Transactional Causal Consistency For Microservices Architectures

(extended abstract of the MSc dissertation)

João Queirós
joao.miguel.queiros@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa
Advisor: Professor Luís Rodrigues

ABSTRACT

The microservices' architecture is a software engineering approach that structures an application as a set of loosely coupled services. Each microservice manages a small, cohesive, subset of the domain entities and can be implemented, deployed, and managed independently of other microservices. 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×, thanks to being able to reduce read transaction abortion frequency and respective re-executions.

KEYWORDS

Microservices, Transactional Causal Consistency, Consistency, Isolation

1 INTRODUCTION

The Microservices Architecture is a software engineering approach that structures an application as a set of loosely coupled services. Each microservice manages a small, cohesive, subset of 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 Transactional Causal Consistency (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. In this work, we experimentally show that the proposed solution can efficiently extend TCC support for microservice architectures, by capturing all TCC anomalies in read operations, without incurring in prohibitive latency and memory overheads. The results also reveal 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×, a feat supported by μTCC 's ability to execute functionalities in a single round, effectively reducing transaction abortion probabilities and subsequent re-executions.

2 RELATED WORK

2.1 Transactional Support in Cloud Computing

ClockSI [5] is a system that allows for the execution of transactions with snapshot isolation in a partitioned data center (where different items can be maintained by different servers). One of the key aspects of this system is allowing a client to decide which consistent snapshot it wants to read from without explicitly coordinating with a centralized entity or multiple partitions to determine the most recent versions. This is possible because the system relies on synchronized clocks to define transaction commit times, enabling the client to read from a single synchronized clock to define the capture to read. The read protocol includes mechanisms to handle potential clock desynchronization. In particular, if a node needs to process a read with a timestamp higher than its own clock, the node postpones this operation until the local clock catches up in the future.

Cure [6] was the first system to offer Transactional Causal Consistency across partitioned and georeplicated data centers. The system assumes that each client interacts with only one data center. However, this is not enough to ensure consistency because updates made by a single transaction in different partitions can become visible at different times, both in the data center where the transaction was executed and in remote data centers. Furthermore, updates received from remote data centers can arrive at a data center in an order that violates causality. Cure proposes a set of coordination mechanisms, during data writes and reads, that ensure clients always observe consistent states.

In summary, Cure uses the following mechanisms to guarantee consistency: all writes of a given transaction are marked with the same timestamp, higher than any timestamp assigned to client transactions that confirmed in the past. These writes are applied using a two-phase commit protocol that ensures all writes executed during the transaction are atomically persisted in the database. The consistent state observed by a transaction is characterized by a vector clock V, with an entry for each data center i, where $V[i]$ indicates the last confirmed transaction in i that is visible to the client. The vector clocks are used by each data center to locally apply transactions initiated in remote data centers in an order that respects causality. To read consistent data in the local data center, the corresponding entry for the local data center in the vector clock is replaced by the synchronized physical clock of a given partition, reading the most recent values using the ClockSI protocol.

FlightTracker [7] is a system that ensures clients always observe their own writes across a series of interactions with the system, a property known as Read-Your-Writes (RYW), despite the components being replicated across multiple geographical regions and replicas only ensuring eventual consistency. In a system with these characteristics, in the absence of additional coordination mechanisms, a client can write to one replica and later try to read from another replica where its own write has not yet been applied. To provide RYW guarantees, FlightTracker requires clients to carry a ticket that includes metadata about the writes they have made in the past.

Each ticket includes a collection of records, composed of a union of representations per database, one for each component where a write was performed. Each record is opaque, and its internal structure is known and interpretable only by the component that generated it. By separating the internal structure and the interfaces exposed in the ticket, FlightTracker ensures flexibility, compatibility, and the possibility of optimizing the internal structure of the ticket, transparently to the components that use it. FlightTracker simply collects and shares these records during a sequence of interactions between a client and the system. If a client contacts a replica that does not yet have the updates indicated in the record, the system opts for one of three hypotheses: forward the request to another replica in the same region, where the desired version might already be visible; delay the request, wait for the asynchronous propagation of the client's write to become locally visible (an effective option but potentially deteriorating to the system's performance); or forward the read request to another region.

