Microservice Decomposition for Transactional Causal Consistent Platforms

Madalena Santos
madalenacsantos@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Today, there are many software applications that have been designed using monolithic configurations that could benefit from being decomposed into a combination of microservices or, in some cases, stateless functions. However, when decomposing a monolithic application in microservices, the programmer needs to write additional code to correct the anomalies that may be generated when executing the composition in a decentralized system. Tools that support the decomposition of monolithic applications into microservices automatically compute a number of complexity metrics, providing an estimate for the amount of effort required to code the compensating actions for a given decomposition. This information guides the programmer in finding the most suitable decomposition. A limitation of these tools is that they have been developed under the assumption that the execution environment is unable to offer any type of support for transactions. We aim at extending these tools with mechanisms that can consider the different consistency models supported at runtime, in particular, Transactional Causal Consistency. For this purpose we will use automated procedures to identify potential anomalies generated during the execution of a given decomposition under the TCC model. The identification of these anomalies can be used to guide the development of compensating actions, and offer a principled method to estimate the complexity associated to the deployment of a given decomposition.

Table of Contents

1 Introduction

The microservice architectural style has been widely adopted for the past years. In opposition to monolithic architectures, microservice architectures decompose an application into a set of small and well-contained services, each having its own cohesive set of responsibilities. This modularization of the system function offers many benefits: services are smaller and less complex, and hence easier to implement, modify and test, and each service can be independently deployed using the technology and hardware resources that are more appropriate to its nature. Regarding system performance, microservices allow for higher availability and fault isolation: a fault in one of the services will not bring the whole system down, as is the case with monoliths, where one misbehaving component could compromise the operation of the entire application. Furthermore, each service can also be subject to independent horizontal scaling according to its type of demand.

Implementing a microservice architecture can bring many advantages, but can also impose additional complexity during the development: distributed computing is complex, and adds intricacy to application development, testing and deployment. Services need to be able to handle faulty behaviour and unavailability of other services, and dependencies between them need to be taken into account. Also, to ensure a high decoupling between the different microservices, these are usually deployed on infrastructure that has no support for distributed transactions. Instead, most microservices and FaaS architectures rely on weakly consistent storage services. This means that a modular decomposition of the monolithic application is exposed to intermediate states and to inconsistent data versions that may cause the occurrence of anomalies. To mitigate the impact of these anomalies, the programmer must develop additional code, for instance, compensating actions, that can correct the effects of unintended behaviours generated during the execution. Monolithic versions of the same application are not exposed to these anomalies, considering they usually rely on a transactional substrate that can offer strong consistency, such as Serializability, typically offered by a single datastore that relies on ACID properties ($\underline{\text{A}}$tomicity, $\underline{\text{C}}$onsistency, $\underline{\text{I}}$solation and $\underline{\text{D}}$urability).

Given the tension between the benefits that come from modularity and the additional complexity that results from the lack of isolation, the task of finding the best decomposition for an otherwise monolithic application, i.e., the task of defining the boundaries of each service and the redesign of the system's functionalities to accommodate the partition according to the consistency policy required is not trivial. To ease this task, a number of tools to support the decomposition of monolithic applications to microservices automatically compute a number of complexity metrics, that provide an indicative estimate for the amount of effort required to code the compensating actions that can correct latent anomalies during the execution of a given composition [1, 2]. A limitation of these tools is that they have been developed under the assumption that the execution environment is unable to offer any type of support for transactions.

This project is based on the insight that there are a number of transactional consistency models that have been developed for geo-replicated systems and have enormous potential to simplify the programming of applications that use microservice and FaaS architectures. Most notably, we are interested in materialising the concept of Transactional Causal Consistency (TCC) [3, 4] in this context. It is known today that TCC is the strongest semantics that can be implemented using non-blocking algorithms and without requiring the execution of consensus among participants in a transaction [5–7]. TCC ensures that clients observe a sequence of write operations that respects causality and, furthermore, guarantees that the results of a transaction are atomically visible. This prevents a number of anomalies that can be hard or even impossible to compensate when using the Saga pattern. In virtue of these advantages, the use of TCC has been broadly advocated for several settings, including FaaS architectures [8].

We aim at extending previous tools to support the decomposition of monolithic applications into a set of microservices with mechanisms that can take into account the different consistency models supported at runtime, in particular, TCC. For this purpose, we will use automated mechanisms to identify anomalies that can arise during the execution of a given decomposition under the TCC model. In particular, we plan to leverage on existing tools, such as CLOTHO [9], a framework that detects serializability violations of Java applications executing on top of weakly consistent distributed databases. CLOTHO employs a static analyzer and a model checker to generate abstract executions of the input program, discover serializability violations in these executions and translate them back into concrete test inputs that can then be used for assessment by application developers. The identification of these anomalies can be used to guide the development of compensating actions, and offer a principled method to estimate the complexity associated to the deployment of a given decomposition.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Sections 3, 4, 5 and 6 we present all the background related with our work. Section 7 describes the proposed architecture to be implemented and Section 8 describes how we plan to evaluate our results. Finally, Section 9 presents the schedule of future work and Section 10 concludes the report.

2 Goals

This work addresses the problem of estimating the complexity of decomposing a monolithic system into microservices for execution environments that support Transactional Causal Consistency.

> *Goals:* We aim at designing a tool that takes as input a monolithic application, generates a set of microservice decompositions for the application, and outputs information that can help in assessing the complexity of implementing these decompositions on top of TCC, including details of the anomalies that can occur during the execution and hints regarding the compensating actions required to address these anomalies.

