Transactional Causal Consistency for Microservices Architectures

João Queirós
joao.miguel.queiros@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. The microservices architecture is a software engineering approach that structures an application as a set of loosely coupled services. Each microservice manages a small, cohesive, subset of the domain entities and can be implemented, deployed, and managed independently of other microservices. An implementation of a microservice may need to read data items that are managed by another microservice. This can be achieved by doing remote calls to perform the reads or by reading from a local cache of remote values, that is updated asynchronously using some form of publish-subscribe middleware. In any case, this can result in one microservice reading mutually inconsistent versions of remote data objects. Inconsistent reads can lead to unexpected states (also known as anomalies) and, ultimately, to undesirable results. Often, programmers are required to write code to identify the anomalies and compensating actions to correct them. In this work, we study middleware mechanisms that aim at ensuring that microservices always observe mutually consistent versions of remote objects, reducing the amount of compensating actions required. In particular, we focus on mechanisms that are able to offer transactional causal consistency to microservices.

Table of Contents

1 Introduction

The microservices architecture is a software engineering approach that structures an application as a set of loosely coupled services. Each microservice manages a small, cohesive, subset of the domain entities and can be implemented, deployed, and managed independently of other microservices. This approach makes it easier to evolve an application, as different teams may work and update microservices independently of each other. Each team can pick the programming language and the storage services that are most appropriate for each service, without being constrained by the choices made when implementing other services. Finally, when the application is deployed, microservices also ease the task of provisioning the necessary resources, because different resources can be easily assigned to different services. These advantages have driven many companies to adopt this approach when developing new applications and, in some cases, to refactor legacy monolithic applications as a composition of microservices [1].

Microservices also have a number of disadvantages. In a typical monolithic application, all modules share a single, common, storage system that supports transactional access. Also, each functionality of the application is executed as an atomic transaction, that is isolated from other concurrent invocations of the same or other functionalities. This ensures that, amongst other desirable properties, a functionality always has access to a consistent state of the data store while executing. This guarantee is often not provided to functionalities that execute in a microservice architecture. First, a given functionality may need to interact with multiple microservices. Because microservices are independent of each other, and can use different storage mechanisms, it is much harder to have the entire functionality executed as a single transaction [2]. Instead, in most implementations, the functionality is executed as a composition of multiple independent transactions (where each transaction involves a single microservice). This leads to a loss of transactional properties to the functionality as a whole, as there is no longer an atomic commit across all microservices, leading to a loss of isolation between concurrent executions [3].

Furthermore, even if a functionality only needs to interact with a given microservice, that microservice may need to read the value of data objects maintained by other microservices. This can be achieved by doing remote calls to perform the read operations or by reading from a local cache of remote values, that is updated asynchronously using some form of publish-subscribe middleware. Due to the asynchronous nature of updates, both cases can result in one microservice reading mutually inconsistent versions of remote data objects. Both the lack of isolation, and the possibility of reading mutually inconsistent values, can lead to unexpected states (also known as anomalies) and, ultimately, to undesirable results. Often, programmers are required to write code to identify the anomalies and compensating actions to correct them. The use of compensating actions is an inelegant solution, as it can be both complex and time-consuming for the developers to implement. This can lead to longer development times and increased maintenance costs, which ultimately might affect both the performance and the sustainability of the system.

In this work, we study middleware mechanisms that aim at ensuring that microservices always observe mutually consistent versions of remote objects. In particular, we focus on mechanisms that are able to offer Transactional Causal Consistency (TCC) to microservices. Informally, TCC is a consistency model that guarantees: i) the execution of a functionality reads from a causally consistent snapshot, which represents a view of the domain entities that includes the effects of all executions that causally precede it and ii) the execution of a functionality that updates multiple entities respects atomicity, i.e., all updates occur and are made visible simultaneously, or none does. The advantage of TCC over other stronger consistency models is that it can be implemented in a non-blocking way (namely, without requiring the use of locks or the execution of consensus amongst microservices). It can therefore be implemented without compromising the weak coupling amongst the implementations of different microservices. Having microservices observe TCC can reduce substantially the number of anomalies that need to be compensated.

The rest of the report is organized as follows. Section 2 summarizes the goals and expected results of our work. In Sections 3 and 4 we present the existing background in this field of work. In Section 5 we analyze and compare the related work exposed in the previous section. Section 6 describes the proposed architecture for the solution to be implemented. Section 7 describes the metrics we will be using to evaluate our solution. Section 8 presents the scheduling of future work and, finally, Section 9 presents the conclusions of this report.

2 Goals

This work addresses the problem of offering Transactional Causal Consistency to applications implemented following the microservices architecture pattern. More precisely:

> *Goals:* We aim at studying middleware mechanisms that can be applied to implementations of the microservices architecture pattern to ensure that the execution of a given functionality always observes a state that respects Transactional Causal Consistency. This means that, even if a functionality is composed of multiple transactions, executed in different microservices, all transactions that are part of the same functionality read from the same consistent snapshot, respecting causality and write atomicity.

TCC is a consistency criterion that has been proposed for systems that aim at offering high-availability. Due to this reason, most implementations avoid locking data items and, instead, tag data versions with metadata that allows to identify which versions belong to the same consistent snapshot. To ensure that microservice implementations remain loosely coupled, we also aim at avoiding locking data items. As a result, our middleware will also be based on keeping metadata associated to the object versions. Additionally, we would like to offer TCC transparently. For that purpose, we aim at building mechanisms to tag data

version automatically, exchange metadata transparently among microservices, and keep multiple object versions without requiring programmers to manage version explicitly. To better understand the tradeoffs involved, we plan to analyze the current work supporting transactions in microservices architectures, as well as understand the common data storage systems being used to support these operations in this architecture.

The project is expected to produce the following results.

> *Expected results:* The work will produce i) a specification of the coordination and metadata services; ii) an implementation of a middleware layer for a target microservices architecture, iii) an extensive experimental evaluation of the performance using the developed system.

3 Background

In this section we introduce the main concepts that are relevant to our work, namely we discuss the key properties of microservice architectures and how these features may interfere with the task of providing consistent results to clients. To help us in our exposition, we will use a simple application as an example.

3.1 Example: Tournament Management System

As an illustration, we will use a simple application to manage the schedule of tournaments. A tournament is an event that is characterized by a unique identifier, a number of players, and a date. Each tournament occurs at a given location, characterized by a unique identifier, a name, and an address. Figure 1 depicts the entities managed by the application. Figure 2 depicts the system's microservices, along with the supported methods. The application supports a number of functionalities such as *createTournament*, *deleteTournament*, etc, which are listed in Figure 3. Note that some functionalities only access a single entity (such as the *UpdateNumPlayers* that only accesses the entity *tournament*) while others, access multiple entities (such as the *ScheduleTournament* that accesses both *tournament* and *location*). The system contains a single invariant that must not be violated: there cannot occur more than 1 tournament in a location per day.