This system demonstrates that it is feasible to maintain metadata to offer stronger consistency guarantees, even in large-scale systems.

2.2 Transactional Support in FaaS Systems

FaaSTCC [8] is a system that provides Transactional Causal Consistency guarantees for clients running applications (modeled as a directed acyclic graph of functions) in a computing system that follows the Function-as-a-Service (FaaS) paradigm. Specifically, FaaSTCC ensures that all functions executed within a given graph

observe a consistent cut of the system, even when they run on multiple nodes, each with its own storage system cache. Unlike Cure, the client can execute functions that access different replicas of data (in this case, caches), and the number of replicas is extremely dynamic due to elastic resizing policies of the number of executing nodes. Similar to the Cure system, all writes of a given transaction are timestamped with a common logical clock, higher than any previous writes serialized in the past.

However, unlike Cure, in FaaSTCC, the consistent cut is not defined at the beginning of the transaction, nor does it rely on the use of vector clocks. FaaSTCC associates a consistent interval with each transaction, capturing a range of logical clock values within which the transaction can read with consistency guarantees. This interval is adjusted as the transaction reads objects, based on the versions present in the caches of each node involved in the transaction. FaaSTCC also uses a coordination protocol between the storage service and caches, allowing a node to verify if a given version of an object, confirmed at time t, is still valid at a given time t', where $t' > t$.

2.3 Transactional Support in Microservices

As mentioned, many microservice-based systems use the Saga pattern, which does not guarantee isolation between functionalities, allowing various consistency anomalies to occur. The *"Enhancing Sagas"* system [9] proposes an extension to this pattern to avoid Partial Reads and Dirty Reads. For this purpose, the writes performed by the various transactions that implement a functionality are stored in temporary memory, not visible to other concurrent executions, until the functionality completes and is confirmed. The system uses a coordinator responsible for verifying that all microservices involved in the execution of a functionality have successfully completed before persisting the writes stored in temporary memory to the storage system.

2.4 Discussion

The Enhancing Sagas system prevents the reading of updates made by functionalities that abort. However, this system does not guarantee that functionalities always read mutually consistent values. To achieve this goal, it is necessary to ensure that the data accessed by each microservice is consistent (intra-service consistency) and that the data read by different microservices in the context of the same functionality is also consistent (inter-service consistency). The challenge of maintaining intra-service consistency, when microservices do not replicate each other's data, is quite similar to the problem addressed by ClockSI. If microservices receive updates from data maintained by other microservices and keep some of these updates in cache, the mechanisms proposed by Cure and FaaSTCC are also relevant. In either case, these solutions require clients executing a functionality to carry metadata capturing the consistent cut in which they operate to address the challenge of inter-service consistency; experience with FlightTracker shows that this is feasible, even in large-scale systems with high-performance requirements, such as Facebook.

3 μTCC

When developing μTCC , our focus was on creating a transparent solution for the application, minimizing the adaptation of existing systems for its adoption. To achieve this, μTCC was designed as a set of wrappers that intercept requests between microservices and requests from microservices to their respective storage systems, adding metadata capable of maintaining a consistent causal cut during the execution of a transaction. Specifically, as illustrated in Figure 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.

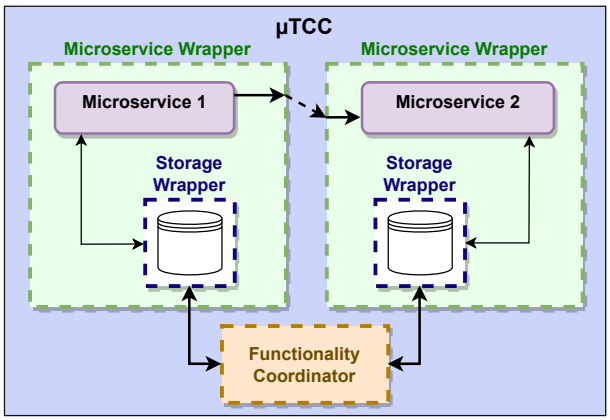

Figure 1: μTCC System Architecture

3.1 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 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, the storage wrapper associates a timestamp for each committed transaction. The value of this timestamp 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 examined in Section 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, 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 our prototype, we implemented one storage wrapper for the Microsoft SQL Server storage system, a relational database management system that uses SQL to construct the relational schemas for each database in our system.

