to create an order. The other five local transactions are each triggered by command events signalling the completion of the previous one.

This saga consists of the following local transactions:

1) Order Service: it creates an order in an `APPROVAL_PENDING` state;
2) Consumer Service: it verifies that the consumer can place an order;
3) Kitchen Service: it validates order details and create a ticket in the `CREATE_PENDING` state;
4) Accounting Service: it authorises the consumer's credit card;
5) Kitchen Service: it changes the state of the ticket to `AWAITING_ACCEPTANCE`;

6) Order Service: it changes the state of the order to `APPROVED`.

An implementation issue which needs to be considered is ensuring that a saga participant is able to map each event that it receives to its own data. For example, when Order Service receives a `CreditCardAuthorised` event, it must be able to look up the corresponding saga. The solution is for a saga participant to publish events containing a *correlation id*, which is data that enables other participants to perform the mapping. For example, the participants of the Create Order Saga can use the `orderId` as a correlation id that's passed from one participant to the next.

A. Compensating transactions

A great feature of traditional ACID transactions is that the business logic can easily roll back a transaction if it detects the violation of a business rule. Unfortunately, sagas can't be automatically rolled back, because each step commits its changes to the local database.

Suppose that the (n + 1) th transaction of a saga fails. Thus, to abort the entire saga, the system must undo each already committed transaction by issuing a new transaction that corrects for its effects: the compensating transaction [20]. Conceptually, each of those steps has a corresponding compensating transaction which undoes the effects of the step. The saga must execute each compensating transaction in reverse order to undo the effects of those first n steps.

To see how compensating transactions are used, imagine a scenario where the authorisation of the consumer's credit card fails. In this scenario, the saga executes the following local transactions:

1) Order Service: it creates an order in an `APPROVAL_PENDING` state;
2) Consumer Service: it verifies that the consumer can place an order;
3) Kitchen Service: it validates order details and creates a ticket in the `CREATE_PENDING` state;
4) Accounting Service: it authorises consumer's credit card, which fails;
5) Kitchen Service: it changes the state of the ticket to `CREATE_REJECTED`;
6) Order Service: it changes the state of the order to `REJECTED`.

The fifth and sixth steps are compensating transactions that undo the updates made by the Kitchen Service and Order Service, respectively. It is important to note that not all steps need compensating transactions. Read-only steps, such as the verification of the order details, don't need compensating transactions.

A saga's coordination logic is therefore responsible for sequencing the execution of both forward and compensating transactions.

B. Lack of isolation

The I in ACID stands for isolation. The isolation property of ACID transactions ensures that the outcome of executing multiple transactions concurrently is the same as if they were executed in some serial order. The database provides the illusion that each ACID transaction has exclusive access to the data.

The challenge of using sagas is that they lack the isolation property of ACID transactions. That's because the updates made by each of a saga's local transactions are immediately visible to other sagas once that transaction commits. This behaviour can cause two problems. Other sagas can change the data accessed by the saga while it's still executing or they can read its data before the completion of all the updates, and consequently, the former sagas are potentially exposed to inconsistent data.

This lack of isolation can cause database anomalies. An anomaly is when a transaction reads or writes data in a way that it wouldn't if transactions were executed one at a time. When an anomaly occurs, the outcome of executing sagas concurrently is different than if they were executed serially.

The lack of isolation can cause the following three anomalies [21]:

1) *Lost updates*: one saga overwrites, without reading, changes made by another saga. As a result, the changes have been lost, and the system state is potentially inconsistent. For instance, a saga may start and later approve an order which has been already cancelled by another saga;
2) *Dirty reads*: a saga reads the updates made by a saga that has not yet completed. Those updates could be still inconsistent or will be reverted by a compensating transaction;
3) *Fuzzy/non-repeatable reads*: two different steps of one saga read the same data and get different results because another saga has made updates in between the two reads.

It is then the responsibility of the developer to adopt a set of countermeasures for handling anomalies caused by lack of isolation that either prevent one or more anomalies or minimise their impact on the business [22].

One example of countermeasure is the usage of semantic locks: a saga's compensatable transaction sets a flag in any record that it creates or updates. The flag indicates that the record isn't committed and could potentially change. The flag can either be a lock that prevents other transactions from accessing the record or a warning that indicates that other transactions should treat that record with suspicion.

A further example of countermeasure is to design the update operations to be commutative. Operations are commutative if they can be executed in any order, thus eliminating the possibility of lost updates. An account's debit and credit operations are, for instance, commutative. Unfortunately, using only commutative operations is tough. For example, account operations with a zero balance check are not commutative since you can deposit an amount of money and then withdraw it, but the opposite is not possible if the withdrawal would leave the balance below zero.