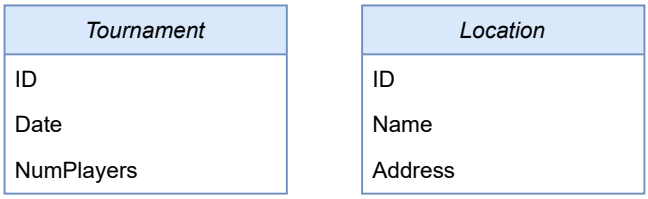
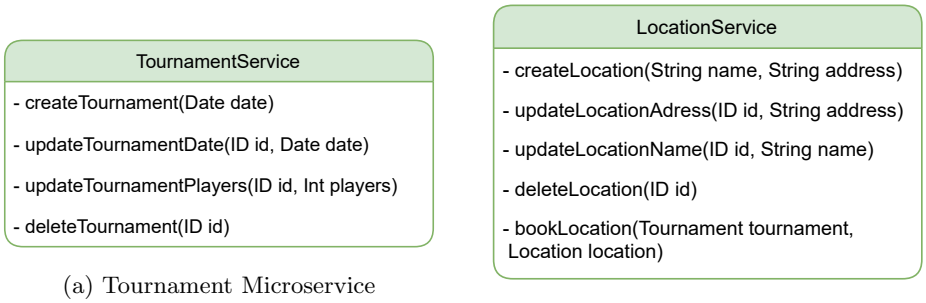


Fig. 1: Entities



(a) Tournament Microservice

(b) Location Microservice

Fig. 2: Microservices

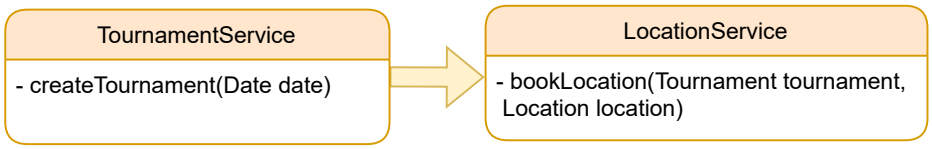
3.2 Monolithic and Microservices Architectures

Many applications are implemented today using a three-tier architecture [4]. This architecture includes a presentation tier that handles the communications with the client, an application tier that runs the business logic, and a data tier responsible for storing data, typically a database. In this section, we focus on the structure of the application tier.

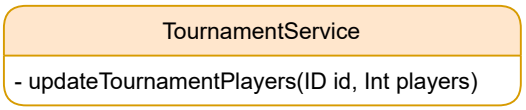
The simplest form to organize the business logic of an application is to structure it as a single executable software component, maintained in a shared codebase, that is deployed and provisioned as a whole. When the business logic is structured in this way, the application is said to follow a *monolithic architecture* [5]. This approach is suitable when the application is small, being developed and maintained by a small team. However, when the application grows, and more and more functionalities are added, the codebase tends to become extremely complex. Also, it becomes difficult to appropriately scale the resources assigned to the application, due to the fact that it is impossible to scale individually the resources assigned to each subcomponent of the application.

The microservices architecture is a design approach that consists in dividing the application into multiple loosely-coupled modules (or services) that can be developed, maintained, deployed and provisioned independently of each other. Microservices can simplify the growth of the applications, given that it allows different teams to be assigned to the development and maintenance of different services. Also, microservices make it easier to provision the right resources to the application, since it is possible to scale each service independently. Due to these reasons, the microservice architecture has gained an increasing acceptance.

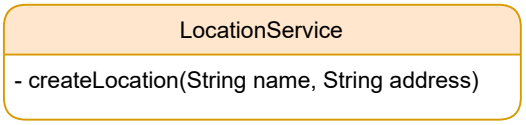
The microservice architecture raises a number of challenges, some of which we aim to address in our work. Given that monolithic applications typically use a single database, it is easy to implement functionalities as atomic transactions.



(a) ScheduleTournament functionality



(b) UpdateNumPlayers functionality



(c) CreateLocation functionality

Fig. 3: Functionalities

In a microservice architecture, different services often use different databases; although it is possible to execute functionalities that span multiple services as a distributed transaction, most microservice deployments opt to run those functionalities as a sequence of independent sub-transactions, breaking the isolation between concurrent executions. Also, in many microservice architectures, some services cache values of data items managed by other services, which creates additional opportunities for reading inconsistent data values.

Typically, in the microservice architecture, systems opt to follow one of two possible approaches to communication across microservices. These are *orchestration* and *choreography* [6]. Following the orchestration approach, microservices communicate with each other using a centralized service controller (also known as the *Coordinator*) that is aware of all the existing microservices. The controller directs each service to perform the intended function. For functionalities that span across multiple microservices, the coordinator listens to all events emitted by the services, triggering the following microservices in the functionality. Additionally, the coordinator is also responsible for handling both error and completion events. Large organizations such as Netflix use orchestration to coordinate their microservices, which in 2015 comprised over 700 microservices [7]. On the other hand, following the choreography approach, there is no need for any centralized service controllers. Microservices emit events that are directly subscribed by the following microservices. This process continues until no microservice emits any event. It is important to note that, following this communication approach, microservices do not need to know of each other's ex-

istence, allowing new microservices to be added to the system without needing to adapt existing microservices.

Our goal is to design mechanisms to improve the consistency guarantees offered to programmers of microservice applications while keeping the implementation of different services loosely coupled. We plan to integrate these mechanisms on systems following the orchestration approach.

3.3 Data Distribution in Microservices

In microservice architectures, for each domain entity, there is a single service that is responsible for performing updates on its data objects. Typically, these updates are performed by running transactions that are local to that microservice and, thus, updates to each item are atomic. However, microservices may need to read values from data items managed by other microservices. There are two main approaches to support such accesses: remote invocations and data replication.

When using the first approach, a microservice performs a remote invocation to another microservice to perform a read. This remote invocation is often implemented using a REST interface [8].

When using the second approach, a microservice keeps a local replica (or cache) of data items that are updated by remote microservices, but that are needed to execute local transactions [9]. Replicas of data items are updated asynchronously using some form of publish-subscribe service: when a microservice performs a local update to a data item, it publishes an event with the new value, which is subscribed by all other interested microservices.

By keeping a local replica of data items managed by other microservices, it is possible to execute all reads required to execute a transaction using local operations only, avoiding the delays associated with remote readings. Unfortunately, because replicas are updated asynchronously, local reads may return inconsistent values. We discuss the problem of consistency with replicated data in the following sections.

3.4 Consistency with Data Replication

In this section, we provide brief explanations over the different consistency models for data replication. An extended comparison between the different consistency models can be found on [10].

We will start by describing various single-object consistency models, that can be applied to transactions that execute on a single object. These transactions include both read and write operations.