To achieve this goal we plan to extend previous tools, namely the work of Santos and Rito Silva [2], that breaks a monolithic application in different sets of microservices, and CLOTHO [9], a tool that automatically detects the anomalies that can occur during the execution of a distributed application.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a prototype of the proposed tool; ii) an experimental evaluation using a number of monolithic applications; iii) a comparison of the complexity metric produced by our tool and the complexity metrics proposed in [2].

3 Background

This section aims to introduce concepts that will be relevant to the understanding of the document. We start by briefly discussing the advantages and disadvantages of microservices architectures versus monolithic architectures. We then address the challenges of providing strong consistency in microservices architectures. Finally, we review commonly used strategies to mitigate the negative effects of weak consistency in microservice architectures.

3.1 Monolithic versus Microservice Architectures

In a monolithic architecture, all functionalities of an application are executed by a single machine or server that implements all the application logic. Furthermore, the application state is typically stored in a single database. This setup makes it straightforward to execute functionalities in the context of transactions, safeguarding isolation between concurrent executions of the same or different functionalities [10].

In a microservice architecture, different functionalities can be executed by different machines, each making use of an independent storage system. Functionalities that are executed uniquely inside a single microservice can be executed employing some transactional substrate, but those that are executed over multiple microservices cannot be guaranteed to yield high isolation semantics [10], as will be discussed below.

Both architectural patterns have advantages and disadvantages.

Monoliths, on the one hand, present limitations in performance due to the large shared data domain that is accessed by all functionalities of the system. This provokes a major setback in availability and fault-tolerance, and thus instigates the need for a novel, distributed architectural pattern that addresses these concerns. On the other hand, when using monoliths, the programmer can leverage transactions to avoid reasoning about concurrency.

In turn, microservice architectures have the opposite pros and cons. For one thing, the modularity provided by this paradigm allows the allocation of different developer teams, programming languages, and data storage technologies to each service. Also, individual services execute in individual processes or machines, which provides the additional benefit of lowering the probability of full-scale failure when a set of the services is anomalous. For another, distributed systems are harder to program, and those who take up this pattern have to tackle the overhead caused by remote communication and global synchronization of data. Maintaining strong data consistency is immensely challenging and the tendency of faults is significantly larger. To design performant and correct microservices, architects and programmers need to consider all the consequences of failure for every remote execution, as well as those deriving from the difficulty of synchronizing distributed objects.

3.2 Serializability

A *transaction* is an abstraction that allows the programmer to group a sequence of operations on multiple objects of a data store such that they are executed as an atomic unit. Transactions can either commit or rollback: if a transaction commits, all its effects are permanent and visible to other transactions; if it cannot commit, the transaction will be rolled-back, reversing all operations that it consists of and leaving the database unchanged. Furthermore, the execution of a transaction is isolated from the concurrent execution of other transactions, relieving the programmer from explicitly implementing concurrency control. The properties of transactions are also known as the ACID properties [11]: <u>A</u>tomicity, stating that all changes to data are performed as if they were a single operation and either all changes happen or none do; <u>C</u>onsistency, which requires that the transactions always leave the database in consistent states that respect business rules; <u>I</u>solation, specifying that intermediate states of a transaction should not be seen by other transactions; <u>D</u>urability, implying that the changes to data are to be definitive after the transaction is committed, and cannot be undone even in the case of system failures.

Transactional systems have been widely studied in the literature, namely - but not exclusively -, by the database community. Different consistency criteria that characterize precisely how transactions are isolated from each other have been proposed. The strongest consistency model for transactional systems is *serializability*, stating that a concurrent execution of a set of transactions should be equivalent to some serial execution of these transactions. Serializability is intuitive for programmers and designers: if an application is correct in serial executions, it will remain correct in concurrent executions.

However, enforcing serializability is expensive, because automated techniques to enforce concurrency control introduce inefficiencies in the system operation. In distributed systems, enforcing serializability requires ordering the transactions in a total order and coordination, typically in the form of a two-phase commit protocol [12]. For these reasons, the consistency models implemented by modern datastores are often weaker than serializability. Bailis *et al.* [5] conducted a survey where 18 off-the-shelf popular database systems were analyzed, and only 3 of those provided serializability as the default consistency model. Perhaps surprisingly, 8 of the systems considered in their evaluation did not provide serializability at all.

3.3 Consistency in Microservice Architectures

Microservice architectures are deployed on distributed systems and therefore inherit the advantages and challenges associated with distribution. On the one hand, as discussed before, microservice architectures can be made more fault-tolerant and more scalable than centralized systems. On the other hand, implementing coordination among multiple services is costly and may impair system availability. The trade-off between availability and consistency in distributed systems is captured by the CAP Theorem [13], stating that any given distributed system can deliver only two of the following three desired characteristics: consistency, availability, and partition tolerance. Intuitively, this limitation results from the fact that nodes may not be able to coordinate when there is a partition in the network. Therefore, in the presence of a partition, one must choose between consistency and availability.

Most microservice architectures favour availability and, therefore, avoid depending on distributed transactions that span multiple services. Instead, transactions can be used internally by each individual microservice, such that functionalities that are executed by the same microservice are isolated from each other, but functionalities that are executed by multiple microservices are assumed to execute without any form of concurrency control. This implies that end-users will be exposed to intermediate states in the functionality execution graph that would not occur in a monolithic system. Furthermore, intermediate states of different functionalities can interact with one another, which adds to the number of inconsistent states that the business logic of one functionality needs to consider.

3.4 Anomalies

Given that most microservice architectures favour availability and scalability, avoiding the costs of ensuring strong consistency, programmers need to deal explicitly with the anomalies that may be generated when executing different microservices concurrently, without isolation. In this section we enumerate the main anomalies that can occur for which the programmer needs to write compensating actions.

Lost Update A Lost Update occurs when one transaction T1 reads some data object and another transaction T2 updates the same object. Then, T1 modifies that object based on its local value for the object, never observing T2's more recent write.

Read Skew The Read Skew anomaly describes the case where, inside the same transaction, two read operations to the same object return different values.

Write Skew The Write Skew anomaly can be described as a generalization of Lost Update to multiple data objects. It occurs when transaction T1 reads object x and writes to object y, and transaction T2 reads object y and writes to object x. None of the transactions is able to observe the other's updates.

Dirty Read This anomaly occurs whenever one transaction is able to read a value that is being updated on a different transaction that has not yet commited.

Dirty Write Berenson *et al.* [14] define a Dirty Write anomaly by drawing on the following example: some transaction T1 updates one data object and, before it has the opportunity to commit or rollback, transaction T2 initiates and updates the same data object. If either transaction T1 or T2 were to rollback, it is unclear what the value for the data object should be.

3.5 Managing Weak Consistency

The Saga [15] pattern is one of the main alternatives to the use of distributed transactions in the context of the microservice architecture. Each transaction that spans multiple services is a *saga*. A saga is a sequence of local transactions, where each local transaction updates data within a single service according to the ACID properties, and then triggers the next local transaction in the saga. If some local transaction fails because it somehow violates a business rule, then the saga must execute a series of compensating transactions that will rollback the changes made by the local transactions that precede the one that failed.

Sagas can be coordinated in two manners: a *choreography* is a saga where each local transaction publishes an event upon completion that will trigger the next local transaction in a different service; an *orchestration* is a saga that is coordinated by an orchestrator object or class that is responsible for telling the participants of the saga when to execute the corresponding local transactions.

Sagas cannot be automatically rolled back as is the case with traditional ACID transactions. Each step of the saga commits its changes locally, thus if one of the steps fails, the effects of each of the previous transactions must be undone through the use of *compensating transactions*. The saga executes these compensating transactions in reverse order of the forward transactions.

Considering the execution of a single saga, there are three types of local transactions:

Pivot Transaction The pivot transaction divides the execution of the saga. It is considered to be the go/no-go point at which the success of the saga is determined. If this transaction commits, then the saga will run until completion. An example for this would be the last local transaction that verifies the conditions for a money transfer to be possible, such as the confirmation that the account has more money than the value to be withdrawn.

Compensatable Transactions In the case that the pivot transaction fails, all transactions that have executed before it must be compensated for. These are the compensatable transactions. There must be a compensating transaction for all compensatable transactions that write or update data.

Retriable Transactions Retriable transactions are the ones that follow the pivot transaction, and thus are guaranteed to succeed. There is no need to write compensating transactions for these.

Sagas differ from ACID transactions in that they lack the isolation property [10], which ensures that the result of executing a set of concurrent transactions is the same as if that set of transactions was executed sequentially. In sagas, updates made by each local transaction are immediately visible to other sagas as soon as the transaction commits, meaning that it is possible that sagas change data accessed by another saga while the latter is executing, and that sagas can read the updates of others before they have completed, which can lead to inconsistent states and anomalies. One way to deal with the lack of isolation in sagas is through the use of countermeasures such as Semantic Locks, Commutative Updates, Pessimistic View, Reread Value or Version File [10].

4 Weak Consistency Models

As we have discussed above, monoliths are commonly built on top of transactional systems that offer serializability, while microservices are generally built assuming no consistency guarantees across different services. However, there are a number of weaker forms of consistency that can be enforced without incurring the costs associated with serializabilty, namely without compromising availability. There is potential in simplifying the implementation of applications using the microservice architecture if some of these models are supported by the execution environment. In this section, we survey some of the weak consistency models that have been proposed in the literature.

4.1 Session Guarantees

A *session* is an abstraction that captures a sequence of read and write operations executed by a program. *Session guarantees* are properties that are ensured by the system on individual operations of a session (in opposition to properties enforced on groups of operations, as in transactional systems). Session guarantees are assured even if the program interacts with different servers during the session, and have been defined to simplify the design of distributed applications. Terry *et al.* [16] define four different session guarantees: Monotonic Reads, Monotonic Writes, Writes Follow Reads and Read Your Writes:

Monotonic Reads (MR) within a session, repeated reads to a data object never return older versions than the last observed version.

Monotonic Writes (MW) writes become visible to other participants in the order that they were submitted by the originating session.

Writes Follow Reads (WFR) if a session performs a write operation W and afterwards performs another write operation W', then any other sessions that can observe the effect of W' will also be able to observe the effects of W, since W *happens before* W'.

Read Your Writes (RYW) whenever a session reads a data object after updating it, the read value will always yield the updated value, or a value that overwrote that update.

Causal Consistency (CC) The combination of all of the four specified guarantees originates Causal Consistency [17, 18], which has been proven to be the strongest guarantee compatible with high availability [5,19]. This guarantee captures the notion that causally-related operations should appear in the same order to all sites in a system. If an update is visible at some site, then all the updates that it is dependant on should also be visible at that site. Because causally-consistent memory does not require the establishment of a total order of events, it allows for scalable, partition tolerant and available implementations, and thus is widely used in practice.