V. Communication

As we have seen with the previous patterns, the services within a microservice architecture must frequently collaborate in order to complete a saga or to handle a query request. A microservices-based application is a distributed system running on several processes or services, usually even across multiple servers or hosts. Each service instance is typically a process. Therefore, services must interact using an inter-process communication (IPC) protocol.

The choice of IPC mechanism is an important architectural decision since it can impact application availability, and it intersects transaction management.

Synchronous protocols such as REST [23] and RPC [24] are used mostly to communicate with other applications. Using REST is simple, familiar to most of the developers and it doesn't require an intermediate broker, which simplifies the system's architecture. However, the problem with REST is that it is a synchronous protocol: an HTTP client must wait for the service to send a response. Whenever services communicate using a synchronous protocol, the availability of the application is reduced as the overall value is the product of the availability of all of the participants in the communication. The same issue also applies to RPC.

As we have seen with the CQRS and saga patterns, services can communicate with one another using asynchronous messaging to improve availability and partition tolerance.

A service client makes a request to a service by sending it a message. If the service instance is expected to reply, it does so by sending a separate message back to the client. Because the communication is asynchronous, the client doesn't block waiting for a reply. Instead, the client is written, assuming that the reply won't be received immediately.

A messaging-based application typically uses a message broker, which acts as an intermediary between the services. A sender writes the message to the message broker, and the message broker delivers it to the receiver. An important benefit of using a message broker is that the sender doesn't need to know the network location of the consumer, decreasing the coupling between services. Another benefit is that a message broker persists messages until the consumer is able to process them, usually using message queues and thus improving partition tolerance.

A. Message Queuing

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues by using a message broker, an infrastructure service through which all messages flow.

A critical aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behaviour of the recipient. These semantics permit communication to be loosely coupled in time.

When selecting a message broker, you have various factors to consider, including the following [25]:

- Messaging ordering: does the message broker preserve ordering of messages?
- Delivery guarantees: what kind of delivery guarantees does the broker make?
- Persistence: are messages persisted to disk and able to survive broker crashes?
- Durability: if a consumer reconnects to the message broker, will it receive the messages that were sent while it was disconnected?
- Scalability: how scalable is the message broker?
- Latency: what is the end-to-end latency?

Each broker makes different trade-offs. For example, a very low-latency broker might not preserve ordering, make no guarantees to deliver messages and only store messages in memory. A messaging broker that guarantees delivery and reliably stores messages on disk like RabbitMQ [3] will probably have higher latency.

After picking the proper broker solution for your application, we can see an example of message communication by exhibiting how the saga orchestrator communicates with the participants using async reply-style interaction based on message queues. To execute a saga step, the saga sends a command message to a participant telling what operation to perform. After a saga participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.

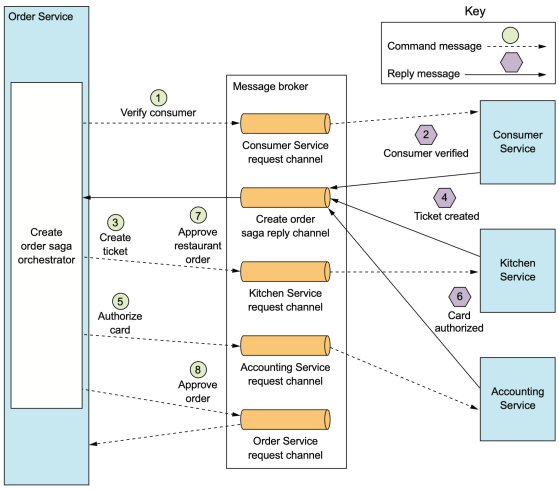

Fig. 6. Order Service implements a saga orchestrator, which invokes the saga participants using asynchronous request/response [19].

Messaging doesn't come without downsides. First, there is a risk that the message broker could be a performance bottleneck or a potential single point of failure, although fortunately, many modern message brokers are designed to be highly

[3] https://www.rabbitmq.com/

scalable and available. Second, the messaging system results in additional operational and infrastructural complexity.

Message-based communication is more likely to be unfamiliar compared to REST, and the developers have to face issues like duplicate messages or transactional messaging.

Ultimately, the two commonly used protocols are HTTP request/response for regular web APIs (when querying most of all), and lightweight asynchronous messaging when communicating updates across multiple microservices [26].