One of the weaker consistency models that has been used in practice is *Eventual Consistency*:

Eventual Consistency (EC): this model allows for concurrent updates to be executed in different replicas and assumes that these updates are propagated asynchronously to the other replicas. This allows replicas to momentarily be inconsistent from each other, eventually converging to the same value if the

workload becomes quiescent. Concurrent updates may either be merged or, if this is not possible, select deterministically one of the concurrent updates and discard the others (using criteria such as *last writer wins*).

Eventual consistency permits executions where a client observes an update and, subsequently, in future access to another replica, no longer sees that update. Some of these behaviors can be prevented by using a stronger consistency model, such as *Monotonic Reads* and *Monotonic Writes*.

Monotonic Reads (MR): a consistency model that ensures that if a process performs a read r_1, followed by another read r_2, then r_2 cannot read a state of the object older than read in r_1.

Monotonic Writes (MW): a consistency model that ensures that if a process performs a write w_1, followed by another write w_2, then all processes will see w_1 before w_2.

Another consistency models that help systems maintain consistency in single-object scenarios are *Read Your Writes* and *Writes Follow Reads*.

Read Your Writes (RYW): As defined in [11], this consistency model ensures that, while considering two operations: a read r_1 and a write w_1 by the same process, if w_1 is executed before r_1, then r_1 must include the changes made by w_1.

Writes Follow Reads (WFR) also known as *session causality* implies that if a process executes a write operation w_1, and follows up by executing a read operation r_1 that included the changes made by w_1, then any future write operation w_2 must become visible after w_1.

By combining these four models' properties, we can derive a stronger consistency model – *Causal Consistency*, which ensures that the system maintains a consistent view of its data and that operations are ordered meaningfully.

Causal Consistency (CC): Steams from Lamport's definition of *happened-before* relation [12]. This relation (denoted: $\rightarrow$) is a partial ordering of events that reflects their causal relationship, such that if an event happens before another, the result must reflect that. Simply, if events a and b occur on the same process, $a \rightarrow b$ if event a precedes b. In another case, if an event a is an event that sends a message and event b is the event that receives this message, then $a \rightarrow b$. The formal definition for the happened-before relation can be found in [12]. CC embodies the idea of potential causality by linking successive operations inside a single process and operations that occurred in other processes but possibly became visible due to messaging mechanisms. Causal Consistency ensures that all processes see the same order of causally-related operations.

Despite providing stronger guarantees, *Causal Consistency* does not ensure that replicas converge to the same state under concurrent operations [13]. To guarantee this property, while maintaining the precise guarantees that the state applications observe, we define *Causal+ Consistency* as introduced in [14, 15].

Causal+ Consistency (CC+): an extension to *Causal Consistency*, that provides the extra guarantee of replica convergence under concurrent operations' scenario.

Despite providing a stronger guarantee, *Causal+ Consistency* only guarantees that operations are executed in a causally consistent order. *Linearizability* provides stronger guarantees, at the cost of needing more complex mechanisms.

Linearizability is the strongest single-object consistency model, that implies that if an operation A is completed before another operation B is started, then the effects of operation A should occur before operation B affects the object. This consistency model requires changes to be made atomically. Besides that, it extends *Causal+ Consistency* by ensuring that operations are consistent with the same real-time ordering as they took place.

We continue by exploring the different isolation mechanisms, capable of supporting *transactions*, i.e., sequences of read and write operations for multiple objects.

3.5 Isolation in Microservices

Ideally, one would like to have functionalities that are executed concurrently in a microservice architecture to be isolated from each other. The strongest isolation level is known as *strict serializability* (Strict 1SR), defined as:

Strict Serializability (Strict 1SR): The strongest consistency policy. As defined in [16], this model ensures that the execution of transactions take place atomically. That is, the sub-operations of a transaction do not appear to interleave with the sub-operations of concurrent transactions, leading to a sequential execution of the set of transactions. This execution must be one that is consistent with the real-time ordering of the transactions. Simply, if a transaction T_1 finishes before a transaction T_2 starts, then the transaction T_2 must be able to see the results of the transaction T_1 in the serialized order. Strict 1SR implies both serializability and linearizability.

However, for efficiency reasons, even in monolithic applications, weaker isolation levels are used, such as *serializability* (1SR), defined as:

Serializability (1SR) Similarly to Strict 1SR, Serializability is also considered a strong consistency policy. It offers the same guarantees as Strict 1SR, except for Linearizability's real-time constraints. In this way, as an example, if a process P_1 completes a write x, a following process P_2 is not guaranteed to observe the write operation performed by A. Serializability implies both Repeatable Reads and Snapshot Isolation.

A monolithic application can also offer *Snapshot Isolation* (SI), which is often the isolation level offered by default in many commercial databases:

Snapshot Isolation (SI): much like the other stronger consistency models, in SI transactions operate on an independent, consistent snapshot of the database. SI guarantees that all reads made in a transaction will always see the last committed values that existed in the database at the time the transaction started. This means that if a transaction T_1 writes an object obj_1, and a concurrent transaction T_2 commits a write operation to the same obj_1 after transaction T_1 began, that will cause T_1 to abort on commit time. Unlike serial-

izability, SI only enforces a partial order, that is, sub-operations of a transaction can interleave with sub-operations of other concurrent transactions.

It is possible to offer these isolation guarantees in a multi-database scenario by running distributed transactions. The X/Open XA eXtended architecture[17] is a standard that allows multiple databases to coordinate the execution of sub-transactions to achieve global isolation guarantees. However, running distributed transactions may create undesirable dependencies among microservices. For instance, to provide serializability, a transaction may lock data items in a given microservice until another microservice is ready to commit. To avoid this type of dependencies, most microservice deployments do not enforce full isolation among functionalities that span multiple services. Instead, a functionality is broken into multiple (sub-)transactions that are executed independently of each other.

It is worth noticing that it is possible to define weaker isolation levels among concurrent functionalities; some of these levels can be implemented while preserving a loose coupling among different microservices. These include *Repeatable Reads* (RR) and *Transactional Causal Consistency* (TCC):

Repeatable Reads (RR): The definition of Repeatable Reads is broad and can be ambiguous. For our work, we follow the definition of Repeatable Reads as defined in [18], which states that transactions read from non-changing snapshots, over the data items. This means that if a transaction reads the same data object multiple times, it will always the same value each time.

Transactional Causal Consistency (TCC): This is the strongest model a system can achieve under high-availability and low-latency [19]. TCC extends CC+ functionality. In this consistency model, transactions read from a causally consistent snapshot. This means that transactions read from a view of the data store that includes all the effects of the transactions that preceded it in the causal chain. As an example, consider a transaction T_1, that writes an object X_0 that depends on another object Y_0. Now suppose there is a running transaction T_2 that reads X_0. When T_2 reads the object Y, it must read a version that has not occurred before Y_0, due to X_0's dependencies. T_2 can either read Y_0, a concurrent version of object Y or a more recent version. Note that T_2 cannot read a version of object Y that depends on a newer version of X than version X_0 because T_2's snapshot already contains version X_0.

We will discuss the potential advantages of using weaker isolation guarantees in microservice architectures later in the report.