Causal+ Consistency (Causal+) Causal+ is an extension of CC. In addition to guaranteeing causality, Causal+ further ensures that copies of the same data objects will eventually converge to the same value, by forcing that, when concurrent updates are seen, one of the updates is applied last at all sites.

4.2 Highly Available Transactions

Session guarantees, as previously stated, are defined on individual operations, and do not apply to groups of operations, i.e., to transactions. It is also possible, in a transactional context, to define consistency criteria that are weaker than serializability and do not compromise availabililty; these criteria are known to provide *Highly Available Transactions* [5]. The semantics vary in their level of transaction isolation, and those compatible with high availability are described in the following lines.

Read Uncommited (RU) A total order for all writes is established, and updates should be applied at each service according to that ordering. This isolation level prevents the Dirty Writes anomaly.

Read Commited (RC) Isolation level RC prohibits the Dirty Writes and Dirty Reads anomalies by buffering new updates either on the client or the server side, until the data is able to be commited. This ensures that transactions will never read intermediate versions of data.

Monotonic Atomic View (MAV) MAV provides a higher level of atomicity: it ensures that if some effects of a certain transaction are observed by other transactions, then all its effects should also be seen by those transactions, guaranteeing an "all or nothing" visibility of transactions. MAV prevents Dirty Reads and Dirty Writes, and is considered to be relatively stronger than RC, but slightly weaker than TCC.

Cut Isolation (CI) Cut Isolation states that each transaction reads from a non-changing cut or snapshot over the data items. There are two isolation levels that stem from this concept: Item Cut Isolation (I-CI), where this property holds over reads from discrete data items, and Predicate Cut Isolation (P-CI) where the cut is maintained over predicate-based reads (e.g. SELECT WHERE). To implement this, transactions store a copy of all data read, and subsequent reads are loaded from this local copy, returning the same value observed before. This value only changes if the transaction overwrites it itself. Cut Isolation prevents Read Skew but allows Dirty Writes and Dirty Reads.

Transactional Causal Consistency (TCC) Causally consistent memory, as previously discussed, is defined for single operations on single data objects. This allows for certain abnormal behaviors to arise. Consider the following example, based on a social network scenario, where people can create profiles for themselves and define reciprocal friendship relations, meaning that the profiles that maintain this relationship can mutually access content published on the other's profiles. Profile A and profile B are friends with each other. Suppose that at a certain point in time, profile A decides that it does not want profile B to access profile A's content anymore and it removes the friendship relationship with B. After this, profile A publishes a post, assuming that profile B will not have access to it. If this application is based on the aforementioned CI isolation level, the program will mistakenly allow profile B to see the new post of profile A, because the cached value for the friendship will not have been updated.

This anomaly happens because operations do not respect causality: profile A's friendship removal *happens-before* its new post, and thus, read operations that observe the writing of the post, should also observe the update on the friendship state. Transactional Causal Consistency (TCC) is an extension of Causal+, where the definition of Causal Consistency is lifted to the level of transactions, guaranteeing the consistency of reads and writes for a set of keys by demanding that all operations are applied on top of the same causal snapshot. In this specific case, it would impose that the new post from profile A is observed together with the relationship removal, which in turn would block profile B from seeing the new post.

TCC ensures atomic visibility of written keys - either all writes from a transaction are seen or none are - by using non-blocking algorithms that employ control information stored together with the data, in order to verify if transactions are reading from a causal cut or not. However, this semantics does not require the operations to be totally ordered - as is the case with Snapshot Isolation and Serializability - meaning that it can be achieved without the use of expensive consensus protocols and thus providing the strongest semantics attainable with high availability [20].

5 Decomposing Monolithic Applications into Microservices Compositions

In this section, we briefly introduce two tools that have been designed with the goal of supporting the decomposition of monolithic applications into microservices.

5.1 A Complexity Metric for Microservices Migration

In [2], Santos and Rito Silva propose a tool to estimate the cost of migrating a monolith to a microservice architecture, and mechanisms to generate several different decompositions based on a proposed set of criteria. The tool works by collecting data from the source code of the monolithic system using static analysis. More precisely, the tool assembles the read and write operations made to the system's domain entities and the sequence of those accesses done by each functionality. This information is used to derive metrics of correlation between domain entities. Intuitively, two entities are correlated if they are accessed together by one or more functionalities. The work is based on the premise that one should favor decompositions where the entities that are more frequently accessed together should be clustered in the same service, to reduce the amount of synchronization needed between clusters. Entity correlation is measured by four *similarity measures* that are described below:

- **Access**: considers the number of functionalities that access two entities, for each pair of domain entities in the system
- **Read**: this measure is an instance of the *Access* measure, where accesses made are reads, by counting the number of functionalities that read two given entities, for each pair of domain entities in the system;
- **Write**: this measure is an instance of the *Access* measure, where accesses made are writes, by counting the number of functionalities that write two given entities, for each pair of domain entities in the system;
- **Sequence**: considers the number of cases where the two domain entities appear in consecutive positions in the sequence of accesses of the functionalities, for each pair of domain entities in the system.

The values for the similarity between entities capture how coupled they are, and this information is fed to a clustering algorithm that will generate new candidate decompositions for the monolith.

To evaluate the candidate microservice configurations, the authors propose complexity metrics that estimate the development effort needed to migrate the original system into each of the decompositions. These metrics are related to the number of accesses made by distinct microservices to correlated entities. The rationale for this is that, as we have discussed earlier, when entities are accessed by functionalities implemented by the same microservices, the accesses can be performed in a transactional context, but when they are made by functionalities in different microservices, the accesses cannot be protected by a transaction and will expose anomalies that need to be compensated for, generating complexity in development.

The authors of this work compute the value for the complexity of one candidate decomposition in the following manner:

Complexity of decomposition d: The complexity of a decomposition is the average of the complexities of all the functionalities in *d*.

Complexity of a functionality f in a decomposition d: The complexity of *f* in *d* is the sum of the complexities of accessing the clusters in the sequence of accesses of *f*.

Complexity of accessing cluster c on the sequence of accesses of a functionality f: The complexity of accessing *c* is the sum of complexities of the accesses made by *f* to the entities in *c*.

Complexity of accessing entity e in cluster c by functionality f: The complexity of accessing an entity depends on the type of operation being made: if entity *e* is being read by *f*, the complexity of the access is related to the number of other functionalities that write to *e*. If entity *e* is being written to by *f*, the complexity of the access is related to the number of other functionalities that read *e*.

The value for the complexity of a given decomposition helps architects and system designers in choosing the most valuable one in the set of generated decompositions, taking into account the available resources (e.g. number of developers and time) to carry out the process of partitioning a monolith.

Their work also makes a valuable contribution to the problem of deciding the boundaries and responsibilities of each service when decomposing a monolith. The clustering algorithm used by the authors takes as input the values of the similarity measures for each pair of entities in the system, but the four similarity measures can have different weights in the generation of the decomposition. For each input monolith, different combinations for the valuation of each similarity measure are created, and for each combination, a decomposition is generated. After calculating the complexity for each generated decomposition, the authors reckoned that there is no single combination for the weights of the similarity measures that can be universally applied to all monoliths and originate the decomposition with the lowest complexity.

5.2 Monolith Migration Complexity Tuning Through the Application of Microservices Patterns

In [6], Almeida and Rito Silva extend the work above, and propose a refinement to the complexity metric of [2] by splitting it into two new ones: the complexity introduced specifically by the redesign of each of the monolith's functionalities to accommodate the decomposition into microservices and the complexity added to the system when obliged to deal with inconsistent views introduced by the distributed nature of the new version of the system.

The authors further explore the effort in decomposing a monolith by introducing a representation scheme for reasoning about microservice functionality: a *functionality execution graph.* In this graph, the nodes are the local transactions that execute inside a single service, and the edges are the remote invocations between those local transactions.

A distinct contribution of this work was the creation of a set of operations to be performed over the initial functionality execution graph. The purpose of these operations is to redesign the execution flow of the application's functionalities before applying the Saga pattern to the monolith decomposition, avoiding some compensating actions, for instance by merging local transactions and, in this way, avoiding some intermediate state to become visible. To apply these operations, one needs to study the source code of the application and determine which parts of the code should be separated into different functionalities and, for each functionality, the possible points-of-failure. Examples of this would be exception-throwing fragments in the code, which are parts of the execution flow where, if there is a failure, the ACID transaction in the monolith will abort. However, in the corresponding microservice decomposition, such exceptions correspond to a local transaction in a single service that has failed and compensating transactions will have to be triggered. The proposed operations are described below:

Sequence Change Given a functionality and its functionality execution graph, we will consider three distinct local transactions (nodes) lt_1, lt_2 and lt_3, the remote invocation (edge) $ri = (lt_1, lt_3)$ and the additional information that lt_3 executes after lt_2. However, in the case that it is required that the microservice's functionalities execute as a Saga orchestration, it is useful to have one of the local transactions trigger all the others. In this example, since lt_2 executes before lt_3, that is, it does not depend on data generated by lt_3, it would be possible to replace ri by $ri' = (lt_2, lt_3)$, if lt_2 was to be the orchestrator node of the Saga.

Local Transaction Merge This operation is useful for, when during the redesign process two different local transactions in the same service become adjacent in the functionality execution graph, and thus, can be merged into a single local transaction, which can aid in reducing the number of intermediate states.

Add Compensating This operation is used to add a new local transaction and a remote invocation to connect the new node to the already existing functionality execution graph. The new node represents the compensating transaction that deals with one of the previously-mentioned *compensatable transactions* in a Saga.

6 Verifying Serializability of Applications to Calculate Complexity

While the two previously-mentioned works provide essential insights on how to determine boundaries between microservices and contribute with valuable complexity metrics, they only reason about static relations between the entities of the system when calculating complexity, not taking into account the specific parameters given as inputs to the programs on each individual execution. This does not allow for a precise identification of the interactions that may generate concurrency problems, generating a large number of false positives when counting potential sources of anomalies in applications.

In order to solve this, our work tends towards a more dynamic approach for the computation of complexity. We assessed a number of works that provide mechanisms or tools to determine consistency anomalies in the form of serializability violations of programs.

6.1 Robustness Against Transactional Causal Consistency

The work of Beillahi *et al.* [21] investigates the relationships between different variations of Causal Consistency, and provides theoretical proofs for mechanisms that automatically verify the serializability of a transactional program executing on top of a causally consistent database. Their main effort is towards investigating the decidability for the problem of checking robustness of programs. A program executing on top of a weaker semantics is said to be *robust* against serializability if the effects of executing that program while enforcing serial behaviors are equivalent to the effects of executing the same program relying instead on the original weaker semantics.

The authors consider three different variations of Causal Consistency: weak causal consistency (CC), causal memory (CM) and causal convergence (CCv). CC has been discussed in section 4. CCv differs from CC because the former enforces a total order between all transactions that defines the order in which delivered concurrent transactions are executed at every site, guaranteeing that all sites reach the same state after delivering all transactions. CM differs from CC because it assures that all values read by a site can be explained by an interleaving of the transactions consistent with the causal order. CC is strictly weaker than both CM and CCv.

The notion of robustness presented in this work relies on an interpretation of program behaviors as *traces* that document causal dependencies between transactions, which allows for a more precise identification of serializability violations than other state-based approaches. Their advances are purely theoretical, opening the door to the use of existing tools and frameworks to check robustness of applications.

6.2 Decidable Verification under a Causally Consistent Shared Memory

Lahav and Boker [22], similarly to [21], make efforts towards establishing the decidability for the problem of verifying safety properties - serializability - of finite-state transactional programs executing on top of Causal Convergence (CCv), which is also given the name of Strong-Release-Acquire (SRA). The authors deduce that reasoning about the problem of safety verification under SRA is equivalent to reasoning about the problem of *SRA reachability*: considering the execution graph of a program P executing on top of SRA, a state p of P is reachable under SRA if some execution of P that satisfies the conditions of SRA generates state p. Despite providing theoretical grounds for the implementation of novel frameworks and tools that check serializability of programs, this work does not propose one.

6.3 Static Serializability Analysis for Causal Consistency (C^4)

C^4 [23] is an end-to-end static analysis framework for client-applications of causally consistent databases. The authors propose a novel serializability criterion for local evaluation and combine the already existing graph-based techniques with the encoding of the new criterion into first-order logic formulae. This framework is independent of the datastore API or programming language and thus can be used with any system that satisfies convergence, atomic visibility and causal consistency.

C^4 starts its analysis by inferring the *abstract history* of the program. An *abstract history* of a program is a generalization of all possible ways in which said program can interact with the datastore. The graphic representation of an abstract history is a *Static Serialization Graph* (SSG), which is derived from the abstraction of all possible concrete *Dependency Serialization Graphs* (DSG). DSGs are graphs representing concrete executions of a program, where there is a node for each executed transaction and an edge between each two nodes depicting session order and dependencies. If the SSG for a particular program is acyclic, then it can be deduced that said program is serializable. Despite being fast and efficient, SSG-based analysis does not capture specific semantics of the exact objects being manipulated because it generalizes all existing dependencies for the set of possible executions of the program. In an individual execution, the dependencies may not exist, and thus this technique generates a considerable number of false positives. To overcome this, the authors propose a complementary procedure to vouch for the results given by the SSG-based analysis. It consists in the encoding of the input program's SSG to logical formulae to be checked by SMT solvers. This allows to precisely reflect control-flow between operations to eliminate infeasible cycles in the abstract history. The SMT-based analysis is applied whenever the SSG-based analysis indicates a potential serializability violation, and produces a counter-example for each proven anomaly.

For a given program executing on top of a causally consistent distributed database, C^4 either proves that the program is serializable, or detects a non-serializable behavior. If the program is not serializable, the tool outputs the set of violations found for up to two sessions, and determines if this result is generalizable to an arbitrary number of sessions.

6.4 Automated Detection of Serializability Violations under Weak Consistency (ANODE)

Nagar and Jagannathan [24] propose a fully automated approach for finding serializability violations under any weak consistency model. The framework takes as input a program written in a simplified version of the SQL language, which is described in detail in their report.

Their main effort is to determine the conditions under which said transactional program can be statically identified to always yield a serializable execution without the need for global synchronization. The ANODE framework can be used with any weak consistency model whose specification can be expressed in first-order logic.

From the input program, a dependency graph with a cycle is construed. The framework then tries to discover a valid execution of the input program under the given consistency specification that can result in such graph. The authors propose two different approaches for verifying serializability: the Shortest Path approach and the Inductive approach. Both are employed, and the anomalies found are output to the user together with the transactions involved and their parameters.

Since this framework is parametric over the consistency specification, it can be used to determine the weakest consistency policy for which the program is serializable, or simply modify the transactions where anomalies are present.

6.5 Directed Test Generation for Weakly Consistent Database Systems (CLOTHO)

CLOTHO [9], an improvement of the tool of name ANODE discussed in Section 6.4, is a framework that detects serializability violations of Java applications that make use of weakly consistent distributed databases. It employs a static analyzer and a model checker to generate abstract executions of the input program, discover serializability violations in these executions and translate them back into concrete test inputs that can then be used for assessment by application developers.

More specifically, CLOTHO takes as input a Java class that manipulates a database through a JDBC API where each method is treated as a transaction, and outputs a set of satisfying assignments to the parameters of the input application that cause serializability anomalies.

CLOTHO generates a precise encoding of database applications, which allows it to accurately represent the complex dependency relations between SQL

	Trace/State	Original Memory Model Assumed	Detects Violations of	SMT-based analysis	Filtering Methods	Output	Available
Decidable Verification under a Causally Consistent Shared Memory	State	CCv/SRA	Serializability	No	No	-	No
Robustness Against Transactional Causal Consistency	Trace	CC, CCv or CM	Serializability	No	No	-	No
C4	State	CC	Serializability	Yes	Yes	The set of violations and whether this result is generalizable to an arbitrary number of sessions	Yes
ANODE	State	Any (parameter of the tool)	Serializability	Yes	No	The serializability anomalies and the transactions involved and their parameters	No
CLOTHO	State	Any (the tool determines the underlying consistency model)	Serializability	Yes	Yes	Concrete tests to replay the discovered anomalies (database file and annotated Java class files)	Yes

Table 1: Comparison of different approaches for detecting anomalies in transactional programs

select and update operations. As in many other works, the authors reason over *abstract executions* of input applications. An abstract execution of a program is a generalization of its execution that captures visibility and ordering relations among read and write operations on the database. Potential serializability violations in an abstract execution manifest as cycles in a dependency graph that represents said visibility and ordering relations. When encountering such violations, CLOTHO synthesizes concrete tests that can be used to drive executions of the program that will exhibit its points of failure. The abstract representation of database programs used by CLOTHO is automatically generated from the input program's Java source code. It is then passed to an encoding engine that constructs first-order logic formulae that captures the conditions under which a dependency cycle forms. A theorem prover is then used to compute the generated SAT representation of the problem. All satisfying solutions given by the solver are converted to test configuration files that contain the collected abstract anomalies. Such files provide details about concrete executions that can potentially manifest the discovered anomalies. This work stands out from others for the fact that it offers a test-and-reply environment that allows mapping anomalies identified in the abstract executions to be translated to concrete inputs that can be executed subsequently.

6.6 Comparison

We now provide a brief comparison of the systems surveyed in the previous paragraphs. Table 1 summarizes the key aspects of the different studied tools.

The work of Beillahi *et al.* [21] offers a trace-based approach for detecting serializability of applications, which is more precise than the other state-based approaches, that require the set of reachable states under serializability to be equal to the set of reachable states under a weaker consistency model. State-based approaches are more prone to false positives for the violations in robustness. However, the authors of this work only provide the theoretical proof for the decidability of this problem and do not implement any tools that we can make use of. Similarly, neither [21] or [22] implement tools or frameworks that we can use in our project.

The ANODE [24] and CLOTHO [9] projects have produced tools that we can use. Since ANODE is a predecessor of CLOTHO, we have decided to adopt the latter, as it includes a number of improvements over ANODE: while ANODE receives as an input the underlying consistency model of the program to be tested, CLOTHO discovers these semantics automatically by capturing the various visibility and ordering relations between reads and writes. This allows users to strengthen these characteristics of input programs as needed, while still being able to use the testing framework to discover new serializability violations. Only the code for CLOTHO was available at the time of this project.

To compare CLOTHO with C^4, we consider the inputs and outputs of both frameworks: while CLOTHO can be configured to address datastores with different consistency guarantees, C^4 assumes that the input program executes on top of causally consistent databases. In the context of our project, we will be assessing the impact of executing microservice compositions on top of storage layers that provide either Transactional Causal Consistency or Eventual Consistency. Modifying C^4 to deal with Eventually Consistent datastores may be challenging. Furthermore, CLOTHO also provides a testing environment to reproduce the discovered serializability anomalies, which can be given as an additional output to the developers of microservices.

7 Architecture

We aim at designing a tool that aids in the decision of migrating a monolith to a microservice architecture. The output for this tool is a set of candidate microservice decompositions, along with a meaningful estimate value for the cost of development of the decompositions on top of TCC, which includes detailed reports for the consistency anomalies that are bound to happen and what should be done in order to compensate these anomalies.

To do so, we intend to employ the efforts done by [2] and [6] considering monolith decomposition. The generated configurations are then tested with the aid of CLOTHO [9], a framework that will detect violations of serializability for each decomposition previously generated. The output of CLOTHO translates the discovered anomalies into concrete tests that can be executed in order to replicate the points of failure of the program, which provides architects and system designers with meaningful hints on how to solve the reported anomalies.

Since the complexity of development is inherently related to the lack of memory consistency, and therefore, to the occurrence of failures, detecting serializability violations for the execution of a set of microservices provides a purposeful metric for estimating the effort in developing the microservices. Each serializability anomaly represents one or more compensating actions that will have to be developed upon the decomposition of the monolith.

The first step towards accomplishing our goals will be to compare the complexity metric generated by [2] and [6] to the one generated by CLOTHO [9], in order to further understand if the latter can actually provide information that is more interesting to the process of decomposing a monolith. Our intuition is that, since our metric will be based on actual anomalies that occur given the specific parameters of transactions and not only on the static dependencies between them, it will provide more accurate estimates.

After this, we intend to modify the encoding made by CLOTHO of the underlying consistency model for the input program, so that we can compare complexity levels for programs that execute on top of Eventual Consistency versus the values for those that use Transactional Causal Consistency instead. Comparing these two complexity values will allow us to understand if it is worth providing stronger semantics to a microservice architectural system and if the level of consistency impacts the effort needed to decompose a monolith.

To experiment with the system, we will need to identify a set of monolithic systems that can be tested using both frameworks.

8 Evaluation

In order to evaluate our efforts, we will assess, for some fixed level of decomposition complexity, if an application providing TCC guarantees can be partitioned into more services - resulting in a higher level of decentralization - when compared to an application that does not provide any consistency guarantees.

The inverse evaluation can also be done: for a fixed level of monolith decomposition (e.g., number of clusters generated by the decomposition), can systems providing TCC guarantees be produced with lower levels of complexity than systems with no consistency guarantees?

The level of complexity represents the effort needed to decompose the initial monolith, and, on a more practical level, can be thought of as hours of work that developers need to receive payment for. Accordingly, if the owner of an application is willing to pay only a very low price to support the decomposition (low complexity level), our intuition is that the attainable decomposition level will also be low. This is, the resulting system will still be highly centralized. However, we expect that systems guaranteeing TCC will be able to generate more decentralized decompositions than systems with no guarantees, for a fixed level of complexity.

9 Scheduling of Future Work

Future work is scheduled as follows:

- May 21 - August 31: Detailed design and implementation of the proposed architecture, including preliminary tests.
- September 1 - September 30: Perform the complete experimental evaluation of the results.

– October 1 - October 31: Write a paper describing the project.
– November 1 - December 31: Finish the writing of the dissertation.

10 Conclusions

As computer systems mature and expand, the requirements for fault-tolerance and availability increase. This explains the emergence of a novel design model, the microservice architectural pattern. Our work discusses the main barriers to the application of this pattern, while aiming to provide a testing framework that aids in the decomposition of an otherwise monolithic application into a set of microservices.

The process of decomposing a monolith into different microservices is not straightforward: besides deciding on the boundaries and responsibilities of each service, developer and designer teams need to sketch compensating actions and structures to deal with the relaxation in data consistency across the new microservice architecture.

In this report, we surveyed the theoretical grounds necessary for reasoning about microservices: we discuss the advantages and disadvantages of this pattern as opposed to a monolithic one; we present serializability as a correctness criterion in a transactional context; we examine several weak consistency models and the anomalies they introduce; we study state-of-the-art works that propose techniques for decomposing monoliths and discover serializability anomalies in concurrent executions.

Our work leverages on the previously-developed methods for decomposing monoliths and discovering anomalies in the form of serializability violations. We use the set of discovered anomalies in a decomposition to predict the effort needed to implement the microservice partition. We believe that this metric will guide architects, developers and designers of computer systems in the decomposition of monoliths, by providing accurate recommendations of points-of-failure for applications.

Acknowledgments We are grateful to Taras Lykhenko for fruitful discussions and comments during the initial stages of this work. We would also like to thank Ranadeep Biswas, Daniel Ramos and Ricardo Brancas for the help in deploying CLOTHO. This work was partially supported by the projects PTDC/CCI-INF/32038/2017 (NG-STORAGE) and by UIDB/ 50021/ 2020.

References

1. Hirzalla, M., Cleland-Huang, J., Arsanjani, A. In: A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures. Springer-Verlag, Berlin, Heidelberg (2009) 41–52
2. Santos, N., Rito Silva, A.: A complexity metric for microservices architecture migration. In: 2020 IEEE International Conference on Software Architecture (ICSA). (2020) 169–178

3. Lu, H., Hodsdon, C., Ngo, K., Mu, S., Lloyd, W.: The SNOW theorem and latency-optimal read-only transactions. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USENIX Association (November 2016) 135–150
4. Tomsic, A.Z., Bravo, M., Shapiro, M.: Distributed transactional reads: The strong, the quick, the fresh & the impossible. In: Proceedings of the 19th International Middleware Conference. Middleware '18, New York, NY, USA, Association for Computing Machinery (2018) 120–133
5. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. Proc. VLDB Endow. **7**(3) (November 2013) 181–192
6. Almeida, J.F., Silva, A.R.: Monolith migration complexity tuning through the application of microservices patterns. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **12292 LNCS** (2020) 39–54
7. Aguilera, M.K., Leners, J.B., Kotla, R., Walfish, M.: Yesquel: Scalable sql storage for web applications. In: Proceedings of the 2015 International Conference on Distributed Computing and Networking. ICDCN '15, New York, NY, USA, Association for Computing Machinery (2015)
8. Wu, C., Sreekanti, V., Hellerstein, J.M.: Transactional causal consistency for serverless computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. SIGMOD '20, New York, NY, USA, Association for Computing Machinery (2020) 83–97
9. Rahmani, K., Nagar, K., Delaware, B., Jagannathan, S.: CLOTHO: Directed test generation for weakly consistent database systems. Proc. ACM Program. Lang. **3**(OOPSLA) (October 2019)
10. Richardson, C.: Microservices Patterns: With examples in Java. Manning Publications (2018)
11. Vossen, G. In: ACID Properties. Springer US, Boston, MA (2009) 19–21
12. Lechtenbörger, J. In: Two-Phase Commit Protocol. Springer US, Boston, MA (2009) 3209–3213
13. Gilbert, S., Lynch, N.: Perspectives on the CAP theorem. Computer **45**(2) (2012) 30–36
14. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. SIGMOD '95, New York, NY, USA, Association for Computing Machinery (1995) 1–10
15. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. SIGMOD '87, New York, NY, USA, Association for Computing Machinery (1987) 249–259
16. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.: Session guarantees for weakly consistent replicated data. In: Proceedings of 3rd International Conference on Parallel and Distributed Information Systems. (1994) 140–149
17. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1) (1995) 37–49
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
19. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, and convergence. (05 2012)

20. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11, New York, NY, USA, Association for Computing Machinery (2011) 401–416
21. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. CoRR **abs/1906.12095** (2019)
22. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2020, New York, NY, USA, Association for Computing Machinery (2020) 211–226
23. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.: Static serializability analysis for causal consistency. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2018, New York, NY, USA, Association for Computing Machinery (2018) 90–104
24. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. CoRR **abs/1806.08416** (2018)