VI. Functional Reactive programming

Functional Reactive Programming (FRP) [27] is a paradigm for programming hybrid systems (systems containing a combination of both continuous and discrete components) in a high-level, declarative way. The key ideas in FRP are its notions of continuous, time-varying sequences of discrete events. Being able to define and manipulate continuous values in a programming language provides great expressive power.

In a microservices application, especially when the CQRS or the saga patterns are applied, it is common for a service to send messages to another one as soon as they are available and they are usually stored in a queue until processed. The latter service subscribes to the queue and it "reacts" as soon as some message has been placed on the queue. However, FRP goes further than that; otherwise, it would be just a normal Publish-Subscribe pattern [28]. The programming model sees everything as a stream of data, from the call to a database to the retrieval of a message from a queue. All is handled as a stream of time-varying values.

Frameworks like RxJS (Reactive Extensions for JavaScript) [4] were conceived centred around the concept of streams, and to provide combinators and operators to combine, create and filter any stream.

The following example shows the implementation of an API composition to retrieve the details of an order as a combination of streams.

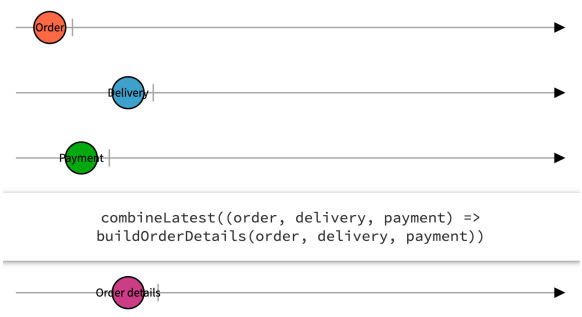

Fig. 7. Combination of streams to query the order details.

Each service request is a stream since it does not matter if a query operation has only one event, it is still a stream of data, and it is represented in the figure as a directed line. A vertical dash, if present, indicates the termination of a stream.

[4]https://rxjs-dev.firebaseapp.com/

When the API Gateway calls another service, it doesn't wait for the result, it merely subscribes for the result and continues its processing. In particular, the three provider services are called in parallel and thanks to the `combineLast` combinator, only when the result of all three of them has arrived, the API Gateway processes that result and dispatches it an event of the resulting stream.

The resulting stream terminates when all the three requests end and can be converted as a different structure, like the returning response, or even used as an input to another stream. Even multiple streams can be used as inputs to another one, which is actually what the example did. You can also filter a stream to get another one that has only those events you are interested in, which is especially useful in the CQRS read side or within a saga to check only events with a correct correlation id.

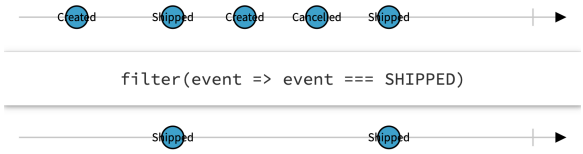

Fig. 8. Filtering a stream of order events to return a new stream made up of only SHIPPED events.

The functional style results in code that looks more declarative than imperative: instead of giving a sequence of instructions to execute, the FRP describes relationships between streams.

Functional Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic. The benefit is evident in microservices applications with a multitude of events related to data changes and saga transactions.

VII. Conclusion

We started the paper by exhibiting the reasons behind a database-per-service solution in a microservice architecture. Unfortunately, no technology is a silver bullet, and this solution has several significant drawbacks as well as risky complexity.

One issue is that developers must deal with the additional complexity of managing distributed databases. The goal of each microservice is to be independent and available to the client consumer, even if the other services that are part of the end-to-end application are down or unhealthy. Asynchronous communication between services offers a good compromise for this goal and makes the inter-process communication very explicit, so developers aren't misled into a false sense of security. It is though crucial to use an interprocess communication mechanism such as message queuing, since missing a message like a command event could indeed leave the system in an inconsistent state.

However, implementing use cases that span multiple services requires the use of unfamiliar techniques like CQRS

and sagas. One big obstacle that developers will have to face when adopting microservices is moving from a single database with ACID transactions to a multi-database architecture with ACD sagas, because they're used to the simplicity of the ACID transaction model.

All the patterns explored in this paper require a significant mental leap, so they shouldn't be tackled unless the benefit is worth the jump. Having a database-per-service might be needed on specific portions of a system and not the system as a whole, based on the different scaling/usage characteristics. In this way, some portions can use a shared database without encountering all the complexities seen before, while others benefit from a separated data store and the previous patterns. The former services may no be truly independent microservices, more just services in Service-Oriented-Architecture [29], but it is crucial for development teams to understand how to balance the cost and benefits of microservices.