3.6 Anomalies

We also discuss common anomalies that can occur in these models. Each type of anomaly will be followed by a simple example, based on the scenario of Figure 2. By understanding the differences between each model and the pitfalls associated with each of them, it is possible to choose the right consistency mechanisms that answer the needs of our system.

The lack of isolation in microservices environments results from running functionalities as a sequence of independent transactions and the lack of consistency of read operations on remote entities. This allows the occurrence of operation

interleaving in ways that can never occur when functionalities are executed as an atomic transaction in a monolithic application, also known as *anomalies*. Here, we list some of the most relevant anomalies that may cause the functionality to yield incorrect results.

Dirty Reads: an anomaly that occurs when a transaction reads uncommitted data, created by another concurrent transaction. Considering the example presented in Figure 2, and Table 1, we consider a scenario where transaction T_1 creates a tournament X_1, using the *TournamentCreator* service. Concurrently, another transaction, T_2, is able to read the newly created tournament, before the writing transaction T_1 commits the changes to the database. In this case, T_2 reads dirty data.

Fuzzy Reads: in this anomaly, the same transaction reads different values for the same object at different times, resulting in inconsistent data. As an example, consider the scenario presented in Figure 2, and Table 2. A tournament X has been previously created and has been stored in the database as X_1. A transaction T_1 reads X_1, while a concurrent transaction T_2 updates the number of players participating in the tournament, writing X_2. T_1 continues its transaction by reading the tournament data again, but this time the data read is inconsistent with the first Read operation, as T_1 now reads X_2.

T1	T2
W(X_1)	
	R(X_1)
Commit	

Table 1: Dirty Reads

T1	T2
R(X_1)	
	W(X_2)
	Commit
R(X_2)	

Table 2: Fuzzy Reads

Fractured Reads: an anomaly that occurs usually associated with database shard replication and weaker consistency policies. As an example, consider the scenario presented in Figure 2, and Table 3. Transaction T_1 creates a tournament X_1 and books the location as Y_1. After that, transaction T_2 updates both the number of players participating in the tournament as X_2, and the location of the tournament as Y_2. We have a Fractured Read if a transaction T_3 reads the tournament with the updated number of players X_2, but on the older location Y_1. In this case, the result only captures partial transactional updates.

Lost Updates: an anomaly that occurs when two concurrent transactions read the same data and concurrently try to update it with different values. Following these events, one of the updates is lost as it is overwritten by the update done by the other transaction. Taking in consideration the scenario presented in Figure 2, and Table 4, transaction T_1 reads an existing tournament X_1. Concurrently, transaction T_2 increases by 5 the number of players participating in the tournament as X_2. Transaction T_1 then proceeds to increase by 2 the number of

players participating in the tournament. However, since T_1 only considers data read on X_1, the new update of X overwrites the update made by T_2.

Write Skew: an anomaly that occurs when two different transactions T_1 and T_2 concurrently update the entities in each other's read sets. For example, if a database guarantees serializability, then either T_1 executes first, preventing T_2 from achieving an unexpected state, or vice-versa. However, this is not the case if the database is under the SI consistency model.

As an example, consider the scenario presented in Figure 2 and Table 5. Consider that a Transaction T_1 reads all the registered locations in the system. Concurrently, a transaction T_2 reads all registered tournaments. Transaction T_1 proceeds by registering a new tournament on Location l_1. Concurrently, seeing that no tournament is currently registered in location l_1, Transaction T_2 deletes location l_1. Depending on the real order that these transactions are executed, we might observe a violation of the system, that is either registering a tournament on a non-existing location, or deleting a location that has active tournaments registered to it.

Real Time Violation: an anomaly that occurs when the execution of transactions does not respect the real-time order of the involved transactions. In a system that offers Serializability as the Isolation policy, if a process p_1 runs a transaction T_1 that executes a write operation w, we are not guaranteed that a subsequent process p_2 will be able to read the operation concluded by T_1. These real-time guarantees are only offered by Strict 1SR. As an example, consider the scenario presented in Figure 2 and Table 6. Consider that we have two transactions: T_1 updates the number of players participating in a tournament $t1$, while another process executes a transaction T_2, responsible for updating the date of the tournament $t1$. For Strict 1SR, we would have an anomaly if we couldn't guarantee the exact order by these two transactions occurred, according to Real-time ordering.

T1	T2	T3
W(X_1)		
W(Y_1)		
Commit		
	W(X_2)	
	W(Y_2)	
	Commit	
		R(X_2)
		R(Y_1)

Table 3: Fractured Reads

T1	T2
R(X_1)	
	W(X_2)
	Commit
W(X_1 + 2)	

Table 4: Lost Updates

T1	T2
R(Y_1)	R(X_1)
W(X_2)	W(Y_2)

Table 5: Write Skew anomaly

We finish this section by presenting a table that shows which anomalies are prevented by which consistency models, demonstrating the direct relation between consistency strength and isolation levels.

T1	T2
W(X_1)	
	W(X_2)

Table 6: Real Time Violation anomaly

	Dirty Reads	Fuzzy Reads	Fractured Reads	Lost Update	Write Skew	Real Time Violation
EC	🛡	✗	✗	✗	✗	✗
RR	🛡	🛡	✗	✗	✗	✗
TCC	🛡	🛡	🛡	✗	✗	✗
SI	🛡	🛡	🛡	🛡	✗	✗
1SR	🛡	🛡	🛡	🛡	🛡	✗
Strong 1SR	🛡	🛡	🛡	🛡	🛡	🛡

Table 7: Consistency Models and Isolation Levels: a comparison of Anomaly Prevention Capabilities. Shields represent protection against the anomaly, checkmarks represent vulnerability to the anomaly.

3.7 Addressing Consistency Problems in Microservices

As depicted in Table 7, only the strongest consistency model prevents the occurrence of all anomalies presented. Furthermore, the weaker the model, the more anomalies are allowed. Unfortunately, as we have also discussed, enforcing strong consistency in microservices is expensive and reduces the decoupling among multiple storage services. Due to the latter, many microservice implementations do not enforce strong consistency. Instead, functionalities are executed as a sequence of independent transactions that read from local (possibly inconsistent) replicas of data items maintained by other data services. As a result, programmers have to deal explicitly with anomalies and with the lack of atomicity in the executions of functionalities that span multiple services [20].

Sagas[21] is a design pattern that helps programmers to structure the code in a way that helps in mitigating the effects of anomalies and lack of atomicity. This pattern considers the inclusion of *compensating actions* that aim at re-establishing the consistency of an application when an anomaly occurs. For instance, consider the ScheduleTournament functionality of Figure 3. Consider that a tournament is created at the *Tournament* service but that no location is available when booking takes place; in this case, a compensation action could delete the tournament, to avoid having tournaments without an associated location.