3.2 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 determine 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.

3.3 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.

In orchestration mode, all microservice invocations are made by the coordinator itself, so there is no need to rely on a token to detect the completion of a functionality.

In choreography mode, the execution of a microservice can trigger multiple other microservices. In this case, tokens are used and 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.

In both scenarios, if all microservices have completed their execution and are in a position to confirm the functionality, 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.

3.4 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 additions 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.

3.5 Protocols

We proceed by detailing μTCC 's protocols. The algorithms presented include the pseudocode describing the protocols that are executed in the Microservice and Storage wrappers.

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 assumes 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, 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.

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.

3.5.1 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.

3.5.2 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 [5], 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               */
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)
```

3.5.3 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.

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
```

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 [5, 10]. 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).

3.6 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.

3.7 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.

We quantify 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.

4 EVALUATION

In this section, we evaluate the performance of μTCC . We assess μTCC in terms of anomaly prevention and performance by integrating it into a reference application developed by Microsoft, specifically the eShopOnContainers application [11]. We structure our evaluation around two key questions: first, how prevalent are TCC consistency violations, and how effectively does μTCC prevent them? Second, what is the impact on performance when introducing the use of μTCC in the base application?

4.1 Experimental Workbench

The eShopOnContainers 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), a REDIS server (storing shopping carts), and RabbitMQ (used in payment process management). Business microservices encompass a Catalog Microservice (that allows registration of new items, price updates, and product queries), an Order Microservice, a Shopping Cart Microservice (enables shopping cart reading, adding new products, etc.), and an Identity Microservice (customer identification management). Additionally, we extended the application by implementing a new Discount Management Microservice to enhance the analyzed case study.

4.2 Test Case

In the evaluation process, we focused on 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 microservices, a customer is prone to reading an altered price alongside an unchanged discount if their reading occurs during the product update. μTCC guarantees that 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 .

4.3 Prevention of TCC Anomalies

We begin by measuring the prevalence of read operation anomalies between the base system and μTCC in Figure 2. 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

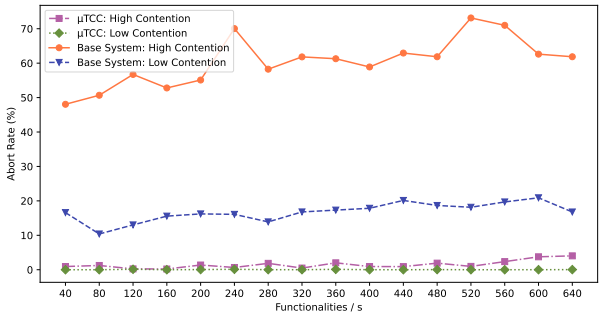

Figure 2: Average Abort Rate for Read Operations

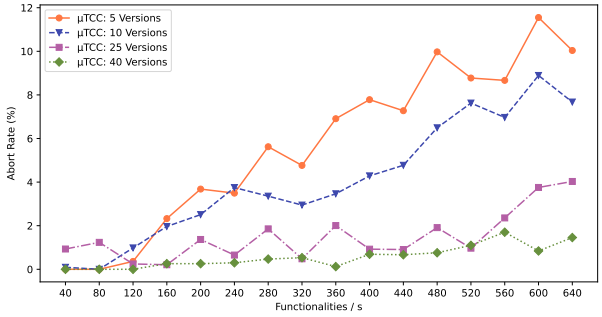

Figure 3: Average Abort Rate for Read Operations — Version Study under High Contention

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 in each scenario, as described previously.

In Figure 3, 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.

4.4 Latency overhead introduced by μTCC

4.4.1 Read-only Transactions: Read Shopping Cart Functionality. Figure 4 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

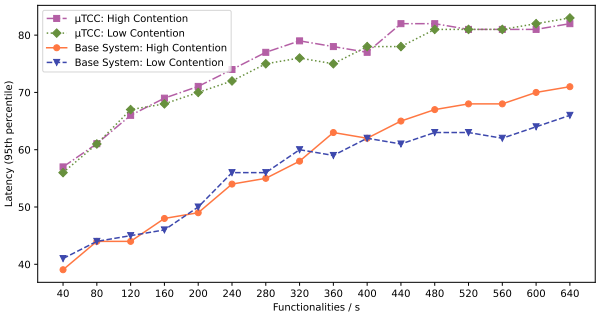

Figure 4: Latency of Read-only Functionalities: Read Shopping Cart functionality

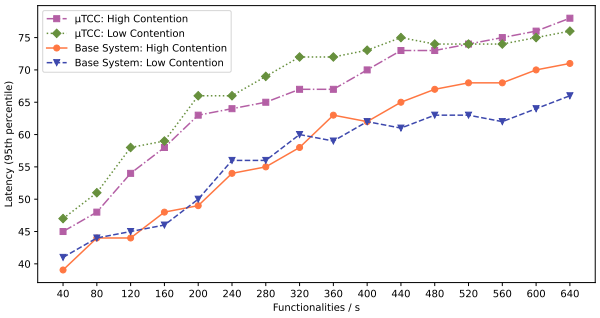

Figure 5: Latency of Read-only Functionalities: Read Shopping Cart functionality (Optimized)

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× 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× 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

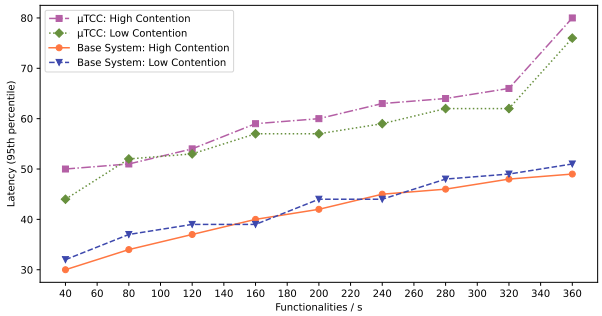

Figure 6: Latency of Write-only Functionalities: Update Price and Discount functionality

thus do not require the execution of the commit phase with the coordinator. Figure 5 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× 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× 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 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. [10].

4.4.2 Write-only Transactions: Update Price and Discount Functionality. Figure 4 illustrates the overhead latency for the 95^{th} percentile of 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× 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 320 requests per second in a lowly contended scenario, μTCC presents a latency 1.26× 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.

4.4.3 Mixed Transactions. The mixed transactions tested in this scenario include both Read Shopping Cart and Update Price and Discount functionalities. Figure 7 shows the impact on latency for

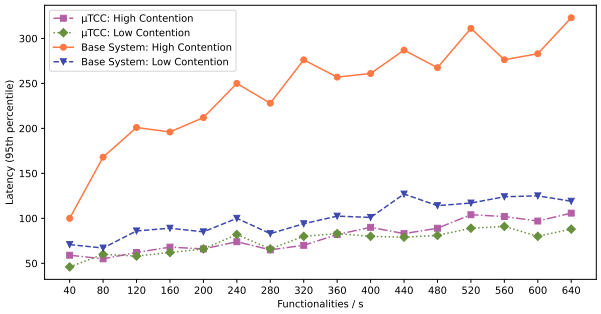

Figure 7: Latency of Mixed Operation Functionalities

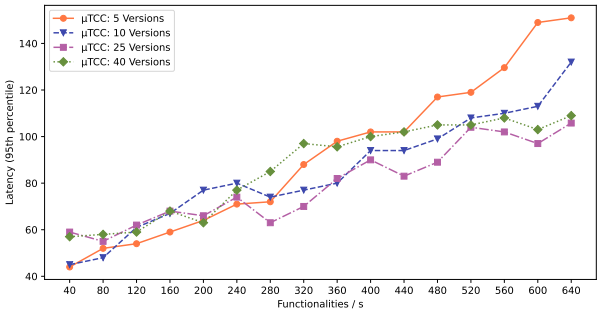

Figure 8: Latency of Mixed Operation Functionalities over different Versions per Data Object Policies

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

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.

