TÉCNICO LISBOA

Transactional Causal Consistency For Microservices Architectures

João Miguel Rocha Queirós

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. José Orlando Roque Nascimento Pereira

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to express my deepest gratitude to my thesis advisor, Prof. Luís Rodrigues for his unwavering support, guidance, and expertise, all of which have made this Thesis possible.

A special mention goes to Rafael Soares, whose guidance and willingness to help with all the questions I had throughout the past year were fundamental to the completion of this Thesis. Your dedication and knowledge were invaluable, and I am deeply grateful for your assistance.

I am also profoundly thankful for the endless caring, love and encouragement from my parents and my brother Artur. Their unwavering support and belief in my capacities have been part of my driving force throughout my academic journey. I am grateful for their sacrifices and understanding throughout these past years.

I extend my appreciation to my cousin, Filipe Paredes, whose technical skillfulness and patience helped me during the development of this work.

Lastly, I am indebted to all of those who shared their time to discuss the ideas that lead to the creation of this Thesis, without whom none of this work would have been possible.

To each and every one of you – Thank you.

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. The execution of a microservice may need to read data items that are managed by other microservices. These read operations can be completed either by doing remote calls or by reading from a local replica (possibly inconsistent) of the data managed by the other microservices, that is updated using some form of publish-subscribe middleware. In any case, there is a possibility of the microservice reading mutually inconsistent versions of data objects, generating consistency anomalies that would never occur in a monolithic system. To correct the effects of these anomalies, programmers often have to develop compensating actions responsible for restoring the consistency of the system. In this work, we propose and evaluate a mediating layer, that we designated μTCC, that offers the Transactional Causal Consistency guarantees for microservices' architectures. Using a version control mechanism, μTCC guarantees that different microservices, when executing a given functionality, observe mutually consistent versions of data, thus reducing the number of anomalies. Our experimental evaluation shows that the proposed solution prevents the occurrence of Transactional Causal Consistency anomalies while reducing the overall latency of functionalities by $2.63\times$, thanks to being able to reduce read transaction abortion frequency and respective re-executions.

Keywords

Microservices, Transactional Causal Consistency, Consistency, Isolation

Resumo

A arquitetura de microsserviços é uma abordagem de engenharia de software que estrutura uma aplicação como um conjunto de serviços fracamente acoplados. Cada microsserviço é responsável pela gestão de um subconjunto pequeno e coeso das entidades de domínio, podendo ser implementado, executado e mantido independentemente dos outros microsserviços. A execução de um microsserviço poderá requerer a leitura de dados geridos por outros microsserviços. Estas operações de leitura podem ser concluídas através da execução de invocações remotas ou lendo a partir de uma réplica local (possivelmente incoerente) dos dados geridos pelos outros microsserviços, a qual é atualizada usando algum tipo de camada intermédia de publicação/assinatura. Em qualquer caso, existe a possibilidade do microsserviço ler versões dos dados mutuamente incoerentes, dando origem a anomalias de coerência que nunca ocorreriam num sistema monolítico. Para corrigir os efeitos destas anomalias, os programadores frequentemente desenvolvem ações de compensação responsáveis por restaurar a coerência do sistema. Neste trabalho, propomos e avaliamos uma camada de mediação, que designámos por μTCC, que oferece as garantias de Coerência Causal Transacional a arquiteturas de microsserviços. Utilizando um mecanismo de controlo de versões, o μTCC garante que diferentes microsserviços, ao executarem uma dada funcionalidade, observam versões dos dados mutuamente coerentes, reduzindo o número de anomalias. A nossa avaliação experimental mostra que a solução proposta evita a ocorrência de anomalias Transactional Causal Consistency (TCC) garantindo a redução da latência global das funcionalidades em $2.63\times$, graças à diminuição de transações de leitura abortadas e respetivas reexecuções.

Palavras Chave

Microsserviços, Coerência Causal Transacional, Coerência, Isolamento

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

1SR	Serializability
CC	Causal Consistency
CC+	Causal+ Consistency
CRDTs	Conflict-free Replicated Datatypes
CRUD	Create, Read, Update and Delete
DAG	Directed Acyclic Graph
FaaS	Function-as-a-Service
gRPC	gRPC Remote Procedure Call
MR	Monotonic Reads
MW	Monotonic Writes
REST	Representational State Transfer
RYW	Read Your Writes
RPC	Remote Procedure Call
RR	Repeatable Reads
RTT	Round Trip Time
SI	Snapshot Isolation
SQL	Structured Query Language
Strict 1SR	Strict Serializability
TCC	Transactional Causal Consistency
WFR	Writes Follow Reads
XCY	Cross-Service Causal Consistency

1

Introduction

Contents

This work addresses the problem of offering Transactional Causal Consistency (TCC) to microservice applications. It describes the design and implementation of a middleware layer that ensures the atomicity of the results of functionalities that span multiple microservices and that ensures that functionalities always read versions of data objects that are mutually consistent. The thesis also reports the results of an experimental evaluation of our middleware layer.

1.1 Motivation

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 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 select 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 services [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.

The use of the Sagas [4] pattern is one of the approaches to deal with read anomalies, such as non-repeatable reads and fractured reads, that can occur in microservice systems. This pattern models a business transaction as a sequence of local transactions. Each one of these transactions runs in the microservices' own local database. If a microservice in the Saga needs to abort, due to a violation of a business invariant, compensating actions are carried out capable of reestablishing the consistency of the applications, by reversing the effects of update operations committed by previous local transactions. The use of compensating actions can be both complex and time-consuming for the developers to implement; it 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 thesis, we address the problem of providing transactional consistency guarantees in microservices systems, in order to minimize the consistency anomalies related with the lack of Isolation and Atomicity. We take particular care to offer the highest consistency guarantees that can be achieved without jeopardizing the inherent advantages of the use of microservices, namely the high availability and low latency promises. In particular, we study the necessary mechanisms to offer TCC to the microservices architectures.

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 the identification of 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 versions automatically, exchange metadata transparently among microservices, and keep multiple object versions without requiring programmers to manage the version mechanisms explicitly.

1.2 Contributions

The main contribution of this thesis is:

- The design of a set of coordination mechanisms to support TCC across multiple microservices. These mechanisms ensure that all invocations to microservices, when performed in the context of a given functionality, read mutually consistent versions of data objects.

The proposed mechanisms avoid the need for the development of compensating actions to over-

come consistency anomalies such as non-repeatable reads and fractured reads that may occur when executing functionalities that span multiple microservices without coordination mechanisms.

1.3 Results

This thesis has produced the following results:

- A novel middleware layer, named μTCC, which implements the mechanisms required to provide TCC support in existing applications that follow the microservices architectural pattern.
- An experimental evaluation of μTCC, including the study of the overhead introduced by the use of the proposed coordination mechanisms and the corresponding impact on the execution of functionalities.

1.4 Research History

This work was developed in the context of the DACOMICO research project, which has, as an objective, the advancement of the architecture and deployment practices of contemporary cloud-based applications. By concentrating on the development of middleware extensions to support TCC, where the work presented in this thesis focuses on, and automated mechanisms for anomaly detection, this research project aims to provide developers with practical tools and methodologies for breaking down large monolithic applications into smaller, loosely-coupled microservices. The primary aim of this project is to address anomalies commonly associated with eventual consistency while preserving a robust level of microservices decoupling, thereby enhancing the reliability and resilience of cloud-based application facing escalating complexity and scalability challenges.

Being part of this research project, I benefited from the useful feedback from the research team of DACOMICO, both from INESC-ID and from LASIGE.

Parts of the work described in this thesis have been published as:

- J. Queirós, R. Soares and L. Rodrigues. Suporte para Coerência Causal Transacional em Sistemas de Microsserviços In *Actas do décimo quarto Simpósio de Informática (Inforum)*, Porto Portugal, September 2023.

This work was supported by national funds through FCT - Fundação para a Ciência e a Tecnologia as part of the projects with references UIDB/50021/2020 and DACOMICO (financed by the OE with ref. PTDC/CCI-COM/2156/2021).

1.5 Structure of the Document

The rest of the thesis is organized as follows: Chapter 2 introduces the key concepts relevant to our work and provides an overview of the related work; Chapter 3 describes the architecture of μTCC and its implementation; Chapter 4 reports the results obtained from our experimental evaluation, where we study the impact and overhead associated with the use of μTCC; Finally, Chapter 5 concludes this document, highlighting the main findings of this thesis and providing some directions for future work.

2

Background and Related Work

Contents

In this chapter, we introduce the core concepts that are relevant to our work. Specifically, we delve into the fundamental characteristics of microservice architectures and explore how these attributes can interfere with the challenge of providing consistent results to clients. Section 2.1 commences with the presentation of a simple application as an illustrative example, enriching our exposition. Section 2.2 proceeds with an examination of the characteristics of both Monolithic and Microservices Architectures. Subsequently, in Section 2.3 we present a description of the Data Distribution view within microservices, followed by a description of various consistency models for Data Replication in Section 2.4. We continue by introducing several Isolation levels in Section 2.5, along with an examination of the diverse types of Consistency Anomalies stemming from the use of each specific consistency policy in Section 2.6. Section 2.7 addresses the consistency problems in microservices. In Section 2.8 we present some previous relevant works that support distributed transactional execution across both Cloud and Function-as-a-Service (FaaS) environments. Finally, in Section 2.9 we discuss the advantages and challenges of the surveyed solutions.

2.1 Example: Online Shop System

For illustrative purposes, we will employ a straightforward application mimicking an online store. This virtual store enables clients to perform various actions such as item searching, adding products to their shopping baskets, and proceeding with their respective payments. Each item within this application is distinguished by a unique identifier and, in addition to its descriptive name, it also boasts attributes such as price and applicable discount. Both the price and the discount properties can be modified during the application's execution. Notably, the shopping basket entity is specific to each client, and maintains references to the selected items. The entities overseen by the application are represented in Figure 2.1.

Figure 2.1: Online Shop Entities

The architecture of the Online Shop system follows a possible microservices decomposition. Figures 2.2, 2.3 and 2.4 portray the individual microservices that comprise this system, each accompanied by a description of the supported methods. Additionally, the system uses a separate microservice, the Frontend service, that assumes the role of the system's entry point. It acts as the primary interface,

facilitating user interactions with the online shop system.

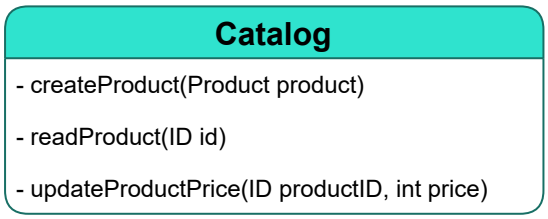

Figure 2.2: Catalog Microservice

Figure 2.3: Discount Microservice

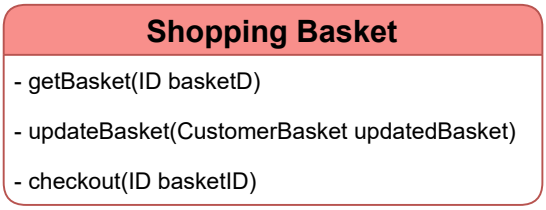

Figure 2.4: Shopping Basket Microservice

The application provides a diverse set of functionalities, including tasks such as *updatePriceAndDiscount*, *addProductToBasket* and *readBasket*, all of which are enumerated in Figures 2.5 to 2.7.

These functionalities are characterized by the following operations:

- *UpdatePriceAndDiscount functionality* (Figure 2.5): involves the Frontend, Catalog and Discount microservices. It is responsible for updating the price and discount properties of a single Product. To execute this, the functionality sequentially invokes the Catalog and Discount microservices.

- *AddProductToBasket functionality* (Figure 2.6): involves the Frontend, Catalog, Discount, and Shopping Basket microservices. It focuses on the addition of a single product to the client's shopping basket. To do so, the functionality first acquires the necessary information regarding the product's current price and discount from both Catalog and Discount services. Subsequently, it provides the Shopping Basket service with the updated basket information.

- *ReadBasket functionality* (Figure 2.7): involves the Frontend, Catalog, Discount, and Shopping Basket microservices. Its function entails retrieving product information for all items within the client's shopping basket. To accomplish this, the Shopping Basket service invokes the Catalog and Discount services to obtain the current values for the product's dynamic properties, namely the price and the discount values. The result includes the client's shopping basket containing all the product's information.

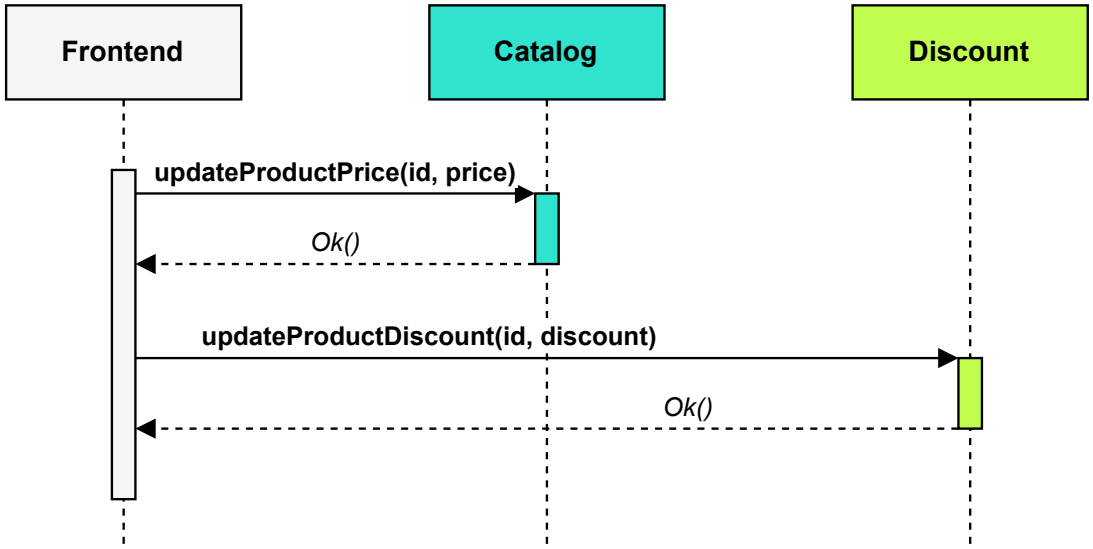

Figure 2.5: UpdatePriceAndDiscount functionality

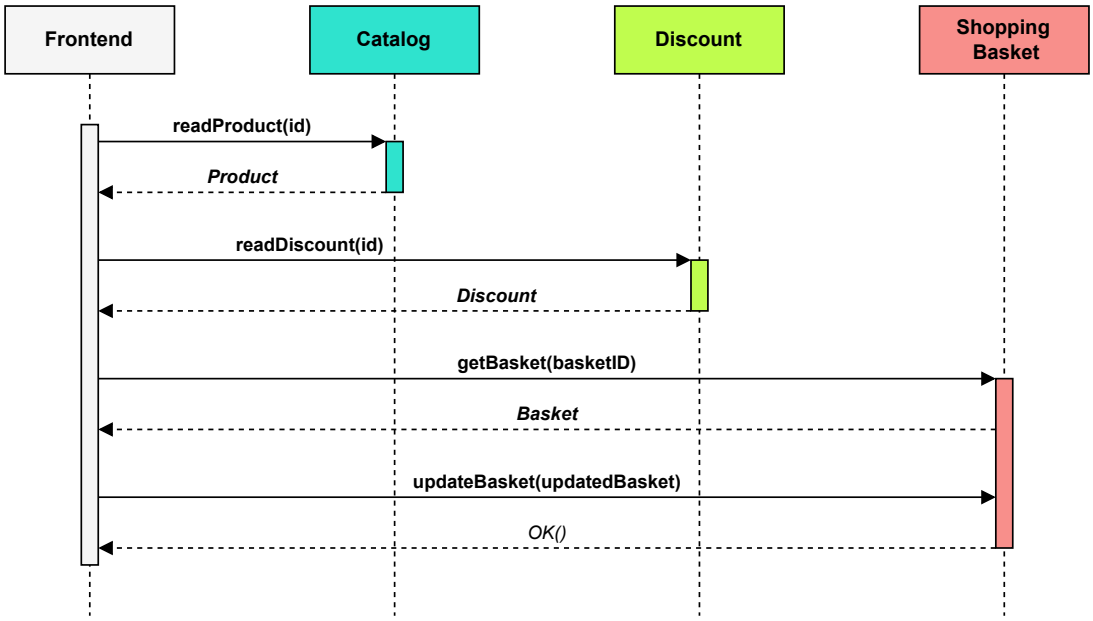

Figure 2.6: AddProductToBasket functionality

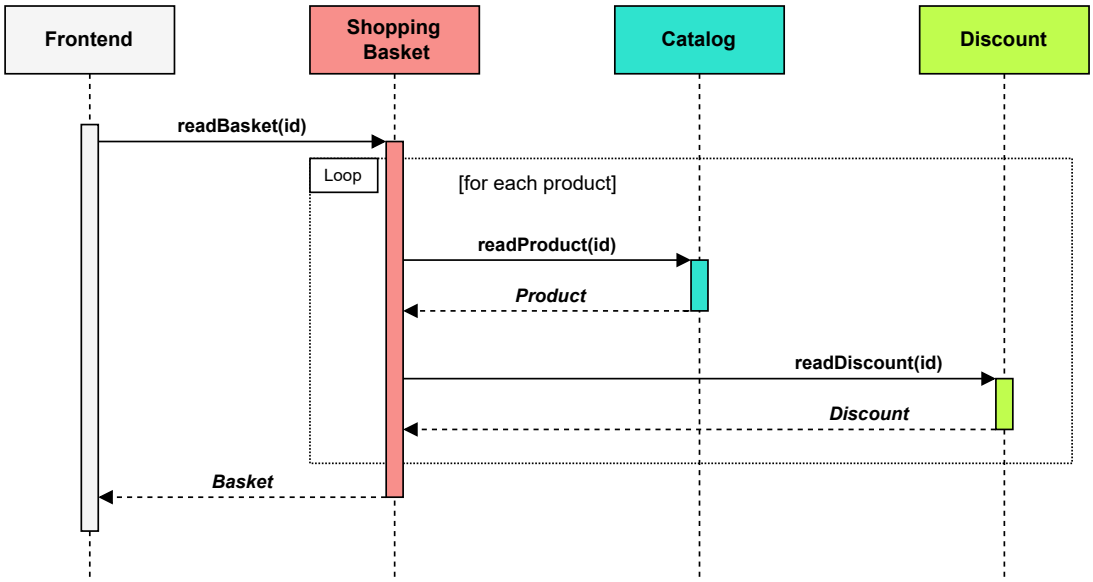

Figure 2.7: ReadBasket functionality

2.2 Monolithic and Microservices Architectures

Contemporary applications often adopt the three-tier architecture [5]. This architecture includes a presentation tier that handles the communications with the client, an application tier responsible for executing business logic, and a data tier dedicated to data storage, typically in a form of a database. In this section, our emphasis is directed towards 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* [6]. With regard to the separation of concerns, the monolith approach is suitable when the application is small, being developed and maintained by a small team. However, as the application expands, in both scope and functionality, the complexity of its codebase tends to escalate [7]. Furthermore, the task of achieving optimal resource scaling for the application becomes more challenging, as it is impossible to individually scale each subcomponent within the application [8].

The microservices architecture is a design paradigm characterized by the subdivision of an application into multiple loosely-coupled modules (or services) that can be developed, maintained, deployed and provisioned independently of each other. Microservices can typically 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, according to the specific needs. Consequently, microservices have earned growing acceptance within the software development community [9], having

established themselves as an essential part in the architecture of applications developed by prominent companies such as LinkedIn [10], Netflix [1], Uber [11] or SoundCloud [12].

There are a number of challenges raised by the microservice architecture that we aim to address in this work. Given that monolithic applications typically use a single database, it is easy to implement functionalities as atomic transactions. 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* [13]. 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 [14]. 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 existence, 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.

2.3 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 retrieve and process data values from multiple sources, either data stores managed by other microservices or legacy systems, for example. There are two main approaches to support such

accesses: real-time remote invocations and data replication.

When adopting the first approach, a microservice performs a remote invocation to another microservice to fetch the required data. Such remote invocations are commonly implemented through a Representational State Transfer (REST) interface [15] or the gRPC Remote Procedure Call (gRPC) framework, Google's high-performance, open source Remote Procedure Call (RPC) framework [16].

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 [17]. 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. Leveraging these caching mechanisms allows systems to minimize the latency associated with inter-service communications in microservices architectures. Unfortunately, due to asynchronous updates of replicas, local reads may yield inconsistent values. We discuss the problem of consistency with replicated data in the following sections.

2.4 Consistency with Data Replication

Current microservice-based systems typically enforce weak consistency policies when executing transactions across multiple microservices. In contrast to monolithic applications, where typically a single logical database if favored for data persistency, microservices often allow individual services to manage their own database. This flexibility is achieved by deploying multiple instances of the same database technology, or entirely different database systems (loosely-structured NoSQL databases versus relational databases, for example), a strategy commonly referred to as *Polyglot Persistence* [18]. However, the latter approach introduces new challenges, as the heterogeneous database choices may provide different levels of consistency guarantees. Although not exclusive, this approach is typically more common in systems that follow the microservices architecture pattern [19].

We will now proceed by analyzing different existing consistency models for data replication, following the analysis outlined in [20]. We will introduce the most relevant consistency levels commonly employed in both monolithic and microservice architectures. Our exploration will follow the hierarchy of consistency strength, starting with the less strict consistency models, designed for transactions involving a single-object, before moving on to examine several multi-object consistency models. The transactions associated with these consistency models encompass both read and write operations.

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

Eventual consistency: 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 the following:

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.

Read Your Writes (RYW): as defined in Viotti et al. [21], 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.

Causal Consistency (CC): a combination of the previous four models' properties. Steams from Lamport's definition of *happened-before* relation [22]. 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 Lamport [22]. CC captures the concept of potential causality by establishing connections between consecutive operations within a single process and operations that occurred in other processes but possibly became visible due to messaging mechanisms. CC guarantees that all processes perceive the same order of causally-related operations. In essence, CC guarantees the system's consistent view of its data maintains and ensures that operations are ordered meaningfully.

Despite providing stronger guarantees, CC does not ensure that replicas converge to the same state under concurrent operations [23]. To guarantee this property, while maintaining the precise guarantees that the state applications observe, we define a stronger consistency model:

Causal+ Consistency (CC+): as introduced in [24,25], CC+ is an extension to CC, 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 CC+ 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.

2.5 Isolation in Microservices

We will now explore various consistency models designed to uphold isolation properties for multi-object transactions. In contrast to single-object transactions, these consistency levels determine the extent of isolation guaranteed between transactions. Just as in our examination of single-object transaction models, we will delve into several consistency models, beginning with the less strict isolation levels. In microservices architectures, preserving loose coupling often leads to the implementation of weak consistency policies across concurrent functionalities. Common 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. Some levels of isolation in these types of deployments include:

Repeatable Reads (RR): The definition of RR is broad and can be ambiguous. For our work, we follow the definition of RR as defined in Bailis et al. [26], 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 [27]. 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.

Although not common, microservice architectures can enforce stronger isolation policies. This, however, comes with the cost of efficiency loss, as stronger isolation models require stricter control over concurrent functionalities. The next example on the isolation hierarchy is Snapshot Isolation (SI), which is often the isolation level offered by default inside many commercial databases.

Snapshot Isolation (SI): in this isolation model, 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.

As stated before, SI can be enforced over transactions spanning across multiple services, and in a multi-database scenario. To do so, it is necessary to execute distributed transactions. The X/Open XA eXtended architecture [28] 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. Depending on the data criticality of the functionality executed, monolithic systems may require the enforcement of stronger isolation policies. For instance, to provide Serializability (1SR), a transaction may lock data items in a given microservice until another microservice is ready to commit.

Serializability (1SR) this model is considered a strong consistency policy. It 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. 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 RR and SI.

The strongest isolation policy guarantees that functionalities that are executed concurrently in a microservice architecture are isolated from each other. This level of isolation is known as Strict Serializability (Strict 1SR), often referred to as the *"golden standard"* of distributed transaction semantics [29]. It is defined as follows:

Strict Serializability (Strict 1SR): represents the strongest consistency policy. As defined in Herlihy [30], akin to 1SR, this model guarantees that transactions execute atomically. The key distinction lies in Strict 1SR's requirement that transactions must align with the real-time ordering of the events, adhering to Linearizability's real-time constraints. In simpler terms, if a transaction T_1 completes before a transaction T_2 begins, then transaction T_2 must be able to observe the results of the transaction T_1 in the serialized order. Strict 1SR implies both *Serializability* and *Linearizability*.

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

2.6 Anomalies

The lack of isolation in microservices' environments stems from executing 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 atomic transactions in a monolithic application, also known as *anomalies*.

In the following section, we discuss common anomalies that may cause the functionality to yield incorrect results, anomalies which can arise within the models introduced earlier. Each anomaly type will be illustrated with a simple example, based on the functionalities outlined in Figures 2.5 to 2.7. By discerning the differences between each model and recognizing the associated pitfalls, one can select the right consistency mechanisms that align with our system's requirements.

Dirty Reads: an anomaly that occurs when a transaction reads data that has been written but not yet committed by another concurrent transaction – uncommitted data. This anomaly also identifies the case where a transaction reads aborted data.

To illustrate, consider the functionalities depicted in Figures 2.5 and 2.7, and a simplified transactional representation in Table 2.1. Imagine a scenario where transaction T_1 updates the Price and Discount properties of a product X_1 in the *Catalog* and *Discount* services. Concurrently, another transaction, T_2, is able to read the updated Price and Discount values before the transaction T_1 commits the changes to the database. In this situation, T_2 reads data that is, in essence, "dirty", leading to data inconsistencies. The scenario in which a transaction T_2 reads data that has been aborted by a transaction T_1 is illustrated in Table 2.2.

T1	T2
W(X_1)	
	R(X_1)
Commit	

Table 2.1: Dirty Reads anomaly: Read Uncommitted

T1	T2
W(X_1)	
Abort	
	R(X_1)

Table 2.2: Dirty Reads anomaly: Read Aborted

Fuzzy Reads: in this anomaly, a transaction reads different values for the same object at different times, leading to inconsistent data.

To illustrate, consider the functionalities depicted in Figures 2.5 and 2.7, and a simplified transactional representation in Table 2.3. Imagine a scenario where transaction T_1 reads the Price and Discount properties of product P_1, placed in the client's basket and represented by X_1. The basket contains 2 references to product P_1, requiring T_1 to read product P_1's Price and Discount again. Concurrently, transaction T_2 updates product P_1's Price and Discount values, identified by version X_2. When transaction T_1 reads the second reference to product P_1, the data read is inconsistent with the first Read operation, as T_1 now reads X_2.

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

Table 2.3: Fuzzy Reads anomaly

Fractured Reads: an anomaly that occurs, usually associated with database shard replication and weaker consistency policies.

For instance, consider the scenario depicted in Figures 2.5 and 2.7, along with a simplified transactional representation in Table 2.4. Imagine a scenario where transaction T_1 updates the Price (X_1) and Discount (Y_1) properties of product P_1. After T_1 successfully commits its updates to the data storage, another transaction T_2 updates the Price (X_2) and Discount (Y_2) properties of the same product P_1, and also successfully commits its updates to the data storage. A Fractured Read anomaly occurs if a subsequent transaction T_3, when reading the Price and Discount properties of product P_1, observes price X_2 and discount Y_1. In this case, the result only captures partial transactional updates, leading to inconsistent data states.

Lost Update: an anomaly that occurs when two concurrent transactions read the same data and subsequently attempt to update it with different values. In this scenario, one of the updates is lost, being overwritten by the update executed by the other transaction.

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

Table 2.4: Fractured Reads anomaly

Consider the situation illustrated in Figure 2.6, accompanied by the simplified transactional representation in Table 2.5. Imagine a scenario where a transaction T_1 executes the *AddProductToBasket* functionality, aiming to add a new product P_1 to basket X. After obtaining the product's information, it reads the current content on basket - X_1's content. Concurrently, and before the transaction T_1 can update the data read on X_1, another transaction T_2 modifies the same basket with data X_2, adding a new product P_2 to the basket's content. Subsequently, transaction T_1 proceeds to add the product P_1 to the basket. However, as T_1 only considers data read from X_1, the new update on basket X overwrites the update made by T_2. This results in the loss of the update performed by T_2.

T1	T2
R(X_1)	
	W(X_1 + P_2)
	Commit
W(X_1 + P_1)	

Table 2.5: Lost Update anomaly

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.

Consider the microservices outlined in Figures 2.2 and 2.3, along with the simplified transactional representation in Table 2.6. Picture a scenario where two transactions are concurrently executing Price and Discount updates in preparation for an upcoming commercial campaign. Transaction T_1 reads the price linked to product P_1, identified by version Y_0. Concurrently, transaction T_2 reads the discount associated with product P_1, marked by version X_0. Recognizing the high price, transaction T_1 increases the discount value, leading to an update associated with version X_1. Similarly, transaction T_2 opts to reduce the price, due to a low discount, resulting in an update associated with version Y_1. Depending on the real order that these transactions are executed, in the presence of an invariant that mandates a

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

Table 2.6: Write Skew anomaly

minimum price after discount for all products, such concurrent transactions might violate the system's integrity.

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 Figures 2.5 and 2.6 and Table 2.7. Imagine a scenario where a process executes a transaction T_1 that updates the Price (X_1) and Discount (Y_1) properties of product P_1. Concurrently, another process executes a transaction T_2, responsible for adding the product P_1 to the client's basket. 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
W(X_1)	
	W(X_2)

Table 2.7: Real Time Violation anomaly

We conclude this analysis by presenting of a table illustrating the anomalies prevented by specific consistency models. This highlights a clear connection between the consistency strength and isolation levels.

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

(Note: Shields represent protection against the anomaly, checkmarks represent vulnerability to the anomaly.)

Table 2.8: Consistency Models and Isolation Levels: a comparison of Anomaly Prevention Capabilities.

2.7 Addressing Consistency Problems in Microservices

As depicted in Table 2.8, 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 [31].

The Saga design pattern, introduced in Garcia-Molina and Salem [4] provides a framework for the management of data consistency for systems that build their transactional context within a distributed environment. This design pattern guides programmers in structuring their application code in a way that helps to mitigate the effects of anomalies and lack of atomicity. To achieve this, the pattern divides a functionality into a series of local transactions, each executed within individual services. Each participating service in the Saga is responsible for sending a message or publishing an event upon the completion of its local transaction, information which is then captured at the next microservice(s) in the microservices graph representing the functionality. This triggers the next local transaction in the Saga sequence. In any of these local transactions abort due to an anomaly, the Saga pattern initiates a set of *compensating actions* that aim at re-establishing the consistency of the application [32]. Figure 2.8 illustrates a typical execution of a functionality modeled following the Saga pattern, showcasing the detection of an anomaly during a local transaction. This anomaly results in the functionality being aborted, triggering the execution of corresponding compensating actions.

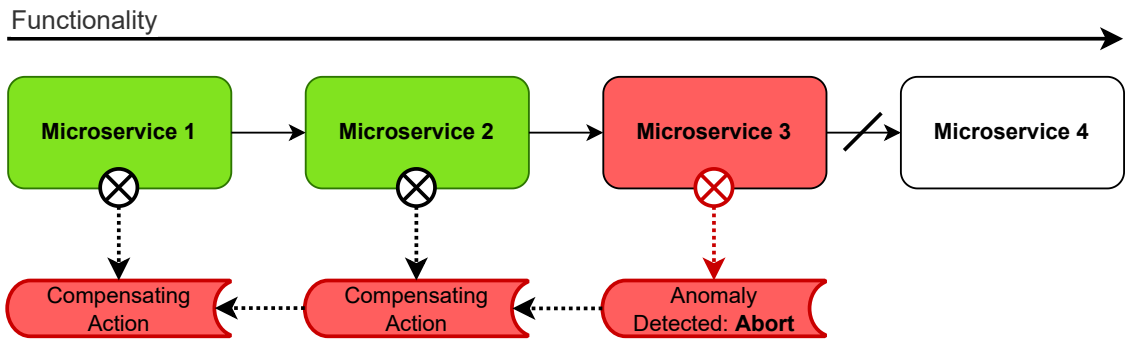

Figure 2.8: Saga pattern: sample functionality

Let's consider the *UpdatePriceAndDiscount* functionality as presented in Figure 2.5. Imagine a scenario where a business campaign is in progress, and the shop administrator intends to raise both the Price and the associated Discount of an item. Following the Sagas pattern, the functionality is divided

into two distinct local transactions—one executed in the *Catalog* microservice, and the other executed in the *Discount* microservice. In this context, the first local transaction successfully executes, increasing the Price of a specific product. However, at the *Discount* service, the system rejects the proposed Discount, deeming the resulting price after discount unacceptable (violating a pre-defined minimum, for example). Consequently, the functionality is aborted. In such a scenario, a compensating action is triggered to revert the updated price, ensuring that no product remains registered in the system with an invalid price/discount combination.

Typically, there are two common saga implementation approaches:

- **Choreography**: following the choreography approach, the saga participants exchange the messages and events between each other without a central point of coordination. The local transactions that finish their execution successfully, directly trigger the following transactions.
- **Orchestration**: following the orchestration approach, a centralized entity personally triggers the local transactions in each microservice. Each service can still publish a message or event once it finishes its execution. However, instead of being subscribed by the other services, only the orchestrator acts upon its receival. The failure recovery through compensating actions is also coordinated by this centralized entity.

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.

2.8 Exploring Transactional Consistency: Systems Overview

In this section, we analyze several systems that offer transactional consistency models in either monolithic or microservice architectures. Each system analysis includes a concise overview of its features, including both the isolation levels and the proposed solution. We delve into the key features and advantages of these systems, while also addressing potential limitations and considerations associated with these approaches.

2.8.1 Cloud Systems with Support for Transactions

2.8.1.1 Cure

Cure [27] was the first system to implement 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 different servers, where each server stores a non overlapping subset of the key-space. Cure also assumes that clients are *sticky*, i.e., that clients access a single datacenter, for all the operation in a transaction. 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) [33], 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 (partitions of a datacenter) are equipped with a physical clock that generates increasing timestamps and that are loosely synchronized with other clocks following a time synchronization protocol such as NTP [34]. 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. This means that, if the client has observed a snapshot more recent than the one available at the local partition, the coordinator is required to retain the transaction. After this condition is ensured, $V[j]$ 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. Considering that all updates become immediately visible to the client, in the presence of a network partition between datacenters, only the transactions executed in remote datacenters are delayed.

2.8.1.2 FlightTracker

FlightTracker [35] provides a solution for managing RYW consistency for clients accessing Facebook's social graph. It offers a system that maintains read efficiency, tolerance for hot spots (frequently read and updated objects), and high availability akin to Eventual Consistency. FlightTracker is designed as a set of APIs and a metadata service. To support RYW, FlightTracker collects metadata associated with a user's recent writes, presented as a data type denominated *Ticket*. Web requests always retrieve the user's Ticket before executing queries on Facebook's data stores.

The decision to embrace a weaker consistency model, as opposed to opting for Linearizability or Causal Consistency, is justified in the tradeoff analysis outlined in [35]. It is proposed that even though stronger consistency models prevent more anomalies, and let developers create mental models more easily, these also impose tighter constraints on service implementation. In the context of FlightTracker, a read-intensive system, the majority of queries results stem from 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. This information allows any generic infrastructure to monitor and recognize writes across numerous independently deployed systems. Included in these details are 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 [35] 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.

2.8.1.3 Clock-SI

Clock-SI [36] introduces a protocol ensuring SI for partitioned data stores. Unlike conventional systems, which depend on a centralized timestamp authority for deriving database snapshots and commit timestamps, Clock-SI utilizes loosely synchronized clocks specific to each data store partition. This design choice eliminates the risks associated with having a single-point of failure and avoids potential bottlenecks during periods of heavy workloads.

A transaction begins when a client connects to an *originating partition*, determined by a load balancing scheme. The originating partition executes the transaction operations sequentially. If the partition does not store the required data for an operation, it is executed remotely at the appropriate partition. The originating partition assigns a snapshot timestamp to the transaction using its local clock. During the commit phase, if the transaction only involved local operations and no write conflicts occurred, the local clock's value becomes the commit timestamp. If the transaction spans multiple partitions, a 2-Phase Commit protocol is initiated, with the originating partition serving as the coordinator. All involved partitions check for write conflicts before proposing the commit timestamp. If conflicts are detected, the transaction is aborted. The coordinator selects the highest timestamp as the final commit timestamp, and all partitions commit the transaction using this selected timestamp.

When executing read operations, a partition may need to temporarily block the transaction. There are two possible cases where this behavior might occur:

- If a concurrent transaction T' is in the process of committing, has updated the data object being read, and the snapshot timestamp from which the client is reading surpasses T''s commit timestamp. In this situation, the read operation might need to incorporate the results of the committing transaction to maintain consistency, so the executing partition blocks the transaction.

- Due to partitions adhering to a loose synchronization protocol such as NTP [34], clock skewing among involved partitions might occur. If the local clock of the partition executing the read operating is lower than the transaction's snapshot timestamp, the partition must block the transaction until the local clock aligns with the snapshot timestamp. This pause is essential to prevent any other transactions from being committed between the local clock timestamp and the transaction's

snapshot timestamp.

To mitigate the delays caused by transaction blocking, the protocol outlined in Clock-SI proposes that clients use an older snapshot. This approach involves a trade-off between increasing the likelihood of transaction aborts, since extending the time gap between the snapshot and commit time provides an opportunity for other concurrent transactions, updating parts of the same key set, to commit; and minimizing transaction blocking due to time skewing. The analysis presented in Du et al. [36] indicates that mechanisms that allow clients to use older snapshots are particularly valuable for read-only transactions. Although allowing clients to use older snapshots reduces the likelihood of delayed transactions, it comes at a cost: transactions may more frequently encounter stale data, leading to transactional aborts.

2.8.2 FaaS Systems with Support for Transactions

2.8.2.1 Hydrocache

Hydrocache [37] introduces a distributed caching layer capable of simultaneously ensuring low-latency data access and TCC to FaaS applications. Hydrocache protocol does not rely on membership of the system and is transparent to the FaaS autoscaling capabilities. To disregard the need for membership, dependencies are managed at the key level.

When executing in a single node, to achieve both causal snapshot reads and atomic visibility, Hydrocache imposes that each cache maintains a single *strict causal cut*, for which is it stated that if a key is in the cut, all its dependencies are also in the cut. The strict causal cut is more stringent than a causal snapshot, as the strict causal cut requires that any concurrent updates cannot enforce their key dependencies, meaning that for any pair of keys a_i,b_j, part of the same causal cut, if $a_k \rightarrow b_j$, then either $a_k == a_i$ or $a_k \rightarrow a_i$. Generally, a key dependency must either: happen after all key dependencies associated with other keys in the cut, or be the exact same key version. To ensure Atomic Visibility, keys updated in the same transaction are made mutually dependent of each other, thus having the strict causal cut enforce atomicity by include all key dependencies. When executing a transaction (represented by a Directed Acyclic Graph (DAG)), to ensure that all updates are mutually dependent, write operations are only performed at the sink function of the DAG, where they are persisted to remote storage.

Despite being able to offer TCC to individual functions, the protocol described is insufficient for serverless applications, composed of multiple functions, where each can be executed in a different physical node. To extend the use of TCC to multiple nodes, Hydrocache proposes 3 protocols: *optimistic*, *conservative* and *hybrid*.

The optimistic approach eagerly starts running the functions in a DAG, performing violation checks of the snapshot property when each function is executed. Before a function is executed, it verifies if

the available snapshot is compatible with the previous function's read set metadata. This metadata, which includes both the read set and the write set of the transaction, is sent across functions. If an incompatibility is found between the causal cut and the received read set, the executing function can try to modify the read set to suit the snapshot. This is done including the missing key in the causal cut, ensuring first that all the fetched key's dependencies are already included in the cut. However, if this is not possible, the transaction is aborted. This verification is applied in cases where a function receives outputs from multiple parallel upstream functions. If the received read sets from the upstream function are not compatible, the transaction is aborted. By building the snapshot along the execution of the transaction, without prior coordination, the results show that this approach, albeit prone to aborting more frequently, achieves low average response times.

The conservative approach instead opts for coordinating all caches involved in a transaction to build the snapshot to be used, ensuring that the transaction never aborts. This approach introduces additional efforts, as coordination with all caches in the DAG is required in order to build a distributed snapshot containing all the function's read sets. This additional step translates into higher average response times.

The hybrid approach combines the benefits of both the optimistic and conservative approaches. The process starts with the execution of the optimistic approach, while simultaneously a simulation of the optimistic dependency checks is performed. This is possible because the optimistic validation only needs the read set of each function and the local cut at the cache level. Since the simulation does not require the actual execution of the functions involved in the transaction, and read sets don't need to be transmitted along functions, this phase is executed in a much faster manner than in the original optimistic approach. If the transaction is aborted, the conservative approach is initiated. This approach restricts Hydrocache to only support transactions where the entire read set is known beforehand.

Throughout the execution of the DAG, updated objects are stored along with metadata that captures the explicit dependencies of their write transaction. The information regarding read objects is also transmitted across the different nodes of the DAG, in the form of metadata as well. This ensures that all executed functions observe key versions and their respective dependencies from a consistent causal cut. This approach can be detrimental to the performance of the system, considering the high volume of metadata that is transmitted across the different functions.

2.8.2.2 FaaSTCC

FaaSTCC [38] offers the TCC consistency model to 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 [27] 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 [37], that provides TCC in the FaaS environment, 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 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.

2.8.3 Microservice Systems with Support for Transactions

2.8.3.1 Antipode

Antipode [39] introduces a bolt-on technique to prevent cross-service violations in distributed applications. This approach involves propagating *lineages* of data store operations along end-to-end requests and within data stores. These lineages capture a tree of events across different services, correspond-

ing to the branching resulting from each service's invocations. Alongside the use of lineages, Antipode introduces the concept of Cross-Service Causal Consistency (XCY) – an extension to existing causality models, rooted in Lamport's idea of *happened-before* relations.

Compared to CC, Antipode extends this model by stipulating that if a read operation b from one lineage reads the results of a write operation a from a different lineage, then any subsequent operations dependent on b must observe the entire lineage to which operation a belongs, not just operation a. In summary, XCY consistency restricts Lamport's causality model by requiring read operations to incorporate causal dependencies from entire lineages and not just the operations that originated the read data objects.

To manage dependencies within lineages, Antipode employs the use of two different tracking methods: *Implicit dependency tracking* and *Explicit dependency tracking*. Implicit dependency tracking typically utilizes context propagation tools for distributed tracking. These tools enable automatic dependency tracking across traversed service graphs that use message passing for interaction between services. However, these tools are insufficient for ensuring accurate propagation and tracking of dependencies through replicated data stores. This limitation arises due to the existence of transactions that involve read and write operations over the same key set on different partitions. Consider the scenario where an object is updated in one partition, and before the updated value is persisted on the other homologous partitions, an older version of the same object is observed by the same transaction. This type of event can happen because data replication across partitions happens asynchronously.

To tackle this challenge, Antipode proposes a bolt-on approach, where interactions with data stores exclusively occur through a shim-layer. This approach ensures that data stores remain transparent to the proposed mechanisms. The shim-layer enables Antipode to capture dependencies across replicas of the traversed data stores.

The second approach, explicit dependency tracking, allows developers to exceptionally selectively add or remove dependencies from the lineage context that might not have been automatically detected. Antipode also introduces additional procedures to control the volume of dependencies transferred between lineages, including *stop* and *transfer*. These procedures let developers manually drop the ongoing set of dependencies or explicitly transfer the set from one lineage to a subsequent one.

Apart from tracking dependencies, Antipode ensures the enforcement of dependencies across services. In conventional systems, these dependencies are typically enforced implicitly, requiring services to enforce the set of dependencies in every operation. However, this approach introduces a significant drawback: it necessitates frequent cross-service communications, leading to additional overhead. To address this, Antipode adopts explicit dependency enforcement, attributing the responsibility of defining the dependency enforcement locations to the developers. In this manner, developers are tasked with assigning *barrier* calls at specific points in their applications. Upon reaching these barriers, the system

enforces an order of operations that is consistent with the XCY consistency model. It does so by halting the execution until all writes contained in the lineage are visible in the underlying data stores. Particularly in systems where user experience is crucial, these mechanisms allow developers to finely balance dependency enforcement.

Concurrently with the development of this thesis, a related work supported by Antipode's findings and aimed at the FaaS environment has been published as Rendezvou [40]. This research strives to enforce consistent cross-function view of application data. Tailored for a geo-replicated context, Rendezvou extends the previous work, providing a framework capable of automatically ensuring data consistency for serverless applications.

2.8.3.2 Enhancing Saga

Enhancing Saga [41] addresses a key drawback of the Saga pattern: lack of Isolation. The Saga pattern, as discussed earlier, only offers *ACD* properties (*Atomicity*, *Consistency* and *Durability*), missing the crucial "I" of *Isolation* in distributed transaction executions. Without Isolation, the Saga pattern is susceptible to both Dirty Reads and Fuzzy Reads anomalies.

Enhancing Saga proposes a solution by introducing in-memory data caching to address the lack of read-isolation inherent in the Saga pattern. The system allocates a *quota* of the database storage space to a designated memory cache server, referred to as the *Quota Cache*. This cache server is responsible for storing the results of Create, Read, Update and Delete (CRUD) operations. Instead of immediately committing changed objects to the database, as in the original Saga pattern, this memory cache server retains the values until the microservice receives an order to commit to the data store. This prevents other concurrent transactions from reading uncommitted values.

It's important to note that microservices employing the quota cache benefit from low latency and high throughput. This is because the results of read operations to the remote storage are stored in the cache for future reads. The Enhancing Saga solution associates the cache server only with microservices that require the execution of compensating actions in case of transaction abort. This decision is based on the observation that some microservices either exclusively perform read operations or execute idempotent operations.

The system employs an orchestrator module to configure every microservice with its corresponding web clients. Saga coordination is achieved through two approaches: orchestration and event-choreography. In the orchestration approach, a manager controller oversees all communications among microservices. On the other hand, the event-choreography approach is employed as microservices complete their local transactions. Following this approach, after a local transaction concludes, the responsible microservice generates an event consumed by the subsequent microservices in the functionality.

Once all microservices involved in the functionality complete their local transactions, the coordinator

initiates the commit process. To ensure atomically visibility of changes, an event is dispatched to the microservices that updated objects in that functionality, indicating that it is safe to commit. These microservices are then responsible for executing an eventual commit sync service to carry out the commit. This commit sync service synchronously holds the commit decision until it is confirmed whether all local commits were successful or if any local transaction failed. If any transaction fails, compensating actions are triggered to revert the changes made in the caches.

2.9 Analysis

In this section, we discuss the advantages and disadvantages of the systems surveyed in the previous section. Table 2.9 provides a comparative analysis of the features of non-microservice systems supporting transactions.

Systems	Target	Consistency	Metadata Size	Read RTT	Commit RTT
FlightTracker	Cloud	RYW	*O(W)*	-	-
Antipode	Cloud	XCY	*O(W)*	-	-
Cure	Cloud	TCC	*O(N)*	1*	3*
Clock-SI	Cloud	SI	*O(1)*	2*	3*
FaaSTCC	FaaS	TCC	*O(1)*	1*	2
Hydrocache	FaaS	TCC	*O(K)*	1*	2

(Note: W represents write-set, N represents the number of partitions, K represents the key-space, * means that the time period might include blocking/waiting or aborts.)

Table 2.9: Cloud and FaaS Systems Comparison.

2.9.1 Target Environment

In Table 2.9, 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, FlightTracker or Clock-SI are designed for environments with a static number of data centers and partitions. On the contrary, FaaS systems such as FaaSTCC or Hydrocache are designed to facilitate the auto-scaling, and therefore their components are bounded to be changed frequently.

2.9.2 Consistency Guarantees

The analysis illustrated in Table 2.9 organizes the systems based on consistency strength, grouped by their target environments. Naturally, as the consistency guarantees become stronger, the performance

of the surveyed systems degrades die to the utilization of stricter coordination mechanisms. These mechanisms often involve the use of 2-Phase Commit protocols, as is seen in systems like Cure and Clock-SI. This necessary step introduces one or more additional Round Trip Time (RTT) to the overall latency of the commit phase.

2.9.3 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, FaaSTCC and Hydrocache offer TCC. While Cure provides TCC for sticky client sessions, FaaSTCC and Hydrocache provide 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 and Hydrocache, on the opposite, require 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. Unlike FaaSTCC, Hydrocache requires the exchange of large volumes of metadata, kept for each object read or written in a transaction.

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 Clock-SI system ensures SI for partitioned data stores, using loosely synchronized clocks unique to each data store partition. Transactions span local and remote partitions given the distribution of data, using a fixed snapshot of the data store to guarantee consistency. The commit protocol employs a coordination scheme where a single partition selects the commit timestamp for all involved partitions in the transaction, executed through a 2-Phase Commit protocol.

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.

2.9.4 Transaction Models in Microservice Architectures

We analyzed two systems capable of offering transaction semantics in the microservice architecture. The Saga pattern is considered the standard approach 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 rely 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, failing to guarantee that functionalities always observe mutually consistent values. To achieve this, it's vital that all data accessed within each microservice is consistent (in-service consistency) and that data read from other microservices is also mutually consistent (inter-service consistency). The Antipode proposal introduces a bolt-on technique designed to prevent cross-service consistency violations, extending a stronger consistency policy than Causal Consistency (CC), but weaker than Transactional Causal Consistency (TCC). Ensuring consistency when microservices do not maintain local replicas of data managed by other microservices mirrors the problem tackled by Clock-SI. If microservices receive updates for replicated data values, and maintain part of these updated values in a local cache, then the proposed mechanisms introduced both in Cure and FaaSTCC are also pertinent. In any case, these solutions impose that clients executing a functionality carry metadata capturing the causal cut of the data store from which they operate to guarantee inter-service consistency. The information provided with FlightTracker demonstrates the feasibility of this, even in large-scale systems with strong performance demands, such as Facebook.

Summary

This chapter addressed the intricate challenges inherent in maintaining specific consistency levels within the transactional context, in both Cloud and FaaS systems. Typically, microservice-based systems lean towards weak consistency policies such as Eventual Consistency, as this model ensures high availability and the desired loose coupling that is often sought after in this architecture. However, this consistency model is susceptible to consistency anomalies, requiring the executing of compensating actions to restore the consistency across the affected services, thereby impacting system performance. Recent works have been designed to offer stronger consistency models, such as TCC in similar environments

to the microservice architecture, such is the case of FaaS. To the best of our knowledge, no previous work has addressed the support of TCC for microservices efficiently. In the following chapter, a novel system that is capable of extending TCC to the microservice architecture is proposed, aimed at guaranteeing both high-availability and low latency, qualities intrinsic to this architecture paradigm.

3

μTCC

Contents

In this chapter, we present μTCC, a middleware layer that offers Transactional Causal Consistency (TCC) support for microservice architectures. Section 3.1 describes the goals our system aims to accomplish. Section 3.2 describes in detail the components that constitute μTCC. Section 3.3 describes the protocols used by μTCC to offer TCC to microservice applications. Section 3.4 describes the garbage collection system employed in μTCC. Finally, Section 3.5 explains the necessary efforts to integrate μTCC in existing microservice applications.

3.1 Goals

The primary focus of μTCC was to establish the essential mechanisms for enabling TCC in microservice applications. This involved guaranteeing that all values that originated from operations within a functionality are applied atomically across all microservices and ensuring that within the functionalities, microservices always read versions of data objects that are mutually consistent. μTCC focuses on delivering these guarantees while securing both high-availability and low additional overhead, key properties of the microservice architecture.

3.2 Architectural Details

When developing μTCC, we focused on creating a transparent solution for the application, minimizing the adaptation of existing microservice systems for its adoption. To achieve this, μTCC was designed as a middleware layer composed of a set of wrappers intercepting requests between microservices and requests from microservices to their respective data storage. These wrappers add metadata capable of maintaining a consistent causal cut during transaction execution. Specifically, as depicted in Figure 3.1, μTCC consists of: Storage Wrappers, Microservice Wrappers, and Functionality Coordinators. In the following sections, we describe the functioning and implementation of each of these components.

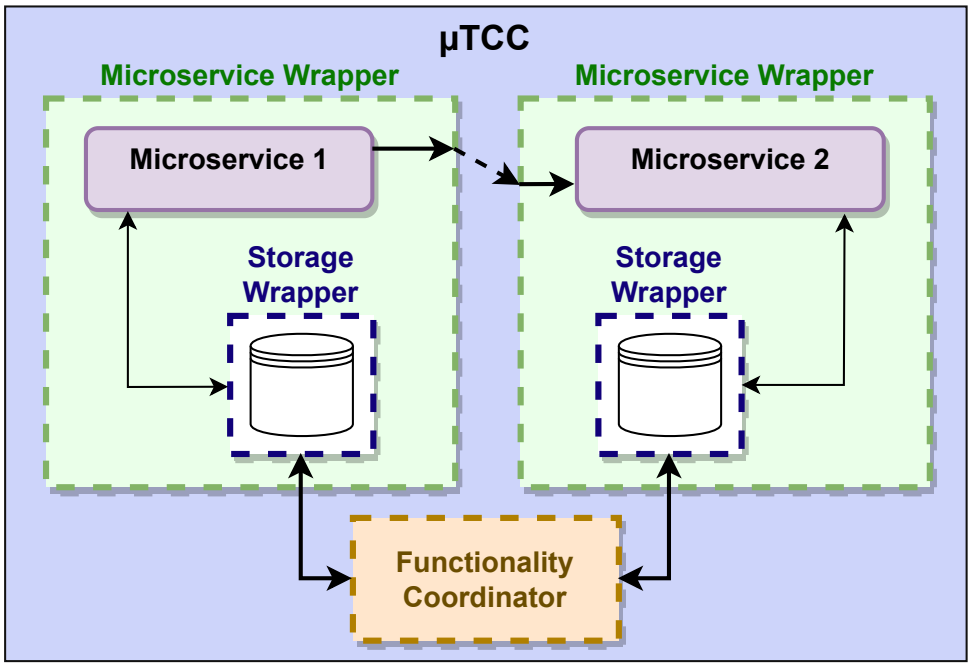

Figure 3.1: μTCC System Architecture

3.2.1 Storage Wrappers

Each storage system used in the context of μTCC is encapsulated by a wrapper that mediates all read and write operations performed by the microservice. To extend Transactional Causal Consistency (TCC) to microservices, we need to ensure that transactions read from a causally consistent snapshot. To be able to identify the causal order between transactions, we associate a timestamp for each committed transaction. The storage wrapper is responsible for this association. The value of the timestamp to associate with each transaction is decided during the confirmation phase. By associating a timestamp to a version of a data object, we are able to store multiple versions of the data, even when the underlying storage system does not support multiversioning natively. Although storing multiple versions of the same data object is not a requirement of μTCC, it can significantly improve our performance, as it will be examined in Chapter 4.

During write operations, the storage wrapper is responsible for locally storing all updates made within the context of a functionality in a local cache, until the functionality is confirmed. Similar to the Enhancing Saga system, as described in Daraghmi et al. [41], values locally stored in this manner remain visible only for invocations made within the context of that functionality.

In read operations, based on the metadata associated with a given execution, retrieves from the local cache or the remote storage system, a version belonging to consistent cut visible for that execution.

During the confirmation phase of a functionality, it negotiates, with the help of the coordinator, the timestamp that will be associated with the version of the data to be persisted in the remote storage. During the negotiation process, access to some version might be blocked. If a functionality aborts, the wrapper simply discards the version kept in the local cache, without ever persisting them.

In this context, a wrapper was implemented specifically for the Microsoft SQL Server storage system, a relational database management system that employs Structured Query Language (SQL) to construct relational schemas for each database in the system. Given the extensive set of statements and functions offered by SQL, the developed wrapper focuses only on a fraction of these available statements and functions, used to evaluate the analyzed system in Chapter 4. This way, the wrapper effectively manages three fundamental SQL statements: *SELECT*, *INSERT* and *UPDATE*. It also supports several clauses, including *WHERE*, *ORDER BY*, *TOP*, *OFFSET* and *FETCH NEXT*. Additionally, the wrapper incorporates support for the *COUNT* aggregation function.

3.2.2 Microservice Wrappers

Besides the lack of Isolation, to extend TCC to the microservice architecture, we need to guarantee the atomicity of the functionalities. This challenge is notably highlighted by the dilemma concerning a functionality's state. Since each microservice is naturally independent of others, it is crucial to deter-

mine when a functionality concludes and which microservices were involved in its execution to proceed with the atomic confirmation of the updated data in each microservice. To solve these challenges, we developed the microservice wrappers.

This microservice wrapper encapsulates all microservices used in the context of μTCC. It mediates interactions with other microservices and the functionality coordinator. This wrapper is responsible for interpreting and maintaining the metadata that captures the consistent causal cut visible to a functionality, ensuring that this metadata is kept updated and passed to the storage wrapper transparently to the application.

Additionally, this wrapper is responsible for passing, between microservice invocations, a token that captures the fact the functionality is in execution. This token can be fragmented when a microservice invokes more than one downstream microservice. If a microservice is a leaf in the microservices graph representing the functionality, token fragmentation does not occur and is sent to the functionality coordinator, which acts as a sink for tokens generated during execution. Along with the token, this wrapper also informs the coordinator whether the local transaction can commit or has been forced to abort.

3.2.3 Functionality Coordinators

The Functionality Coordinator monitors the execution of a functionality and coordinates the data confirmation process when an execution completes successfully. Our system assumes that the coordinator can operate in microservice-based systems using orchestration or choreography.

Consider a variation of the example presented in Figure 2.5, where the Frontend service acts as the Coordinator. If the functionality coordinator operates in orchestration mode, all microservice invocations are made by the coordinator, eliminating the need for a token to detect functionality termination, as is illustrated by Figure 3.2.

In choreography mode, the execution of a microservice can trigger multiple other microservices. In this case, tokens are utilized, forwarded to the coordinator at the end of each microservice execution. The coordinator detects the end of the graph execution after receiving all fragments of the original token, as is illustrated by Figure 3.3.

In both scenarios, if all microservices have completed their execution and are in a position to confirm the execution, the coordinator initiates the confirmation process involving all relevant storage wrappers. If the execution needs to abort, it instructs the storage wrappers to discard the corresponding updates. The confirmation phase is illustrated in Figure 3.4.

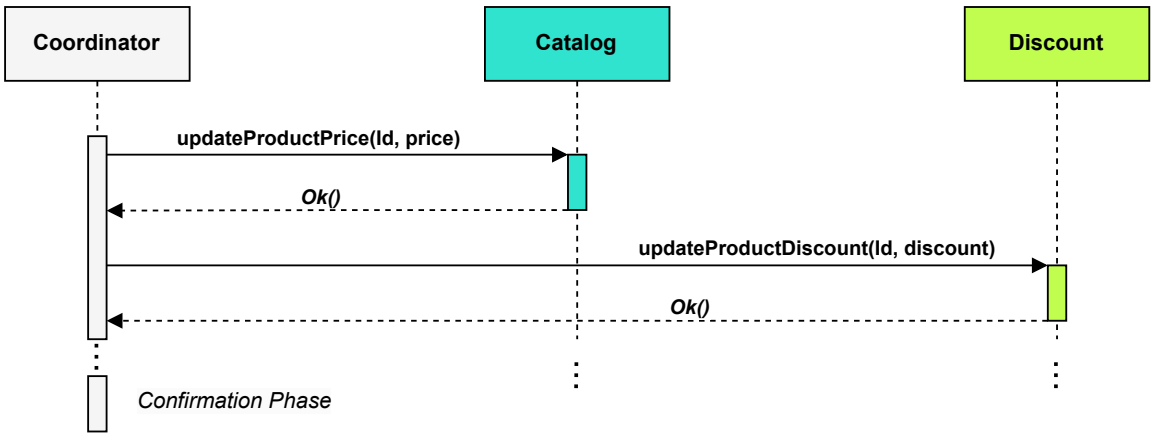

Figure 3.2: Update Price And Discount functionality: Orchestration approach

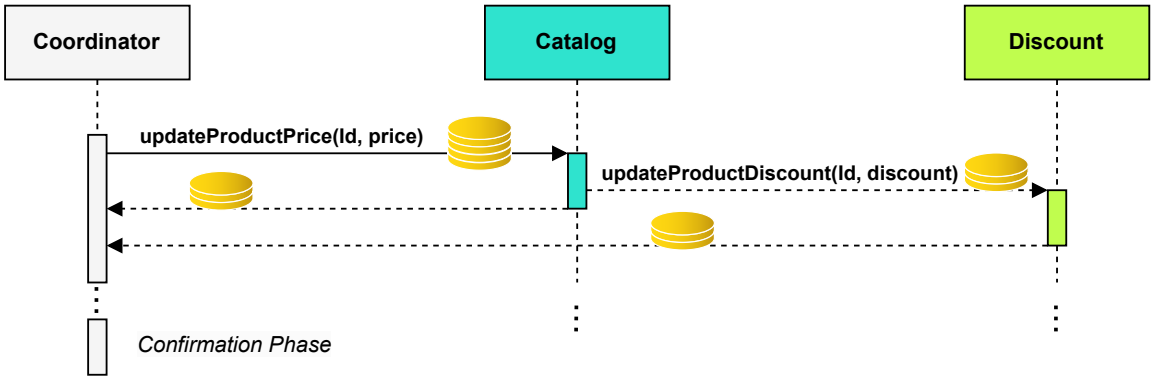

Figure 3.3: Update Price And Discount functionality: Choreography approach

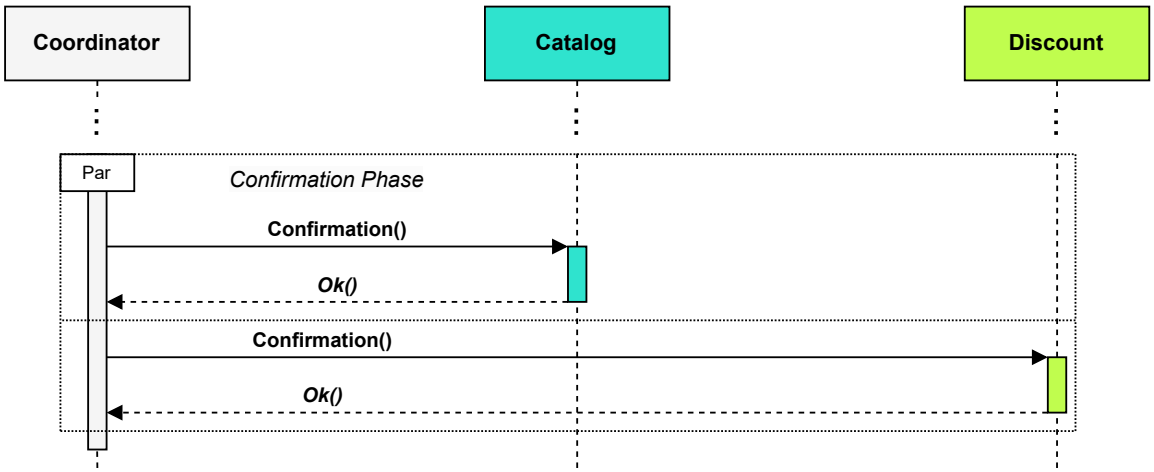

Figure 3.4: Update Price And Discount functionality: Confirmation Phase

3.2.4 Metadata

Leveraging the existing communications between microservices executed within the context of a functionality, μTCC injects metadata to ensure that the consistency of read operations is respected, and that write operations are identified with a unique timestamp assigned during the confirmation phase. In addition to the token fraction, the additional information is transmitted. This includes a timestamp, defining the most recent version of objects that any microservice can read without compromising the functionality's consistency. This timestamp is established at the beginning of the functionality, before the first microservice is invoked.

Furthermore, a unique client identifier, apparent to the system and generated at the onset of the functionality, is sent. This identifier serves both for read and write operations. Concerning write operations, it is employed to identify versions not yet confirmed to the client who generated them. Regarding read operations, the client identifier ensures that versions of objects generated during the functionality, but not yet persisted in the storage system, remain visible to the client.

3.3 Protocols

In this section, we detail μTCC's protocols. The algorithms presented during this section include the pseudocode describing the protocols that are executed in the Microservice and Storage wrappers.

3.3.1 Token Processing

The tokens are used by the microservice wrappers to solve the state of a functionality dilemma. They are split and transmitted between microservice wrappers and sent to the functionality coordinator once the microservice finishes its local transaction.

All the algorithms used in μTCC to fragment the Token assume prior knowledge of two parameters: the maximum branching factor of the functionality execution graph (that is, the maximum number of invocations a given microservice can make), and the maximum depth of the graph. Based on these parameters, the root node of the execution graph receives a token with a defined number of fractions (Alg. 3.1, Line 4).

Algorithm 3.1: Token Initialization Protocol

```
1 b ← #Maximum_branching_factor
2 d ← #Maximum_depth
3 function Initialize_Token():
4 |   functionality_token ← (b + 1)^d
```

When a microservice is invoked, it receives a fraction of the initial token from its parent. Before proceeding with the execution of its business logic, the microservice wrapper reserves a fraction of the

token for itself (Alg. 3.2, Line 3). This step is executed to ensure that, after invoking all microservices, the invoker microservice can also notify the coordinator about its participation in the functionality. The remaining fractions of the token are evenly distributed among invocations to other microservices, if any. For each invoked microservice, the same fragmentation protocol is used, ensuring that enough fractions of the token are sent to each child. This process of fragmentation and collection of the token fractions are managed both automatically and transparently by the microservice wrapper. This mechanism is secure even if the values of the maximum branching factor and maximum depth are estimated incorrectly. If, during the execution of a functionality, the fractions of the token are exhausted, the functionality is simply aborted, and a notification is generated to reconfigure the system. Considering the functionality example present in Figure 2.7, the Maximum Branching Factor $= 2$ and Maximum Depth $= 3$, assuming the existence of a single item in the basket, and thus the initial token is created with 27 fractions.

Algorithm 3.2: Token Subdivision Protocol

```
1 b ← #Maximum_branching_factor
2 function Subdivision_Token(rcv_token):
      /* invoked microservice stores fractions of the token for itself  */
3     fractioned_token ← rcv_token / b+1
```

Additionally, it's worth noticing that apart from the proposed use of *Tokens*, the idea of employing checklists of microservices associated with each functionality, predetermined and known by the coordinator, was also explored. In this approach, to identify the moment when the functionality was ready for the confirmation phase, the coordinator would simply verify the completion status of the microservices on its local checklist. However, this method proved to be error-prone, especially in cases where the invocation of different microservices depended on runtime check conditions. This approach would require the coordinator to know beforehand which microservices would be executed in each functionality, a requirement not necessary in the *Token* approach.

3.3.2 Write Protocol

During the execution of a functionality, all write operations are stored locally in the storage wrappers before being confirmed using a two-phase commit protocol. μTCC makes use of synchronized clocks to associate a timestamp to each new data version. Our storage wrapper intercepts all write operations executed in the context of the microservice, processing them into local memory until given authorization to persist to storage.

Upon confirmation, each storage wrapper persists the update versions on the remote data storage. If the functionality is aborted during the confirmation phase, all new versions written during the execution of the functionality are discarded by the wrapper.

3.3.3 Read Protocol

A functionality observes the consistent state of the system at the timestamp defined in the beginning of the functionality. When reading an object, the storage wrapper retrieves a local version not yet confirmed if the client has performed writes within the current functionality context (Alg. 3.3, Line 7), or a version of the data belonging to the consistent cut of the used timestamp (Alg. 3.3, Line 13). Naturally, since the timestamps used in the read operations are defined using synchronized clocks, there is a chance for clock skewing between microservices. Just like in Clock-SI [36], a read operation may temporarily block due to the clock skew, in this case between microservices in the same functionality (Alg. 3.3, Line 4). To avoid this overhead, our protocol allows for the client to choose an older timestamp, sacrificing freshness for lower chances of read operation blocking. When trying to read data objects, the access to them might also be blocked during the confirmation timestamp negotiation phase, depending on the read timestamp of the functionality. If the read operation timestamp is higher than the timestamp proposed locally for the concurrent write operation, the read operation must wait for the final decision until the microservice confirms the new version of the object (Alg. 3.3, Line 12).

Algorithm 3.3: Read Protocol

```
1  function Read_Data(key, tx_TS, non_persisted_writes, client_id, proposing_clients):
2      conc_write_set ← ∅
       /* accounts for possible clock skew                                   */
3      if tx_TS > Clock() then
4          WAIT tx_TS ≤ Clock()
5      for ⟨ k, val, cli_id ⟩ ∈ non_persisted_writes do
           /* check the non-persisted write set to ensure Read Your Writes (RYW)  */
6          if k == key ∧ cli_id == client_id then
7              return val
8          else
               /* get all concurrent writes for the same key with lower timestamp  */
9              conc_tx_TS ← proposing_clients.GetTimestamp(cli_id)
10             if conc_tx_TS ≤ tx_TS then
11                 conc_write_set ← conc_write_set ∪⟨ cli_id, conc_tx_TS ⟩
12     WAIT ∄⟨ cli_id, prepare_timestamp ⟩ ∈ concurrent_write_set
13     return remote_storage.get(key, tx_TS)
```

3.3.4 Commit Protocol

As each microservice completes its execution, it sends its assigned token fraction, its client ID, its address, and an indication of whether it performed write operations during its execution to the functionality's coordinator. If the microservice has executed write operations, it should be contacted at the of the functionality to confirm its updates (Alg. 3.4, Line 7). On the coordinator's side, a structure associating

each client's unique identifier and the received token fractions is kept in memory. After retrieving all token fractions, the coordinator contacts all microservices involved in the transaction that executed write operations, asking each one to propose a confirmation timestamp for the functionality (Alg. 3.4, Line 10).

At this stage, each microservice proposes the current value of its physical clock as a confirmation proposal for the functionality. Alternatively, if any business logic invariants are violated, it informs the coordinator of its intention to abort the transaction. Furthermore, each microservice marks the updated objects present in the storage wrapper as objects in the process of being persisted. Access to these objects might be blocked during the confirmation timestamp negotiation phase, depending on the read timestamp of concurrent functionalities attempting to read their value.

Note that, due to the clocks not being perfectly synchronized, it's possible for a microservice to receive a message with a timestamp from the future, meaning a value higher than its own local clock. In such cases, the system delays the processing of that message until its clock is aligned with the message's timestamp. This procedure is used in most systems employing physical clocks [36, 42]. If any microservice signals the need for an abort, the coordinator instructs the microservices to discard the versions present in the storage wrapper associated with the respective client identifier. If all microservices confirm the execution, after receiving the clock values from all involved microservices, the coordinator computes the maximum clock value and sends it back to the microservices along with a confirmation order for the in-memory versions associated with the client identifier (Alg. 3.4, Line 16).

3.4 Garbage Collection in μTCC

As previously discussed, storage wrappers assign timestamps to each version of a written data object, enabling the establishment of a multiversioning scheme even in data storage systems lacking native multiversioning support. To handle the introduced overhead of storing multiple versions of the same object with unique timestamps, each storage wrapper employs an independent worker responsible for issuing delete commands to the remote storage. Periodically, for each data object in the remote storage, if the total number of versions N surpasses K, where K is a pre-configured limit of versions for each data object, the worker issues the deletion of the oldest $N - K$ versions. This approach presents a tradeoff: limiting the number of versions per data object enhances performance by reducing the dataset size in remote storage but increases the risk of read operation failures due to a lack of a consistent version in storage. Based on the experimental analysis outlined in Chapter 4, the data storage worker restricts each data object to 25 versions.

Algorithm 3.4: Commit Protocol

```
   /* State kept by the coordinator                                          */
1  coord.tokens ← ∅
2  coord.participating_micro_addresses ← ∅
3  coord.proposals ← ∅
4  function Receive_Token(token, client_id, address, read_Tx_flag):
       /* Increments the fractions of the token received                     */
5      coord.tokens ← coord.tokens + token
6      if read_Tx_flag == False then
           /* Adds the address to the list of participants                  */
7          coord.participating_micro_addresses ← coord.participating_micro_addresses ∪ address
8      if coord.tokens == total_fractions then
           /* Sends Proposals request to all participants                   */
9          for all service_address ∈ coord.participating_micro_addresses do
10             Send(⟨GetProposal⟩, client_id) to service_address

11 function Receive_Proposals(TS, client_id):
       /* Receives proposals from participants                              */
12     while number of coord.proposals ≠ number of coord.participating_micro_addresses do
13         coord.proposals ← coord.proposals ∪ TS
14     commit_TS ← max(coord.proposals)
       /* Issues commit order to participants                               */
15     for all service_address ∈ coord.participating_micro_addresses do
16         Send(⟨Commit⟩, client_id, commit_TS) to service_address
```

3.5 Cost of Adopting μTCC

The implementation of μTCC requires minimal adaptations to the base code of each microservice. Specifically, this includes adapting the database context class to include our storage wrapper and integrating our microservice wrapper into the existing stack of wrappers used by the original system. These adaptions can be automated. It is important to note that the microservice wrapper is independent of the nature of the original application and can be automatically generated. Therefore, the adoption cost of μTCC is primarily associated with the development of storage wrappers, which are specific to each persistence system and/or database.

Using the Cloc tool [43], we quantified the size, in lines of code, of the implemented wrappers and any related data structures responsible for managing metadata transmitted between wrappers. Every microservice wrapper, written in *C#*, consists of 193 lines of code, a constant value across all microservice wrappers due to the wrapper's agnostic view over the service's business logic. The storage wrapper, also written in *C#*, comprises 1749 lines of code. This value remains constant regardless of the microservice business logic's complexity, being correlated solely with the number and type of remote storage systems associated with each microservice. Specifically, the storage wrappers developed intercepted requests directed to a single SQL-based datastore. Additionally, we assessed the lines of code

in the implemented garbage collection mechanisms utilized by microservices that use storage wrappers. Each of these garbage collection mechanisms comprises 47 lines of code.

Summary

μTCC extends the transaction support for TCC to microservice applications through the integration of microservice and storage wrappers, along with functionality coordinators. Utilizing the proposed write protocol, these mechanisms ensure that all updates are applied atomically across the microservices involved in the functionality that originated the updates. Additionally, the proposed read protocol allows clients to read mutually consistent versions of data objects within functionalities. μTCC focuses on delivering these features while ensuring high-availability and low additional overhead.

4

Evaluation

Contents

In this chapter, we evaluate the performance of μTCC. Section 4.1 describes the main goals of the evaluation analysis. Section 4.2 describes the workload and workbench used to test μTCC, along with the reference application where we introduced μTCC. Section 4.3 depicts the test scenarios approached to evaluate the system. Section 4.4 demonstrates how effectively μTCC is able to prevent TCC anomalies. Section 4.5 introduces the overhead costs of using μTCC in pre-existing applications.

4.1 Experimental Goals

The goal of the evaluation is to assess the implications of integrating μTCC into microservice-based applications. Specifically, we seek to comprehend the advantages and limitations associated with ex-

tending Transactional Causal Consistency (TCC) to microservice-based systems, when contrasted with weaker consistency models such as Eventual Consistency. The evaluation of μTCC revolves around several key questions:

- To what extent do TCC consistency anomalies occur, and how effectively does μTCC prevent them?
- What impact does the introduction of μTCC have on the performance of microservice-based application?

4.2 Experimental Workbench

Our evaluation is based on the executions of the *eShopOnContainers* [44] application when running on μTCC. This application is composed of a suite of microservices, each running within a Docker container. For the experiments reported in this article, we deployed all containers on a single physical server equipped with an Intel Xeon Silver 4314 CPU with 32 logical cores, 197 GB of RAM, and 100 GB of SSD storage. The clients of our application were also run on the same server.

The *eShopOnContainers* application simulates an online store that allows customers to search for items, add products to shopping carts, and proceed with their payments. The application consists of various components, primarily implemented using ASP.NET Core 7, which can be categorized as follows: infrastructure services, web applications, and business microservices. Infrastructure services include a SQL Server (maintaining business data for microservices), a REDIS server (storing shopping cart information), and RabbitMQ (used in payment process management and order finalization). Business microservices encompass a Catalog Microservice (that allows registration of new items, price updates, and product queries), an Order Microservice (allows management of client orders), a Shopping Cart Microservice (enables shopping cart reading, adding new products, cart deletion, etc.), and an Identity Microservice (responsible for customer identification management). Additionally, we extended the application by implementing a new Discount Management Microservice to enhance the analyzed case study. The architecture of the used application is similar to the example used in Section 2.1.

4.3 Test Case

This study considers the scenario where an administrator updates the price and discount of a product simultaneously while it is in a customer's shopping cart. In this scenario, a TCC consistency violation occurs when concurrently, the administrator updates the data associated with a product, and a customer reads their shopping cart. Since a product's data update occurs in the Catalog and Discount microser-

vices, a customer is prone to reading an altered price alongside an unchanged discount if their reading occurs during the product update. μTCC resolves this inconsistency through the protocol described in Chapter 3. Specifically, a customer will never be able to read an altered price with an unchanged discount within the same functionality, and vice versa. μTCC ensures that product updates occur atomically for the customer. Therefore, we developed and tested wrappers for the Catalog, Discount, and Shopping Cart microservices. Clients were configured to generate an 80/20 read/write ratio. For experiments, we varied the number of requests per second made by clients (in increments of 40 requests per second). Tests were conducted for reads/writes in low/high contention contexts (set of 22 items and 1 item, respectively). To evaluate μTCC, we varied the number of versions maintained for each object in the storage system, ensuring that the percentage of aborts due to the lack of a consistent version always remains below a predefined value. As an example, we set this value to 4%. It is worth noticing that the abort rate is not the only criterion used to choose the most adequate number of versions per data object in remote storage, as we will see during the evaluation assessment of μTCC.

During the evaluation, our emphasis was on preventing anomalies stemming from the absence of Atomicity in functionalities spanning multiple microservices. While anomalies can arise from the lack of Atomicity, they can also emerge due to the disregard for causality. It's important to note that within the functionalities of the *eShopOnContainers* application described previously, there's never a breach of causality between operations. Therefore, the aborted functionalities are solely caused by anomalies related to the lack of Atomicity.

4.4 Prevention of TCC Anomalies

We begin by measuring the prevalence of read operation anomalies between the base system and μTCC in Figure 4.1. Every time an anomaly is detected (and the transaction is aborted), due to clients requesting causal cuts whose values have been removed by automatic memory recycling, we impose the transaction's re-execution. We observe that, on average, for a load of 520 requests per second in a highly contended scenario, the percentage of read operations where violations of the TCC model occur is 73.13%. For the same test, analyzing the data obtained with μTCC while maintaining 25 versions for each data object in remote storage, we found that only 0.97% of transactions are aborted. This discrepancy is motivated by the need for the base system to re-execute transactions more frequently in the event of clients requesting causal cuts whose values have been removed by automatic memory recycling.

To understand the impact of maintaining multiple versions of each data object in remote storage, we continue by assessing the occurrences of read anomalies captured by the TCC model during system execution with μTCC. We conduct two similar tests, changing the contention context of the data set used

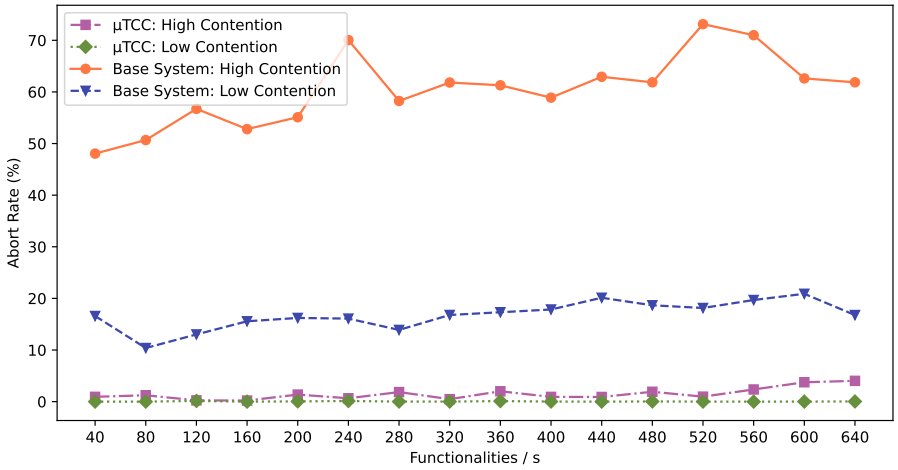

Figure 4.1: Average Abort Rate for Read Operations

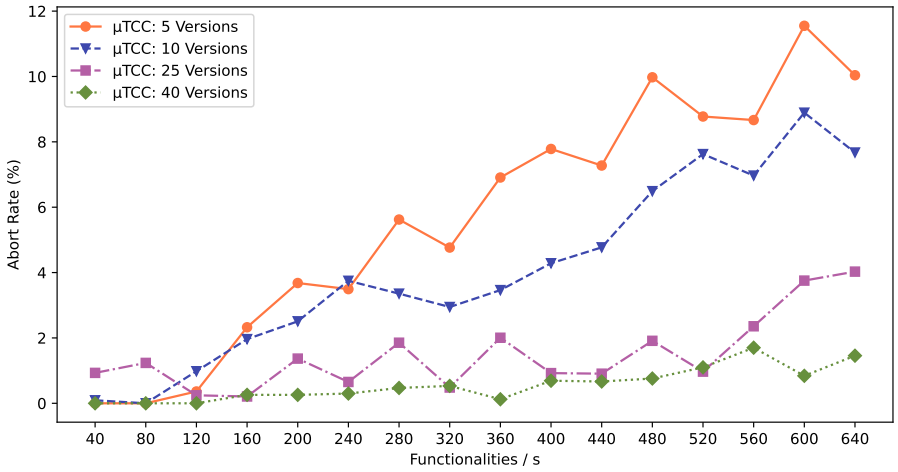

Figure 4.2: Average Abort Rate for Read Operations — Version Study under High Contention

in each scenario, as described previously.

In Figure 4.2, we illustrate the impact of increasing the number of versions maintained for each data object in a highly contended scenario. On average, for a load of 640 requests per second, the percentage of aborted read operations due to a lack of consistent version in remote storage is minimum when remote storage maintains the 40 most recent versions per data object, namely 1.45%. As we decrease the number of versions for each data object, the abort rate value increases.

4.5 Overhead introduced by μTCC

4.5.1 Latency Performance Analysis

We continue by evaluating the latency overhead introduced by μTCC, for read-only transactions, write-only transactions, and mixed operation transactions following the 80/20 read/write ratio.

4.5.1.1 Read-only Transactions: Read Shopping Cart Functionality

Figure 4.3 illustrates the latency overhead for the 95^{th} percentile of functionalities executed, in a test scenario where no TCC anomalies are present. This means that transactions only require a single round to read consistent values across the functionality. We observe that both systems present an increase in latency, as the load of the system increases, as expected. On average, for a load of 640 requests per second in a highly contended scenario, μTCC presents a latency $1.16\times$ higher than the results obtained for the same test in the base system.

Considering the low contention scenario, the results obtained by μTCC reveal a similar pattern. On average, for a load of 640 requests per second in a lowly contended scenario, μTCC presents a latency $1.26\times$ higher than the results obtained for the same test in the base system.

These results obtained in both contention contexts are explained by the fact that no data consistency anomalies captured by the TCC model are visible, and thus, no transaction requires re-execution to read consistent values. The additional latency overhead present in the μTCC is associated with both the execution of the microservice wrappers, where we consider two important aspects: the mechanisms that inject the metadata that captures the causal cut being used, and the communication with the coordinator, to where all microservices that participated in the read transaction send their tokens; and the storage wrappers, where remote storage queries are intercepted and rearranged for the fetching of a version consistent with the causal cut being used in the transaction.

To study the impact in read-only transactions of the communication between the microservice wrapper and the coordinator to where the tokens are sent, we tested an optimized version of μTCC, where read-only transactions are marked from the beginning, and thus do not require the execution of the commit phase with the coordinator. Figure 4.4 demonstrates the result improvements when comparing with the results presented earlier. On average, for a load of 640 requests per second in a highly contended scenario, μTCC presents a latency $1.09\times$ higher than the results obtained for the same test in the base system.

Considering the low contention scenario, the results obtained by μTCC reveal a similar pattern. On average, for a load of 640 requests per second in a lowly contended scenario, μTCC presents a latency $1.15\times$ higher than the results obtained for the same test in the base system.

Using the results obtained from both the non-optimized and optimized versions of μTCC, we can

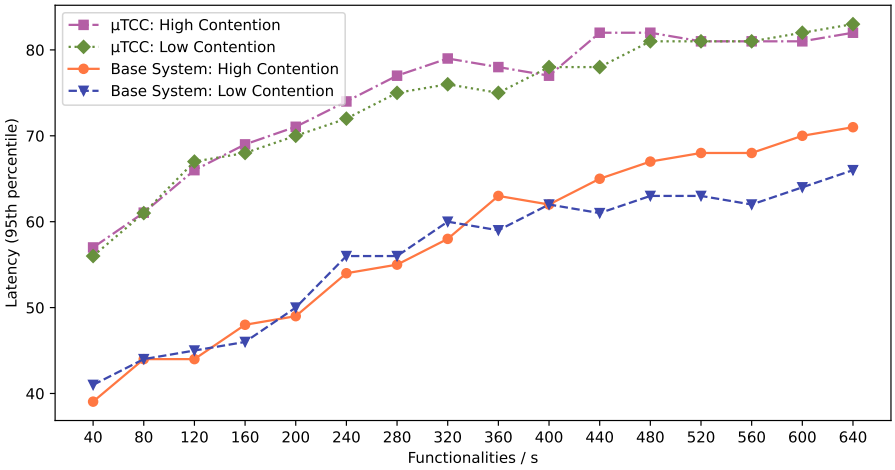

Figure 4.3: Latency of Read-only Functionalities: Read Shopping Cart functionality

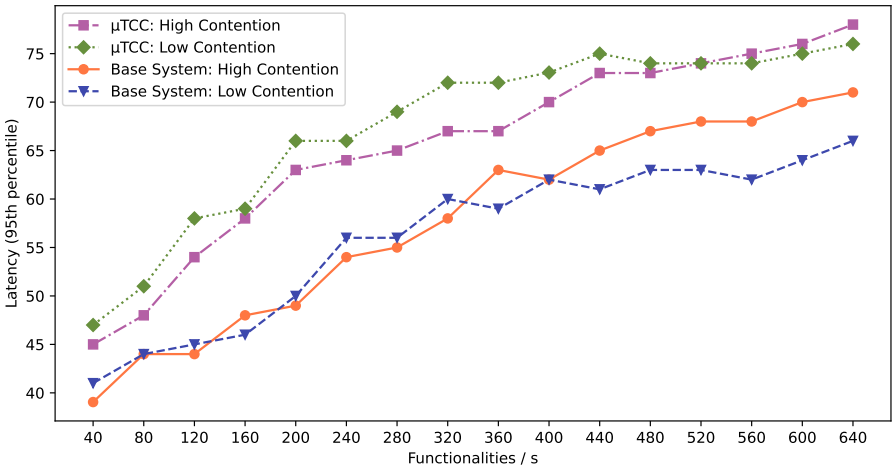

Figure 4.4: Latency of Read-only Functionalities: Read Shopping Cart functionality (Optimized)

estimate that the process of sending the commit tokens to the coordinator represents 40.0% of the additional overhead in μTCC for both the high contention and low contention scenarios, when compared with the base system. Similar optimizations to the ones proposed have been introduced in systems such as Corbett et al. [42].

4.5.1.2 Write-only Transactions: Update Price and Discount Functionality

Figure 4.5 illustrates the overhead latency for the 95^{th} percentile of write-only functionalities executed. We observe that both systems exhibit an increase in latency, as the load of the system increases, as expected. On average, for a load of 320 requests per second in a highly contended scenario, μTCC presents a latency $1.37\times$ higher than the results obtained for the same test in the base system.

Considering the low contention scenario, the results obtained by μTCC reveal a similar pattern. On

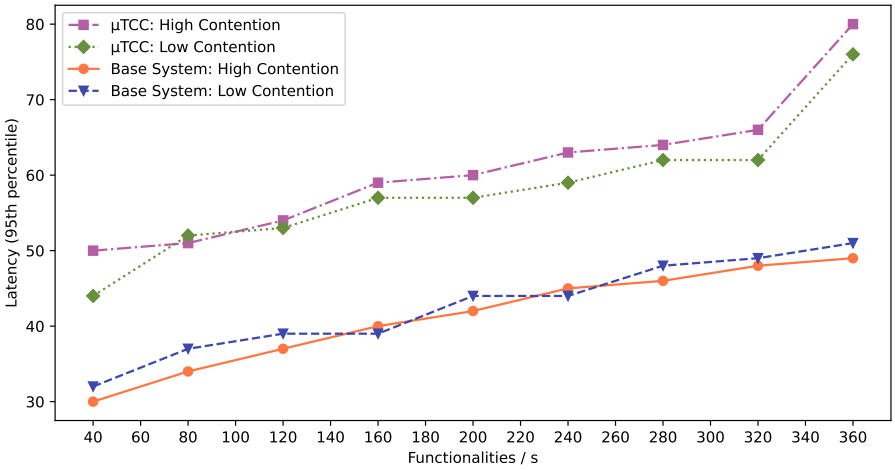

Figure 4.5: Latency of Write-only Functionalities: Update Price and Discount functionality

average, for a load of 320 requests per second in a lowly contended scenario, μTCC presents a latency $1.26\times$ higher than the results obtained for the same test in the base system.

The additional latency overhead present in the μTCC is associated with both the execution of the microservice wrappers, where we consider two important aspects: the mechanisms that inject the metadata that captures the causal cut being used, and the communication with the coordinator, regarding the commit phase; and the storage wrappers, where versions of data objects are temporarily stored until the commit order is issued for persistence.

4.5.1.3 Mixed Transactions

The mixed transactions tested in this scenario include both Read Shopping Cart and Update Price and Discount functionalities. Figure 4.6 shows the impact on latency for the 95^{th} percentile of functionalities executed until consistent data is read, for the various test cases described above, when comparing the base system, and μTCC configured to maintain the 25 most recent versions for each data object. As we can observe, both systems experience an increase in latency with the system load, as expected. As it can be seen, considering the highly contended scenario with a load of 640 requests per second, μTCC exhibits a latency $2.63\times$ lower than the results obtained with the base system. These results are supported by μTCC's ability to satisfy most read operations consistently in just one round, whereas in the base system, due to the frequency of aborted transactions, a read operation might require multiple rounds to read consistent values.

Regarding the low contention scenario, the results obtained by μTCC show a slight reduction in latency for tests with a load of 640 requests per second, specifically $1.04\times$ lower than the result obtained for the base system. This value suggests that the penalty introduced by the wrapping mechanism used in μTCC, associated with the extra effort to filter data access in order to obtain a version that is consistent

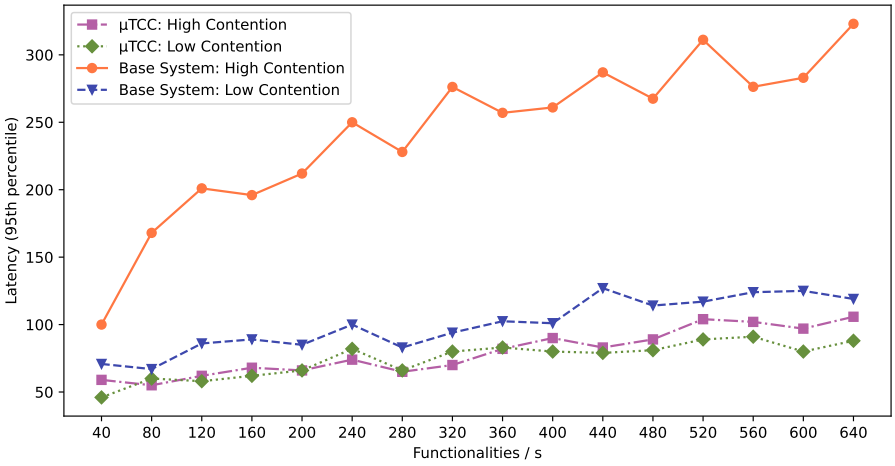

Figure 4.6: Latency (95^{th} percentile) of Mixed Operation Functionalities: Update Price and Discount + Read Shopping Cart functionalities

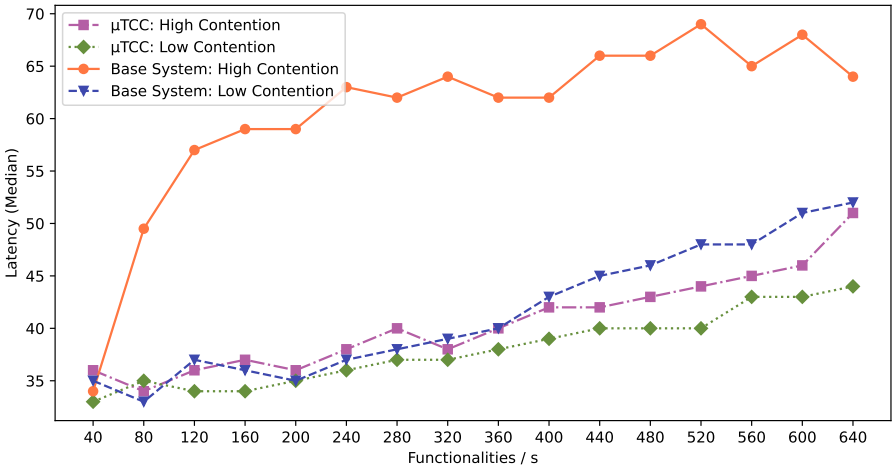

Figure 4.7: Latency (median) of Mixed Operation Functionalities: Update Price and Discount + Read Shopping Cart functionalities

with the client's state, is outweighed by the need for the base system, in turn, to perform multiple rounds to read consistent values.

Besides measuring the tail latency, in Figure 4.7 we measure the median latency for both systems under the same contention scenarios. As it is possible to observe, for a load of 600 requests per second, in high contended scenario, μTCC achieves a median latency $1.32\times$ lower than the result obtained for the base system. Regarding the low contention scenario, for the same load of 600 requests per second, μTCC achieves a median latency $1.15\times$ lower than the result obtained for the base system.

The difference between the results for the two systems mainly arises due to the versioned data storage in μTCC. While the base system maintains only one version in its storage, μTCC keeps, for each of the application, the 25 most recent versions written by the clients.

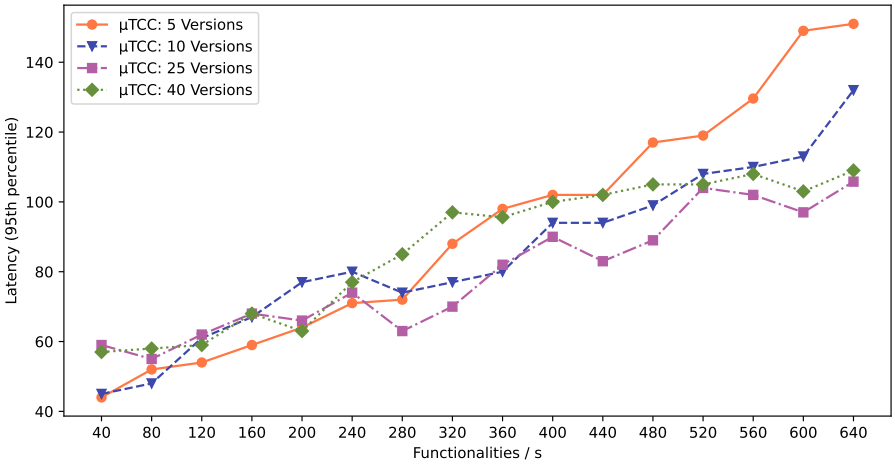

Figure 4.8: Latency of Mixed Operation Functionalities: Update Price and Discount + Read Shopping Cart functionalities over different Versions per Data Object Policies

We follow by studying the impact on latency overhead in μTCC, introduced by increasing the number of versions maintained for each data object. In Figure 4.8, we can observe the impact on latency for the 95^{th} percentile of functionalities executed until consistent data is read, in a highly contended scenario. As we can observe, for a load of 640 requests per second, μTCC, when configured to store the 25 most recent versions per data object, demonstrates the best latency performance, out of all the configurations tested, namely $1.03\times$ lower than the second best-performing configuration (μTCC when configured to maintain the 40 most recent versions per data object).

When taking in consideration the abort rate results as well as the latency overhead introduced in μTCC, it is possible to understand the tradeoff between limiting the number of versions per data object for performance gains (by reducing the dataset size in remote storage) and increasing the risk of read operation failures due to a lack of a consistent version in storage. While configurations for μTCC that store a smaller number of versions per data object typically perform better in terms of latency (while disregarding transaction re-execution), the test results show that, as a consequence of the higher abort rates, transactions require re-execution more frequently, hindering the latency performance gains mentioned earlier.

Generally, we observe that increasing the number of versions stored per data object leads to a performance decrease due to heightened complexity in the storage wrapper when fetching a specific consistent version from remote storage, performance which is then compensated by the lack of transaction re-execution in order for the client to read consistent values.

Pursuing the goal of maintaining an abort rate due to lack of a consistent version in remote storage under 4%, we proceed to evaluate the memory overhead introduced by μTCC with 25 versions maintained for each data object. This configuration of μTCC introduces the lowest latency overhead while

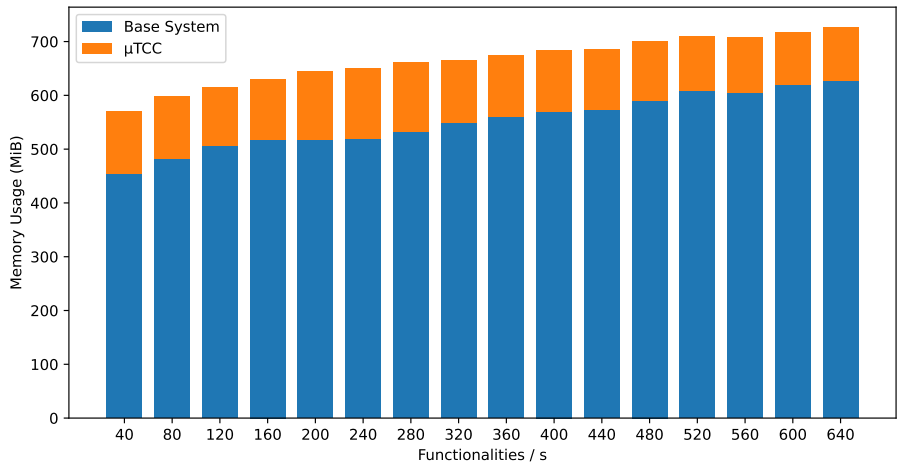

Figure 4.9: Memory Usage of Mixed Functionalities under Low Contention — Catalog Service

guaranteeing that the percentage of aborted transactions remains lower or equal than 4%.

4.5.2 Memory Performance Analysis

Figures 4.9 and 4.10 illustrate the memory usage overhead for a test scenario consisting of both Read Shopping Cart and Update Price and Discount functionalities. This test is executed for the low contention scenario. As it is possible to observe, both services experience an increase in memory usage with the system load, as expected. The memory usage difference when comparing the base system and μTCC is associated with both the microservice and storage wrappers. On average, for a load of 640 requests per second, the memory usage for the Catalog Service using μTCC is $1.15\times$ higher when compared to the base system memory usage. On the Discount Service case, for the same load test, the results show that μTCC's memory usage is $1.18\times$ higher than the results obtained for the base system.

The similar results obtained for the Catalog and Discount services are explained by the architecture and functionalities tested affecting these services. Both store unconfirmed versions of data objects locally until the coordinator confirms the transaction.

Figure 4.11 illustrates the memory usage overhead for a similar test scenario, focused on the Shopping Cart service's memory usage. Similarly to the previous results obtained for the Catalog and Discount services, it is possible to observe an increase in memory usage with the system load, as expected. The memory usage difference when comparing the base system and μTCC is associated with the microservice wrapper. The Shopping Cart service does not store data objects, neither locally nor remotely. This way, the Shopping Cart service does not require the usage of a storage wrapper. On average, for a load of 640 requests per second, the memory usage for the Shopping Cart Service using μTCC is $1.06\times$ higher when compared to the base system memory usage.

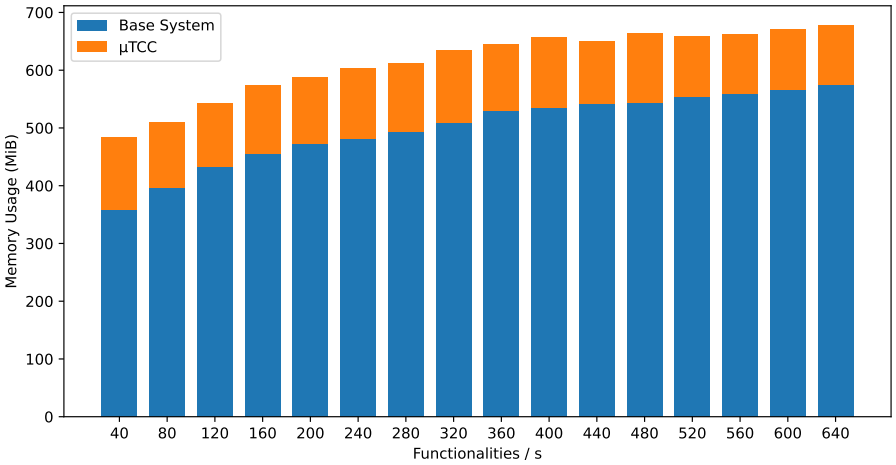

Figure 4.10: Memory Usage of Mixed Functionalities under Low Contention — Discount Service

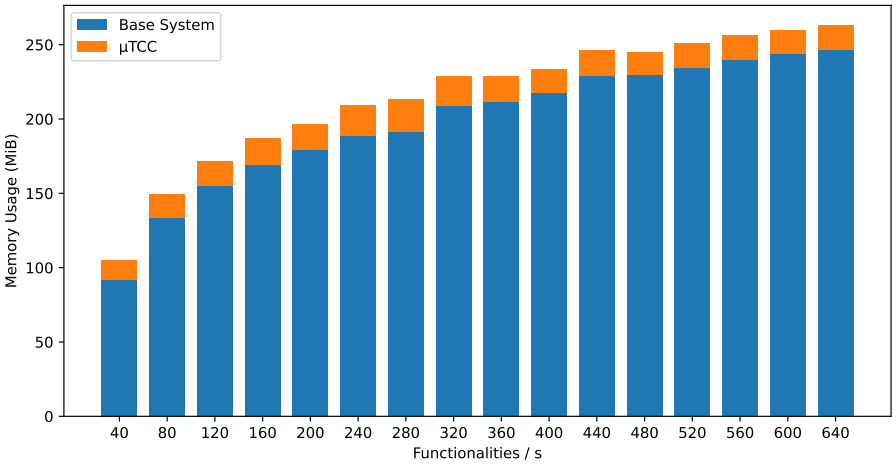

Figure 4.11: Memory Usage of Mixed Functionalities under Low Contention — Shopping Cart Service

Summary

This chapter discussed the evaluation goals set to answer. The results demonstrated that μTCC can efficiently extend TCC support for microservice architectures. It captured all TCC anomalies in read operations, without incurring in prohibitive latency and memory overheads. Our experiments revealed that, despite the introduced overhead latency in each individual operation, μTCC is able to compensate for this by reducing the overall transaction latency by $2.63\times$. This improvement stems from μTCC's ability to execute functionalities in a single round, effectively reducing transaction abortion probabilities and subsequent re-executions.

5

Conclusion

Contents

5.1 Conclusions

In this thesis, we addressed the problem of offering Transactional Causal Consistency (TCC) to microservice applications. In particular, we studied mechanisms that both ensure the atomicity of the results of functionalities that span multiple microservices and ensuring that functionalities always read versions of data objects that are mutually consistent. We have designed and evaluated a mediation layer, that we have named μTCC, that is capable of providing these guarantees. This layer uses wrappers that encapsulate microservices and storage systems, allowing for the seamless provision of the desired consistency guarantees in the implementation of microservices. The results show that μTCC can prevent the occurrence of TCC anomalies, eliminating the need to execute compensating actions, while ensuring both high-availability, low latency, and low memory overhead.

5.2 Future Work

Our prototype uses a very simple strategy to eliminate obsolete versions of data objects. Namely, a garbage collection thread is run periodically to purge old versions of each data object. It would be interesting to explore more sophisticated strategies, that could exploit idle periods to perform garbage collection.

The current version of μTCC uses physical clock values to totally order update transactions. When evaluating μTCC, we have considered a scenario where all microservices execute in a single datacenter, that have their clocks synchronized with a negligible skew. It would be interesting to extend the evaluation to geo-replicated scenarios, where the clock synchronization skew can be larger. In particular, it would be interesting to assess how likely it is that a read operation is temporarily blocked due to the clock skew (this may occur when the clock of a given service is in the past of the read snapshot).

Finally, many microservice architectures use a combination of shared memory and event based communication to coordinate multiple services. The problem of defining a suitable consistency model that integrates both shared memory and event based communication only recently started to be investigated [45]. Augmenting μTCC with support for novel consistency criteria, that can take into account the use of event-based platforms, such as publish-subscribe systems, is an interesting avenue of research.

Bibliography

[1] T. Mauro. (2015, February) Accessed: 28/12/2022. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[2] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, "Microservices in practice: A survey study," in *VI Workshop on Software Visualization, Evolution and Maintenance*, 2018, pp. 75 –82.

[3] J. F. Almeida and A. R. Silva, "Monolith Migration Complexity Tuning Through the Application of Microservices Patterns," in *Software Architecture*. Cham: Springer International Publishing, 2020, pp. 39 –54.

[4] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '87. New York, NY, USA: Association for Computing Machinery, December 1987, p. 249 –259. [Online]. Available: https://doi.org/10.1145/38713.38742

[5] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[6] M. Fowler. (2014, March) Accessed: 14/12/2022. [Online]. Available: https://martinfowler.com/articles/microservices.html

[7] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116 –116, 2015.

[8] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583 –590.

[9] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195 –216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[10] N. Parikh. (2020, April) Accessed: 27/09/2023. [Online]. Available: https://engineering.linkedin.com/blog/2020/continuous-integration

[11] A. Gluck. (2020, July) Accessed: 28/09/2023. [Online]. Available: https://www.uber.com/en-PT/blog/microservice-architecture/

[12] T. Stuart. (2016, August) Accessed: 27/09/2023. [Online]. Available: https://developers.soundcloud.com/blog/microservices-and-the-monolith

[13] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 8, pp. 18 –22, January 2018.

[14] R. Meshenberg and J. Evans. (2015, October) Netflix at aws re:invent 2015. Accessed: 22/12/2022. [Online]. Available: https://netflixtechblog.com/netflix-at-aws-re-invent-2015-2bc50551dead

[15] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[16] V. Talwar. (2016, August) grpc: a true internet-scale rpc framework is now 1.0 and ready for production deployments. Accessed: 30/09/2023. [Online]. Available: https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments

[17] I. Saleem, P. Nargund, and P. Buonora. (2021, July) Data caching across microservices in a serverless architecture. Accessed: 19/12/2022. [Online]. Available: https://aws.amazon.com/blogs/architecture/data-caching-across-microservices-in-a-serverless-architecture/

[18] M. Fowler. (2011, November) Accessed: 01/31/2023. [Online]. Available: https://martinfowler.com/bliki/PolyglotPersistence.html

[19] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," *Proc. VLDB Endow.*, vol. 14, no. 13, p. 3348 –3361, September 2021. [Online]. Available: https://doi.org/10.14778/3484224.3484232

[20] (n.d.). Consistency models. Accessed: 14/12/2022. [Online]. Available: https://jepsen.io/consistency

[21] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, June 2016. [Online]. Available: https://doi.org/10.1145/2926965

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558 –565, July 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[23] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: definitions, implementation, and programming,” *Distributed Computing*, vol. 9, pp. 37 –49, March 1995. [Online]. Available: https://doi.org/10.1007/BF01784241

[24] P. Mahajan, L. Alvisi, M. Dahlin *et al.*, “Consistency, availability, and convergence,” *University of Texas at Austin Tech Report*, vol. 11, p. 158, 2011.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “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, pp. 401 –416.

[26] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” vol. 7, no. 3, p. 181 –192, November 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[27] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, “Cure: Strong semantics meets high availability and low latency,” in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, June 2016, pp. 405 –414.

[28] T. Little. (2022, October) Ensuring data consistency in microservice based applications. Accessed: 18/12/2022. [Online]. Available: https://blogs.oracle.com/database/post/ensuring-data-consistency-in-microservice-based-applications

[29] T. Eldeeb, X. Xie, P. A. Bernstein, A. Cidon, and J. Yang, “Chardonnay: Fast and general datacenter transactions for on-disk databases,” in *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. Boston, MA: USENIX Association, July 2023, pp. 343 –360. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/eldeeb

[30] M. Herlihy, *Linearizability*. Boston, MA: Springer US, 2008, pp. 450 –453. [Online]. Available: https://doi.org/10.1007/978-0-387-30162-4_203

[31] M. Štefanko, O. Chaloupka, and B. Rossi, “The saga pattern in a reactive microservices environment,” in *Proceedings of the 14th International Conference on Software Technologies*, ser. ICSOFT 2019, SciTePress Prague, Czech Republic. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, January 2019, pp. 483 –490. [Online]. Available: https://doi.org/10.5220/0007918704830490

[32] (n.d.). Saga distributed transactions pattern. Accessed: 04/10/2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

[33] M. Letia, N. Preguiça, and M. Shapiro, “Crdts: Consistency without concurrency control,” *arXiv preprint arXiv:0907.0929*, 2009.

[34] “The network time protocol,” 2023. [Online]. Available: http://www.ntp.org

[35] X. Shi, S. Pruett, K. Doherty, J. Han, D. Petrov, J. Carrig, J. Hugg, and N. Bronson, “{FlightTracker}: Consistency across {Read-Optimized} online stores at facebook,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, November 2020, pp. 407 –423. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/shi

[36] J. Du, S. Elnikety, and W. Zwaenepoel, “Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks,” in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*. Los Alamitos, CA, USA: IEEE Computer Society, October 2013, pp. 173 –184. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SRDS.2013.26

[37] C. Wu, V. Sreekanti, and J. M. Hellerstein, “Transactional causal consistency for serverless computing,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, May 2020, p. 83 –97. [Online]. Available: https://doi.org/10.1145/3318464.3389710

[38] T. Lykhenko, R. Soares, and L. Rodrigues, “Faastcc: Efficient transactional causal consistency for serverless computing,” in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware ’21. New York, NY, USA: Association for Computing Machinery, December 2021, p. 159 –171. [Online]. Available: https://doi.org/10.1145/3464298.3493392

[39] J. a. Ferreira Loff, D. Porto, J. a. Garcia, J. Mace, and R. Rodrigues, “Antipode: Enforcing cross-service causal consistency in distributed applications,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 298 –313. [Online]. Available: https://doi.org/10.1145/3600006.3613176

[40] M. S. Ferreira, J. a. F. Loff, and J. a. Garcia, “Rendezvous: Where serverless functions find consistency,” in *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, ser. WORDS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 51 –57. [Online]. Available: https://doi.org/10.1145/3605181.3626290

[41] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan, “Enhancing saga pattern for distributed transactions within a microservices architecture,” *Applied Sciences*, vol. 12, June 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/12/6242

[42] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura,

D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, August 2013. [Online]. Available: https://doi.org/10.1145/2491245

[43] cloc - count lines of code. Accessed: 20/10/2023. [Online]. Available: https://github.com/AlDanial/cloc

[44] *eShopOnContainers*: .NET microservices sample reference application. Accessed: 05/06/2023. [Online]. Available: https://github.com/dotnet-architecture/eShopOnContainers

[45] B. Martin, L. Prosperi, and M. Shapiro, “Transactional-turn causal consistency,” in *Euro-Par 2023: Parallel Processing*. Cham: Springer Nature Switzerland, 2023, pp. 578 –591.