Our work is motivated by the observation that the number of anomalies that need to be addressed by the programmer (via the implementation of appropriate compensating actions) can be reduced if the runtime can offer some minimal consistency guarantees (such as TCC), even when the functionality is split into multiple independent transactions. We hypothesize that consistency

criteria such as TCC can be offered without breaking the decoupling among the implementation of different microservices.

4 Related Work

In this section, we analyze multiple systems that offer transactional consistency models in either monolithic or microservice architectures. For each of the analyzed systems, we provide a brief overview of its characteristics, including both the isolation levels and the proposed solution. We discuss the key features and benefits of each system, as well as possible caveats of these approaches.

4.1 Cloud Systems with Support for Transactions

Cure Cure [19] was the first system to implement Transactional Causal Consistency (TCC). Cure considers the existence of multiple datacenters, located in different geographic regions, each maintaining a full replica of a key-value store. In each datacenter, the data is partitioned across servers. Cure also assumes that clients are *sticky*, i.e., that clients access a single datacenter. In this setting, Cure supports *interactive transactions*, i.e., transactions where the read-set and write set are not known beforehand.

TCC allows concurrent transactions to commit, even if they access the same data items. Cure assumes that objects are implemented as Conflict-free Replicated Datatypes (CRDTs) [22], such that concurrent updates can be merged consistently. Updates are propagated asynchronously among datacenters. Furthermore, updates to different data partitions are propagated independently of each other. Thus, two updates performed in the context of the same transaction can arrive at different points in time to remote datacenters. Updates are tagged with metadata that allows to identify which version belongs to a consistent snapshot when serving reads.

Cure assumes that nodes are equipped with a physical clock that generates increasing timestamps and that are loosely synchronized with other clocks. When a transaction commits, it is assigned a commit time, which is computed as the highest clock value of all nodes (i.e., partitions) that have been updated by the transaction. A consistent read snapshot is captured by a vector clock V, with an entry for each datacenter. The value of each entry $V[i]$ indicates that the snapshot includes all transactions performed at the datacenter i with commit time less or equal to $V[i]$. When a transaction starts at the datacenter j, a partition is selected as transaction coordinator and a suitable read snapshot is selected to ensure that TCC is preserved. $V[j]$ is set to the coordinator's local clock, such that the newly started transaction observes previous transactions committed at the datacenter j. The values $V[i]$ for $i \neq j$ are selected such that all updates from datacenter i, with timestamp smaller or equal than $V[i]$, have already been received locally. These values are selected with the help of a global mechanism, denoted by *Globally Stable Snapshot* (GSS), that keeps track of which transactions have been received by each datacenter. Each update

performed in the context of a given transaction, executed at the datacenter j, is tagged with a snapshot vector clock (SVC), corresponding to the transaction GSS, with the local entry for the datacenter ($SVC[j]$) updated with the commit time of the transaction.

The vector clocks of the updates are used to enforce that updates are applied to each datacenter in an order that respects causality.

FlightTracker FlightTracker [23] offers a solution for managing RYW consistency for clients accessing Facebook's social graph. It offers a system that preserves the read efficiency, the tolerance for hot spots (objects stored that are very often read and written to), as well as the high availability of Eventual Consistency. FlightTracker is built as a family of APIs and a metadata service. To support RYW, FlightTracker collects the metadata associated with a user's recent writes and exposes it as a data type denominated *Ticket*. Web requests will always fetch the user's Ticket before executing any queries on Facebook's data stores.

The decision to support a weaker consistency model over Linearizability or Causal Consistency is justified by the tradeoff analysis made in [23]. It is proposed that even though stronger consistency models prevent more anomalies, and let developers create mental models more easily, the implementation of the service becomes more constrained. On a read-intensive system such as FlightTracker, most results for the queries come from the local cache replicas, even if these replicas are a few seconds behind the desired read timestamp.

FlightTracker decomposes the problem of RYW consistency in 3 parts: the Ticket data type abstraction; an infrastructure service for providing the Ticket once per request; and Ticket-inclusive reads, a mechanism used to include the user's recent writes in query results.

The Ticket data type serves as a medium to store and expose the user's recent writes as metadata. These tickets condense the system-specific details of a user's write sets across the components of the system. Such details include the Transaction ID that is associated with that specific write operation and the resulting node or edge in the social graph. It is important to note that these Tickets do not store the updated data itself.

The tickets are exchanged by the different, independently deployed, components of the system, and thus are designed to be agnostic to the specific component handling them, achieving this through encapsulation, extensible design and forward- and backward-compatibility. After the client writes an object, the Ticket is asynchronously replicated in the FlightTracker system. Considering the infrastructure service that provides the Ticket, it offers an API to the client application, used to request a ticket.

As for Ticket-inclusive reads, data stores must ensure that read operations include the updates reflected in the Ticket. When a Ticket-inclusive read reaches a cache, and the cache determines that it does not have the fresher version of the data required by the Ticket, it is considered to have a *consistency miss*. To solve this, the request can be forwarded to another region, where another cache or database might have the necessary version, although this is highly avoided (only

fewer than 3% of requests that go across regions are due to consistency misses), considering the latency impacts associated with doing this. It is important to note that, when a fresher version of the data is collected, only the exact entry for that object is fixed in the local cache. This fine-granularity allows the system to only perform extra work on specific data objects, instead of the entire cache. There are two other proposals in [23] to handle local staleness, namely delaying the request and retrying it later, which is a strategy that is not sufficient on its own but can help avoid frequently executing more costly strategies; or selecting another nearby replica that might have the necessary version of the object. The latter strategy can lead to correlated failures, such as thundering herd, and for this reason it is only considered a viable solution when used for small workloads.

4.2 FaaS Systems with Support for Transactions

FaaSTCC FaaSTCC [24] offers the TCC consistency model to *Function-as-a-Service*(FaaS) applications in an environment for multiple independent worker processes. One of the challenges the system proposes to solve is the need to overcome the issues inherited from coordinating multiple workers in the FaaS environment, namely to provide a stronger consistency model such as TCC. Initially, Cure[19] proposed a system capable of implementing TCC for sticky client sessions. However, FaaS involves having multiple independent workers and adapting TCC to this scenario imposes the need for a costly coordination mechanism between the workers. This can lead to detrimental overheads.

The solution presented in FaaSTCC rests on two large pillar contributions: adding a cache layer for each worker and the proposal of a protocol that efficiently utilizes this new layer. The cache layer is implemented as an in-memory key-value store that stores the most recent version of an object. In the proposed protocol, the storage system offers a *promise* to the cache layer, which delimits the lifespan in which a specific key is to be considered consistent. This decision is crucial to ensure an adequate use of the cache layer that motivates its use. Unlike HydroCache [25], another system that provides TCC in the FaaS enviroment using per-key dependencies between workers, FaaSTCC proposes the use of *snapshot intervals*. These intervals consist of two timestamps that capture the versions that can be read by the application. By using these snapshot intervals, FaaSTCC prevents large amounts of data from traversing the network, which can lead to impairments to the performance.