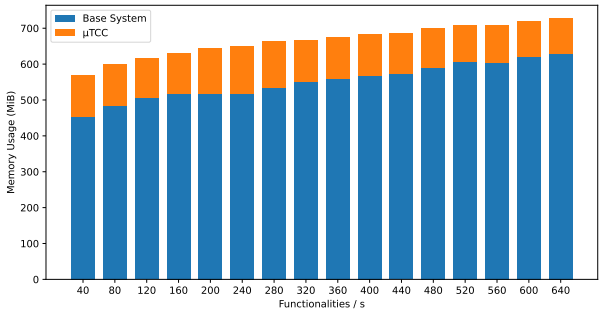

Figure 9: Memory Usage of Mixed Functionalities under Low Contention — Catalog Service

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

4.5 Memory overhead introduced by μTCC

Figure 9 illustrates 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, the service experiences an

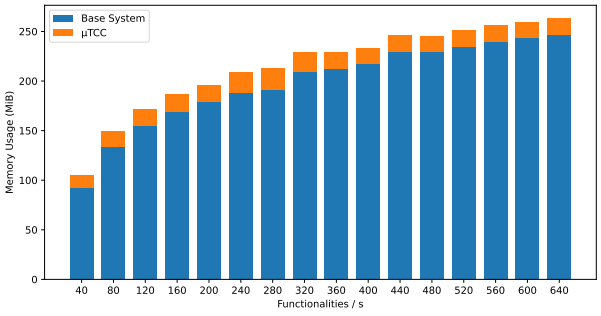

Figure 10: Memory Usage of Mixed Functionalities under Low Contention — Shopping Cart Service

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× higher when compared to the base system memory usage.

Figure 10 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× higher when compared to the base system memory usage.

5 CONCLUSIONS AND FUTURE WORK

In this thesis, we addressed the problem of offering 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.

6 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 [12]. 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.

ACKNOWLEDGMENTS

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).

REFERENCES

[1] T. Mauro, February 2015. Accessed: 28/12/2022.
[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, pages 75 –82, 2018.
[3] J. F. Almeida and A. R. Silva. Monolith Migration Complexity Tuning Through the Application of Microservices Patterns. In Software Architecture, pages 39 –54, Cham, 2020. Springer International Publishing.
[4] Hector Garcia-Molina and Kenneth Salem. Sagas. In Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, SIGMOD '87, page 249 –259, New York, NY, USA, December 1987. Association for Computing Machinery.
[5] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In 2013 IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS), pages 173 –184, Los Alamitos, CA, USA, October 2013. IEEE Computer Society.
[6] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pages 405 –414, June 2016.
[7] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. FlightTracker: Consistency across read-optimized online stores at Facebook. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 407 –423, November 2020.
[8] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. Faastcc: Efficient transactional causal consistency for serverless computing. In Proceedings of the 22nd International Middleware Conference, Middleware '21, page 159 –171, New York, NY, USA, December 2021. Association for Computing Machinery.
[9] Eman Daraghmi, Cheng-Pu Zhang, and Shyan-Ming Yuan. Enhancing saga pattern for distributed transactions within a microservices architecture. Applied Sciences, 12, June 2022.
[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. ACM Trans. Comput. Syst., 31(3), August 2013.
[11] *eShopOnContainers*: .NET microservices sample reference application. Accessed: 05/06/2023.
[12] Benoît Martin, Laurent Prosperi, and Marc Shapiro. Transactional-turn causal consistency. In Euro-Par 2023: Parallel Processing, pages 578 –591, Cham, 2023. Springer Nature Switzerland.