REFERENCES

[1] Alberto Simioni, Tullio Vardanega, "In Pursuit of Architectural Agility: Experimenting with Microservices" in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018. [Online] Available: https://ieeexplore.ieee.org/document/8456408

[2] Randy Shoup, "The eBay Architecture: Striking a Balance between Site Stability, Feature Velocity, Performance, and Cost", pp. 21-22. [Online]. Available: https://www.slideshare.net/RandyShoup/the-ebay-architecture-striking-a-balance-between-site-stability-feature-velocity-performance-and-cost

[3] Chris Richardson, "Database per service" in "Microservice Architecture". [Online]. Available: https://microservices.io/patterns/data/database-per-service.html

[4] Chris Richardson, "Querying using the API composition pattern" in "Microservices Patterns", pp. 221-228, October 2018.

[5] Microsoft Developer Network, "Gateway Aggregation pattern" in "Cloud Design Patterns". [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation

[6] Kisalaya Prasad, Avanti Patil, Heather Miller, "Futures and Promises" in "Programming Models for Distributed Computing". [Online]. Available: http://dist-prog-book.com/chapter/2/futures.html

[7] Martin Fowler, "CQRS". [Online]. Available: https://martinfowler.com/bliki/CQRS.html

[8] Hibernate, "What is Object/Relational Mapping". [Online]. Available: http://hibernate.org/orm/what-is-an-orm/

[9] Chris Richardson, "Overview of CQRS" in "Microservices Patterns", pp. 232-233, October 2018.

[10] MongoDB, "ReportingDatabase". [Online]. Available: https://martinfowler.com/bliki/ReportingDatabase.html

[11] Martin Fowler, "Model Data for Atomic Operations". [Online]. Available: https://docs.mongodb.com/v3.0/tutorial/model-data-for-atomic-operations/

[12] Oded Shopen, "Listen to Yourself: A Design Pattern for Event-Driven Microservices". [Online]. Available: https://medium.com/@odedia/listen-to-yourself-design-pattern-for-event-driven-microservices-16f97e3ed066

[13] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh Balaji Varadarajan et al., "All Aboard the Databus! LinkedIn's Scalable Consistent Change Data Capture Platform". [Online]. Available: https://engineering.linkedin.com/research/2012/all-aboard-the-databus-linkedlns-scalable-consistent-change-data-capture-platform

[14] Eric Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed". [Online]. Available: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

[15] Microsoft Developer Network, "X-Open Distributed Transaction Processing Standard". [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms686548(v%3Dvs.85)

[16] Maarten van Steen, Andrew S. Tanenbaum, "Distributed Systems", pp. 483-490, February 2017.

[17] Eric Brewer, "Why 2 of 3 is missleading " in "CAP Twelve Years Later: How the "Rules" Have Changed". [Online]. Available: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

[18] Werner Vogels, "Eventually Consistent". [Online]. Available: https://queue.acm.org/detail.cfm?id=1466448

[19] Chris Richardson, "Using the Saga pattern to maintain data consistency" in "Microservices Patterns", pp. 114-117, October 2018.

[20] Eric Brewer, "Compensating for mistakes " in "CAP Twelve Years Later: How the "Rules" Have Changed". [Online]. Available: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

[21] Chris Richardson, "Handling the lack of isolation" in "Microservices Patterns", pp. 126-131, October 2018.

[22] Lars Frank, Torben U. Zahle, " Semantic ACID properties in multi-databases using remote procedure calls and update propagations". [Online]. Available: https://dl.acm.org/citation.cfm?id=284472.284478

[23] Roy Thomas Fielding, "Representational State Transfer (REST)". [Online]. Available: https://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm

[24] Andrew D. Birrell, Bruce Jay Nelson, "Implementing Remote Procedure Calls ". [Online]. Available: http://www.cs.princeton.edu/courses/archive/fall03/cs518/papers/rpc.pdf

[25] Chris Richardson, "Overview of broker-based messaging" in "Microservices Patterns", pp. 92, October 2018.

[26] Microsoft Developer Network, "Communication in a microservice architecture". [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture

[27] Zhanyong Wan, Paul Hudak, "Functional reactive programming from first principles". [Online]. Available: https://dl.acm.org/citation.cfm?id=349331

[28] Microsoft Developer Network, "Publisher-Subscriber pattern" in "Cloud Design Patterns". [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

[29] Microsoft Developer Network, "Service Oriented Architecture (SOA)" in "SOA in the Real World". [Online]. Available: https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx