Microservice Decomposition for Transactional Causal Consistent Platforms

Madalena Costa Gonçalves Carlos Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

June 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

Firstly, I would like to thank my dissertation supervisor Prof. Luís Rodrigues for his insight, support and sharing of knowledge that has made this Thesis possible.

I must express my gratitude to my parents and sister Sofia for their encouragement, support and caring over all these years, without whom this project would not be possible.

To the rest of my relatives, who have seen less of me during my years of study, thank you for your love and understanding.

To João, thank you for being an inspiration, for always pushing me to be a better person and for providing me with a safe retreat from everything else.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life.

To each and every one of you – Thank you.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO).

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.

Keywords

Microservices, Data Consistency, Serializability Anomalies, Distributed Systems

Resumo

Hoje em dia, grande parte das aplicações desenvolvidas com base em arquiteturas monolíticas poderiam beneficiar da sua decomposição numa arquitetura de microserviços ou, em alguns casos, funções *stateless*. No entanto, para decompor uma aplicação monolítica num conjunto de microserviços, o programador precisa de escrever código adicional que corrija as anomalias que são geradas ao executar a decomposição numa configuração distribuída. Algumas ferramentas de suporte à decomposição de aplicações monolíticas num conjunto de microserviços calculam automaticamente os valores para uma série de métricas de complexidade, que correspondem à aproximação para o esforço necessário para a implementação das ações compensatórias para uma determinada decomposição. Esta informação orienta o programador no objetivo de encontrar a decomposição em microserviços mais apropriada para um certo sistema monolítico. Uma limitação destas ferramentas é o facto de terem sido desenvolvidas sob a consideração de que o ambiente de execução não oferece nenhum tipo de apoio transacional. O objetivo deste trabalho é ampliar a utilidade destas ferramentas com mecanismos que consigam ter em conta diferentes modelos de consistência, em particular a Consistência Causal Transacional. Com esta finalidade em vista, utilizaremos mecanismos automatizados de forma a identificar potenciais anomalias durante a execução de uma decomposição quando o ambiente de execução garante Consistência Causal Transacional. A descoberta destas anomalias pode ser utilizada para guiar a implementação de ações compensatórias e para oferecer uma estimativa para a complexidade associada ao desenvolvimento de uma determinada decomposição.

Palavras Chave

Microserviços, Consistência de Dados, Anomalias de Serializabilidade, Sistemas Distribuídos

Contents

List of Figures

List of Tables

1 Introduction

Contents

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.

1.1 Motivation

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 (Atomicity, Consistency, Isolation and Durability).

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 extended previous works that 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 used automated mechanisms to identify anomalies that can arise during the execution of a given decomposition under the TCC model. In particular, we leveraged 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.

1.2 Contributions

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

We designed 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.

We extended 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.

1.3 Results

This thesis produced the following results:

- A novel and precise calculation for the complexity of decomposing a monolith
- A propotype of the proposed tool
- A detailed explanation of the proposed solution
- A thorough evaluation of the proposed solution, including a comparison of the complexity metric produced by our tool and the complexity metrics proposed in [2]

1.4 Research History

This work is motivated by two distinct research efforts on-going at INESC-ID. First, under the supervision of Prof. António Rito Silva, several students are working on the identification of metrics and tools that can guide the decomposition of a monolithic application into a micro-services architecture. Second, under the supervision of Prof. Luís Rodrigues, several students are working on providing support for Transactional Causal Consistency (TCC) in micro-service and Function-as-a-Service architectures. Our work aims at bridging these research lines, by researching tools that can help to understand (and quantify) the benefits that TCC support can bring when decomposing monoliths.

1.5 Organization of the Document

The rest of this thesis is organized as follows: Chapter 2 introduces relevant concepts and an overview of the related work; Chapter 3 describes the frameworks and methods used to calculate system complexity; Chapter 4 reveals the results of the experimental evaluation; Chapter 5 concludes this document by outlining the main contributions and directions for future work.

2

Related Work

Contents

This section aims to introduce concepts that will be relevant to the understanding of the document, as well as a review of the related work. We start by briefly discussing the advantages and disadvantages of microservices architectures versus monolithic architectures, before adressing the challenges of providing strong consistency in microservices architectures. We then review commonly used strategies to mitigate the negative effects of weak consistency in microservice architectures. After all background concepts have been introduced, a literary review of the most relevant works for this project is done: we briefly introduce two tools that have been designed with the goal of supporting the decomposition of monolithic applications into microservices. Finally, we assess a number of works that provide mechanisms or tools to determine consistency anomalies in the form of serializability violations of programs.

2.1 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.

2.2 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]: Atomicity, stating that all changes to data are performed as if they were a single operation and either all changes happen or none do; Consistency, which requires that the transactions always leave the database in consistent states that respect business rules; Isolation, specifying that intermediate states of a transaction should not be seen by other transactions; Durability, 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.

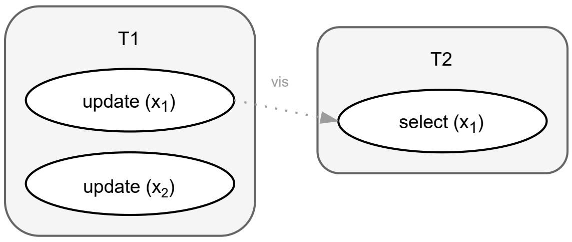
2.3 Consistency in Microservice Architectures

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.

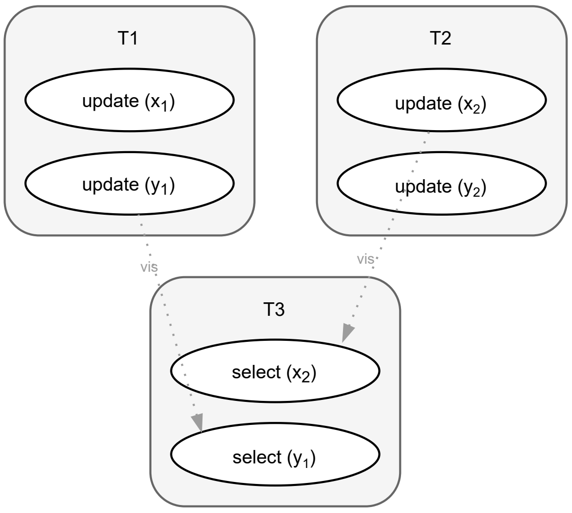
Most microservice architectures favour availability and, therefore, avoid depending on distributed transactions that span multiple services. Instead, transactions can be used internally by each individual microservice, such that functionalities that are executed by the same microservice are isolated 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.

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. These anomalies are illustrated with the help of figures figs. 2.1(a), 2.1(b), 2.2(a), 2.2(b) and 2.3, where each transaction is represented by a gray box, and operations of transactions are represented by ellipses inside the boxes. The connecting dotted arrow with label *vis* represents the *visibility* relation between operations. If two operations are connected by such arrow, it means that the effects of the first can be seen by the second.

Dirty Read A Dirty Read anomaly is the result of some transaction being allowed to observe the effects of either uncommitted, aborted or intermediate states of another transaction executing concurrently. In fig. 2.1(a), transaction T2 reads x_1. This value results of an intermediate state of transaction T1, considering that it is overwritten by value x_2 later in the same transaction. In a serial execution, the read made by T2 should return the committed value x_2.



(a) Dirty read



(b) Dirty write

Figure 2.1: Consistency anomalies: dirty read and dirty write

Dirty Write Berenson *et al.* [14] define a Dirty Write anomaly by drawing on the following example: some transaction T updates one data object and, before it has the opportunity to commit or rollback, transaction T' initiates and updates the same data object. If either transaction T or T' were to rollback, it is unclear what the value for the data object should be.

For the context of this work, we define this anomaly as the behavior perceived by a third transaction T3, when reading updates made by two concurrent transactions, T1 and T2. Under atomic executions, T3 should observe either version 1 or version 2 of both objects, never a different version of each object, which is what is depicted in fig. 2.1(b).

Lost Update A lost update anomaly (fig. 2.2(a)) occurs when two transactions, T1 and T2, update the same object concurrently based on their local values for that object. Because neither of the two transactions observes the other's update, this anomaly originates a faulty behavior. If we consider the initial value $x_1 = 0$, a serial execution of the two transactions would lead to a final state of $x = 20$. However, in a concurrent execution of this example program, if the underlying consistency model does not forbid this behavior, the final state could yield a value of either $x = 10$ or $x = 20$.

Write Skew The Write Skew anomaly (fig. 2.2(b)) can be described as a generalization of 2.3 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*. This is commonly the case when some program state needs to be maintained before making a modification to a certain object. For example, consider a banking system where x and y represent the current values of two different accounts. Transaction T1 reads the value of one account, and, if some constraint is observed, updates the other account. Concurrently, transaction T2 reads the value of the account that is being modified by T1, and decides to update the other account. Since none of the transactions is able to observe the other's updates, an inconsistent final state will be originated by this execution, and user constraints may be violated.

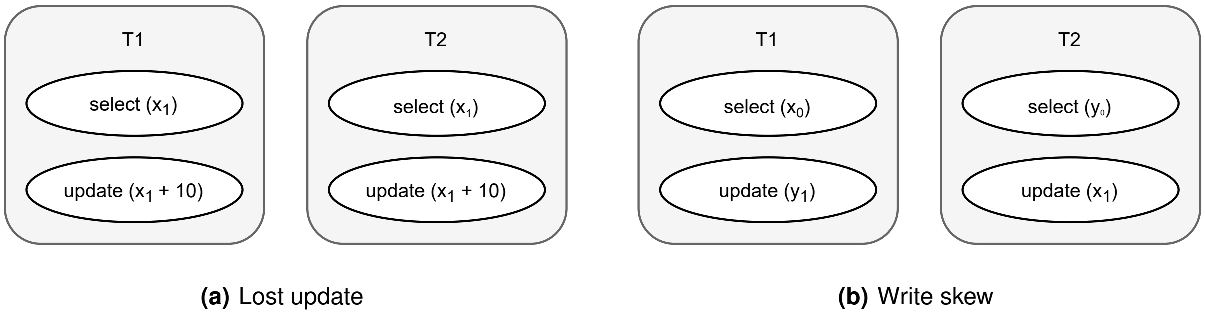


Figure 2.2: Consistency anomalies: lost update and write skew

Long Fork A Long Fork anomaly (fig. 2.3) happens whenever two concurrent transactions, T1 and T2 make concurrent updates to distinct objects. A third transaction, T3, can see the update made by T2, but not the one made by T1. A fourth transaction, T4, only sees the update made by T1. Therefore, from the perspectives of T3 and T4, the writes made by T1 and T2 happen in different orders.

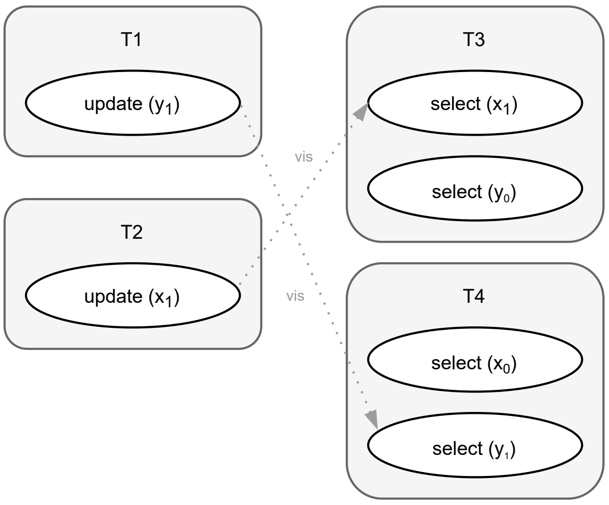


Figure 2.3: Consistency anomaly: long fork

2.4 Weak Consistency

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 serializability, 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 weaker consistency models that have been proposed in the literature.