Every time a function needs to read or write an object, the client library is invoked. The client library is responsible for maintaining a copy of all objects read and written by the function. This component is relevant to guarantee that even though concurrent transactions might change the values of objects previously read, the client library still ensures that only causally consistent values are considered, thus preventing read-only transactions from unnecessarily aborting.

Compositions of executed functions are arranged in a *Directed Acyclic Graph* (DAG). When the execution starts, a DAG context is created. This includes the snapshot interval and the write-set of the transactions to be committed to storage. When a function finishes its execution, it passes this context to its

child functions. It is possible that a child function has more than one parent. In case this happens, the protocol merges the intervals of both parents. It is possible that the parents read from a mutually incompatible interval, and if that is the case, the transaction aborts. The updates written along the DAG are committed to persistent storage only at the end of the DAG, ensuring the atomicity of transactions.

The caching layer keeps the latest version of the object that were read in local transactions, and its purpose is ultimately to reduce the number of unnecessary accesses to remote storage. The updates on data items located in the cache are a result of using a publish-subscribe system between the cache and the persistent storage.

4.3 Microservice Systems with Support for Transactions

Enhancing Saga Enhancing Saga [26] tries to solve one of the issues associated with the use of the Saga pattern – lack of Isolation. As it was mentioned previously, the sagas pattern only offers *ACD* properties (*Atomicity*, *Consistency* and *Durability*) (missing the "I" of *Isolation*) to distributed transaction executions. While not providing Isolation, the Saga pattern is susceptible to both Dirty Reads and Fuzzy Reads anomalies. Enhancing Saga proposes the use of in-memory data caching to solve the lack of read-isolation presented in the Saga pattern. The system proposes allocating a *quota* of the database storage space to a memory cache server denominated as the *Quota Cache*. This memory cache server is responsible for storing the results of *CRUD*(create, read, update and delete) operations. Instead of committing the changed objects directly to the database, as it occurs in the original Sagas pattern, this memory cache server stores the values until the microservice receives an order to commit to the data store. This prevents other concurrent transactions from reading uncommitted values. It is worth noting that microservices that apply the quota cache also profit from low latency and high throughput, as the results of read operations to the remote storage are stored in the cache for future reads. The proposed solution in Enhancing Saga only associates the cache server to microservices that require the execution of compensating actions in case of transaction abort. This is motivated by the fact that some microservices either only execute read operations, or because they execute idempotent operations.

The system uses an orchestrator module to configure every microservice to their corresponding web-clients. To coordinate the sagas, two approaches are followed: orchestration and event-choreography. Following the orchestration approach, a manager controller manages all the communications between the microservices. On the other hand, the event-choreography approach is used as microservices finish their local transactions. Following this approach, after a local transaction completes, the microservice responsible for it produces an event that is consumed by the following microservices in the functionality.

When all microservices in the functionality finish their local transactions, the coordinator is ready to begin the commit process. To guarantee that the changes

are seen atomically, an event is sent to the microservices that updated objects in that functionality, notifying them that it is safe to commit. These microservices are then be responsible for running an eventual commit sync service to perform the commit. This commit sync service is characterized by synchronously holding the commit decision until it is known whether all local commits succeeded or any local transaction failed, triggering compensating actions to unwind the changes made in the caches.

5 Analysis

In this section, we will discuss the advantages and disadvantages of the systems surveyed in the previous section. Table 8 provides a comparative analysis of the features of non-microservice systems supporting transactions.

Systems	Target	Consistency	Metadata Size	Read RTT	Commit RTT
Cure	Cloud	TCC	*O(N)*	1*	3*
FlightTracker	Cloud	RYW	*O(W)*	≥ 2	1
FaaSTCC	FaaS	TCC	*O(1)*	1*	2

Table 8: Cloud and FaaS Systems Comparison. W represents write-set, N represents the number of partitions, * means that time period might include blocking/waiting or aborts.

5.1 Target Environment

In Table 8, we categorize the analyzed systems based on their target environment. It is crucial to understand and consider the specific constraints and challenges that each environment imposes on the solutions proposed. Cloud systems such as Cure and FlightTracker are designed for environments with a static number of data centers and partitions. On the contrary, FaaS systems such as FaaSTCC are designed to facilitate the auto-scaling, and therefore their components are bounded to be changed frequently.

5.2 Limitations of Non-Microservice Transaction Systems

The previously discussed systems all contain their own limitations. To the best of our knowledge, there are currently no systems that actively support transactions on microservice architectures while also offering the TCC consistency model. Considering the analyzed systems, both Cure and FaaSTCC offer TCC. While Cure provides TCC for sticky client sessions, FaaSTCC provides

TCC for non-sticky client sessions. Considering our scenario, functionalities communicate with several microservices in a single transaction. This way, we also face non-sticky client sessions.

The Cure system inherently supports RYW consistency due to its use of sticky client sessions. This way, the RYW consistency is not a significant concern in the Cure system, as the data centers that a client communicates with are always the same. FaaSTCC on the opposite, requires coordination across multiple clients, a scenario that is often similar to the challenges we face, as we propose to offer TCC for multiple microservices, part of functionalities.

In addition, when a client executes a read operation, Cure's algorithm requires a partition to wait before returning the request until its own clock catches up with the snapshot vector clock for the transaction. This guarantees that a read operation always obtains the latest version of the request object with no newer commit timestamp than the one specified in the snapshot.

The FlightTracker system is suitable for Cloud based systems that support multiple, often heterogeneous, client systems. Is also only supports region-sticky user routing. Also, it only offers support for RYW consistency on top of Eventual Consistency, a decision that is based on the nature of Facebook's social graph, and their need to serve most client queries at a local replica, even if these results are a few seconds stale, or don't even present all the dependencies required in a stronger consistency model such as TCC.

The aim of our work targets the lack of systems that offer stronger consistency models than Eventual Consistency, while still ensuring high availability and the atomicity and isolation of transactions without incurring major losses to the system's performance. We believe that combining Cure's use of vector clocks to guarantee TCC, as well as FaaSTCC's use of snapshot intervals to expand the limitations of Cure for non-sticky sessions, can help us define the necessary mechanisms to offer TCC in the microservice architecture.

5.3 Transaction Models in Microservice Architectures

We analyzed two systems capable of offering transaction semantics in the microservice architecture. Sagas is considered the standard pattern for implementing transaction systems in the microservice architecture due to its capabilities for distributing the often large functionalities of an application across multiple smaller transactions. It offers atomicity, guaranteeing that either all transactions are executed, or none does, and compensation mechanisms that undo the necessary changes. The Sagas pattern suffers from the lack of Isolation property, something that is solved with Enhancing Sagas, a system that uses in-memory caches to prevent concurrent transactions from reading partial committed data from the data store. Both systems stand on Eventual Consistency, a weak consistency model. Enhancing Sagas, however, does not propose any specific mechanisms to account for the fact that some microservices might require data that is not managed locally.

6 Architecture

In this section, we describe the architecture of the system we propose to implement. Our approach involves developing a middleware layer, capable of ensuring that microservices always observe mutually consistent versions of remote objects. To achieve this, we plan to provide Transactional Causal Consistency to microservices systems that employ orchestration mechanisms. The middleware layer will primarily consist of a set of database wrappers that intercept read and write requests to the database.

6.1 System Components

Orchestration We consider providing TCC to microservices systems that employ orchestration mechanisms, because we believe that orchestration can help us maintain the transactional context necessary to offer TCC. The coordinator will be able to alert relevant microservices when a commit to the database is required, ensuring the atomicity of the supported functionalities.

Database Wrappers In order to support TCC across different storage systems, we propose implementing a set of database wrappers that are capable of intercepting read and write operations. Ideally, we would like to follow an approach similar to FaaSTCC [24], with the use of snapshot intervals to ensure read coherency across the functionality. To do so, we plan to support a TCC protocol at the database level.

To guarantee isolation, and following an approach similar to the one proposed in Enhancing Sagas [26], we plan to use data caches where write operations are stored before having authorization from the orchestration coordinator to commit to the different main databases. We plan to develop three distinct wrapper implementations, for three types of data store systems: a NoSQL database (MongoDB), and two in-memory data caches such as Redis and Memcached.

Considering the supported database and in-memory caches, and while it is not obligatory that these storage systems provide multi-versioning capabilities, our system is expected to perform more efficiently in these scenarios. This is due to the fact that, transactions that require long periods to execute are prone to struggle to read from the same consistency snapshot they initially read from, as more recent values are inserted in the database by concurrent transactions. By working with systems that support multi-versioning, we are able to prevent unnecessary transaction aborts.

6.2 Data Wrappers Cache Mechanism

Our data store wrappers will contain two different types of data caches: one responsible for holding updated objects inside the microservice, that wait for the commit order from the coordinator to commit these stored value to the data store; and one for storing remote microservices data that is updated in the microservices that committed their transactions. Note that this last data cache

values are read asynchronously, and their sole purpose is to improve latency by reducing the number of remote queries.

6.3 Data Replication

Considering that there are situations in which microservices need to read data that is not managed locally, we try to minimize the number of remote data accesses that a microservice is required to do to execute its logic. We also try to minimize the necessary metadata that needs to be passed across microservices to ensure that the causal dependencies are respected. To do so, we plan to introduce two mechanisms: *asynchronous data replication* across the database wrappers, and a *bitmap tracker* of updated objects, allowing microservices to know when they are required to query remote data in other microservices.

Asynchronous Data Replication: as described before, to improve latency, microservices asynchronously replicate committed data. Using some form of publish-subscribe mechanism, microservices that require this remote data subscribe to the events published and cache the values for future use. This allows services to reduce the number of future remote data accesses and consequent delays associated with these operations.

However, as it was discussed in Section 3.4, asynchronous data replication by itself does not guarantee that the cached values are coherent according to the snapshot being used in the functionality. To guarantee that microservices are always aware that data was updated inside the current functionality, and thus need to obtain the new values to guarantee the RYW property, we propose using some form of a bitmap tracker, capable of identifying updated objects.

Bitmap Tracker: this mechanism allows each microservice inside the functionality to know if an object that it plans to access was updated in a previous microservice. This is an important step to prevent microservices from reading inconsistent data, preventing unnecessary transaction aborts. When the tracker identifies an object as having been updated, the microservice can remotely access its value by asking the coordinator to contact the microservice that was responsible for updating the object, retrieving the more recent value.

6.4 Write Operations

To ensure the atomicity property for the functionalities, updates performed by a client do not immediately become visible in the data storage. Instead, each wrapper locally stores the updates in memory. These are only committed as the transaction ends. The wrappers must retain these updates until receiving a commit instruction from the orchestration coordinator.

6.5 Read Operations

The database wrappers we plan to develop must be able to fetch data according to the causal snapshot the client is reading from, as well as any updates made

to that same data in the current transaction (to guarantee the RYW property). To guarantee this, we plan to include a vector clock V in each database wrapper of size N, equal to the number of microservices in the executing functionality, where for each $i \in [0, N], V[i]$ corresponds to the latest known updates visible at the microservice i with commit time less or equal to $V[i]$. This allows us to perform consistent read operations.

Following a similar strategy as proposed in FaaSTCC [24], when a transaction is started at the beginning of the functionality, the microservice defines a snapshot interval that will be used across the entire functionality. It is initialized using the vector clock defined previously, received by the orchestration coordinator. This vector clock allows the microservice to define the upper bound of the snapshot interval, as it represents the latest known updates visible at each microservice. As read operations are executed, this interval can be shortened. The bounds of this snapshot interval represent the limited timestamps from which the functionality can read objects, to ensure consistent reads.

6.6 Commit

In order to atomically execute the transaction, the orchestration coordinator must relay the commit order to each of the database wrappers. To decide on which mechanism to use to implement the commit protocol, we considered two options: following an approach similar to the Two-Phase Commit protocol [27], with the exception that we do not require blocking the transaction, as it often the case following this approach. This way, upon the completion of the final microservice, the orchestration coordinator sends a prepare message to the wrappers. In response, each wrapper proposes its vector clock as the commit timestamp. After receiving all proposals, the orchestration coordinator chooses the maximum of each entry of the received vector clocks and directs the wrappers to commit to the database using this selected timestamp. Upon receiving the request, the wrappers commit the updates executed in the context of the functionality, which were previously stored in cache. For systems that include thousands of microservices, this approach might pose some drawbacks related to the latency, as we require waiting for all microservices to propose their vector clocks. However, we believe that most functionalities do not span across the entirety of the system's microservices, and thus, these drawbacks have very limited bounds.

We also considered as another viable option, letting the orchestration coordinator decide the timestamp for the commit based on the value of a physical clock. This would imply that all microservices had access to a synchronized physical clock, and that the coordinator could choose the value of the clock of the last microservice that executed its logic. It is important to note that, due to clock skews, the coordinator would have to choose a reasonable value that encompassed the possible delay.

At this point, we propose using the first option, as we believe it to be the most adequate to the problem being solved.

7 Evaluation and Implementation

We plan to implement our system and evaluate its performance using a benchmarking suite, such as DeathStarBench benchmark [28]. We will compare the results obtained using various consistency models in order to thoroughly assess the capabilities of our system. The evaluation will focus on different aspects of the system performance, such as latency, throughput, and the reduction of compensating actions executed in the presence of anomalies.

7.1 Benchmarks

DeathStarBench Benchmark This open-source benchmark suite is built with microservices that include large end-to-end services where we can test the capabilities of our proposed system. This benchmark includes multiple scenarios such as a social network, a media service, etc. We will focus on the social network scenario, as it is a commonly used scenario to motivate the usage of TCC. To evaluate the proposed system, we plan to extend this benchmark, to include an option to run the desired social network scenario on top of either TCC or EC. By comparing the results obtained while executing our system with these two consistency models, we can understand the impacts and benefits of supporting transactions in the microservice architecture using the TCC consistency model.

Social Network: The Social Network scenario, offered by DeathStarBench benchmark, simulates a typical social network scenario, where users can read and write posts from other users. This scenario is implemented using loosely-coupled microservices, and includes unidirectional follow relationship between users. The microservices in this scenario communicate with each other via Apache Thrift RPCs [29]. The system supports a range of actions, such as: Create text post; Read post; Search database for user or post; Follow/ Unfollow user, etc. The system includes separate data stores for different service, including a data store for users, one for posts, and one for the social graph, etc. These databases store data required by the various microservices.

7.2 Metrics

Latency To evaluate the performance of our system, we will measure the delay between the time a request is made and the time the response is returned to the client. We will conduct experiments in which we vary the percentage of read and write operations, as well as the number of remote objects read (i.e., reads of objects across microservices). This process is critical for understanding potential bottlenecks of our system, possibly related to dependency enforcement and the commit protocol developed.

Throughput We will measure the maximum number of transactions per second that the system is able to execute, to understand the amount of load the system can sustain. We consider that measuring throughput can help us to understand, even if indirectly, the overhead impact associated with introducing our mechanisms, to support TCC.

Reduction of compensating actions We will also measure the impact that our mechanisms have in reducing the amount of necessary compensating actions the developers needs to implement in the event of aborting the transaction. With this measure, we plan to study the relevance of the anomalies prevented by introducing TCC in systems that implement the microservice architecture.

7.3 Expected results

Considering the two first metrics: Latency and Throughput, ideally we would like to offer a system that guaranteed high throughput and low latency. However, we know that, by offering a stronger consistency model such as TCC, we are bounded to expect worse performance results than systems that provide weaker consistency models, such as system that follow the Sagas approach. By comparing the results obtained running the base model and the proposed model, offering TCC, we expect to minimize the number of aborting transactions, by reducing the number of anomalies captured by the system.

8 Scheduling of Future Work

Future work is scheduled as follows:

- January 13 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15 Deliver the MSc dissertation.

9 Conclusions

The microservice architecture makes evolution in applications an easier task, allowing services to be developed independently of each other, allowing developers to choose the best-fit programming language and storage system to the specific service, without being restrained by other services. However, this creates a challenge to the atomicity and consistency of transactions.

In this report, we surveyed the state-of-the-art FaaS, Cloud and Microservice systems. We analyzed their consistency guarantees, discussing the benefits and the caveats of each consistency level and implementation. We proposed a solution to support stronger consistency levels in microservice systems. Finally, we defined the evaluation methods and presented the scheduling for future work.

Acknowledgments We are grateful to Rafael Soares and Valentim Romão for the fruitful discussions and comments during the preparation of this report. This work was partially supported by project DACOMICO (via OE withref. PTDC/CCI-COM/2156/2021).

References

1. Mauro, Tony: Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices (Feb 2015) Accessed: 28/12/2022.
2. Viggiato, M., Terra, R., Rocha, H., Valente, M.T., Figueiredo, E.: Microservices in practice: A survey study. arXiv preprint arXiv:1808.04836 (2018)
3. Almeida, J.F., Silva, A.R.: Monolith migration complexity tuning through the application of microservices patterns. In: European Conference on Software Architecture, Springer (2020) 39–54
4. Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., Stafford, R.: Patterns of Enterprise Application Architecture. Addison Wesley (2002)
5. Fowler, M.: Microservices, a definition of this new architectural term. https://martinfowler.com/articles/microservices.html Accessed: 14/12/2022.
6. Rudrabhatla, C.K.: Comparison of event choreography and orchestration techniques in microservice architecture. International Journal of Advanced Computer Science and Applications **9**(8) (2018)
7. Meshenberg, R., Evans, J.: Netflix at aws re:invent 2015. https://netflixtechblog.com/netflix-at-aws-re-invent-2015-2bc50551dead (Oct 2015) Accessed: 22/12/2022.
8. Fielding, R.T., Taylor, R.N.: Architectural Styles and the Design of Network-Based Software Architectures. PhD thesis, University of California, Irvine, Irvine, California (2000) AAI9980887.
9. Saleem, I., Nargund, P., Buonora, P.: Data caching across microservices in a serverless architecture. https://aws.amazon.com/blogs/architecture/data-caching-across-microservices-in-a-serverless-architecture/ (Jul 2021) Accessed: 19/12/2022.
10. (n.d.): Consistency models. https://jepsen.io/consistency Accessed: 14/12/2022.
11. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Comput. Surv. **49**(1) (jun 2016)
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (jul 1978) 558–565
13. Ahamad, M., Neiger, G., Kohli, P., Burns, J., Hutto, P., Anderson, T.: Causal memory: Definitions, implementation and programming. IEEE Transactions on Parallel and Distributed Systems **1** (1990) 6–16
14. Mahajan, P., Alvisi, L., Dahlin, M., et al.: Consistency, availability, and convergence. University of Texas at Austin Tech Report **11** (2011) 158
15. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. (2011) 401–416
16. Herlihy, M. In: Linearizability. Springer US, Boston, MA (2008) 450–453
17. Little, T.: Ensuring data consistency in microservice based applications. https://blogs.oracle.com/database/post/ensuring-data-consistency-in-microservice-based-applications (Oct 2022) Accessed: 18/12/2022.
18. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. **7**(3) (nov 2013) 181–192
19. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). (2016) 405–414

20. Štefanko, M., Chaloupka, O., Rossi, B., van Sinderen, M., Maciaszek, L.: The saga pattern in a reactive microservices environment. In: Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019), SciTePress Prague, Czech Republic (2019) 483–490
21. Garcia-Molina, H., Salem, K.: Sagas. ACM Sigmod Record **16**(3) (1987) 249–259
22. Letia, M., Preguiça, N., Shapiro, M.: Crdts: Consistency without concurrency control. arXiv preprint arXiv:0907.0929 (2009)
23. Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J., Bronson, N.: {FlightTracker}: Consistency across {Read-Optimized} online stores at facebook. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). (2020) 407–423
24. Lykhenko, T., Soares, R., Rodrigues, L.: Faastcc: Efficient transactional causal consistency for serverless computing. In: Proceedings of the 22nd International Middleware Conference. Middleware '21, New York, NY, USA, Association for Computing Machinery (2021) 159–171
25. Wu, C., Sreekanti, V., Hellerstein, J.M.: Transactional causal consistency for serverless computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. (2020) 83–97
26. Daraghmi, E., Zhang, C.P., Yuan, S.M.: Enhancing saga pattern for distributed transactions within a microservices architecture. Applied Sciences **12**(12) (2022)
27. Lampson, B., Sturgis, H.E.: Crash recovery in a distributed data storage system. (1979)
28. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., Delimitrou, C.: An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '19, New York, NY, USA, Association for Computing Machinery (2019) 3–18
29. Apache Software Foundation: Thrift. https://thrift.apache.org/