2.4.1 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.* [15] 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 [16, 17], which has been proven to be the strongest guarantee compatible with high availability [5, 18]. 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.

2.4.2 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]. We define a system as highly available if a user that can contact at least one of the nodes in the system is guaranteed to always get a response, even in the case of network partitions, which would prevent that node to communicate with others in the system. Consistency 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 - also called **Repeatable Read** - states that each transaction reads from a non-changing cut or snapshot over the data items, which means that if a transaction reads the same data more than once, it sees the same value each time. 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 [19].

2.5 Managing Weak Consistency

The Saga [20] 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].

2.6 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.

2.6.1 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.

2.6.2 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.

2.7 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.

2.7.1 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 2.4. 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.

2.7.2 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.

2.7.3 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.

2.7.4 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.

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

CLOTHO [9], an improvement of the tool of name ANODE discussed in Section 2.7.4, 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 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.

2.7.6 Comparison

We now provide a brief comparison of the systems surveyed in the previous paragraphs. Table 2.1 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

	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 2.1: Comparison of different approaches for detecting anomalies in transactional programs

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.

3

Using CLOTHO to Model Microservices

Contents

This chapter describes the work produced on the scope of this project, a study on the complexity of decomposing a monolithic system into a set of microservices, when considering that the execution environment provides some consistency guarantees. Section 3.1 describes the goals our work aims to achieve. Section 3.3 portrays the obstacles to the application of existing frameworks and the settlements and solutions implemented. Section 3.4 describes all the modifications made to CLOTHO [9] in order to accomplish the proposed mission.

3.1 Goals

This work addresses the problem of estimating the complexity of decomposing a monolithic system into microservices for execution environments that support various consistency models, such as Transactional Causal Consistency.

We have implemented a novel method for calculating the complexity of decomposing a monolithic system into microservices, on the basis that the number of anomalies occurring in the execution of a set of microservices equates to effort that needs to be put into developing compensating actions to mitigate these anomalies. Our work leverages upon the efforts made by the authors of [9] and [2].

3.2 CLOTHO

CLOTHO is a testing framework for detecting serializability violations in Java database applications that execute on weakly consistent storage systems. This application combines a static analyzer and a model checker to generate abstract executions and identify serializability violations in concrete executions of the input program. CLOTHO also translates these anomalies back into test inputs, with the specific parameters to the transactions that originated the anomalies. The input programs for CLOTHO must contain a Java class that manipulates a database through the standard JDBC API, where each method is treated as a transaction.

In the first stage of the analysis, the input program is translated into an abstract representation, which captures key features of the program, including the database schema, the set of transactions, and the set of operations (data retrieval and modification) that each transaction consists of. It is assumed that CLOTHO's input programs are interpreted on a finite number of partitions, each of which has its own copy of the database. The execution of these programs is described as a finite sequence of system states. Each state is represented by a triple, $(\mathsf{str}, \mathsf{ar}, \mathsf{vis})$, where str is a set of operations of the program, ar records the exact sequence of database operations that have been executed and vis relates two effects if one witnesses the other at the time of creation. A local view of the database at each partition can be constructed from a system state by applying the effects of operations stored in the str component of the partition according to the respective order in the ar component.

After constructing the abstract model from the input program, CLOTHO begins the identification of non-serializable executions in the abstract representation of the program. The authors consider that a certain execution of the input program is serializable if there exists another strictly serial execution that is constructed by reordering the operations of the first execution, such that the final set of effects in both executions is equivalent. Programs that contain a serializability anomaly can be decomposed into a serial execution followed by a non-serializable execution.

The problem of determining the serializability of an execution is reduced to the detection of cycles in the dependency graph of the final state of the execution. The nodes of this graph are the set of operations in an execution state. The edges of the dependency graph are from the set of dependency relations $\{\mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$. The Read dependency, WR, relates two operations if one witnesses a value that is written by the other; the Write dependency, WW, relates two operations if one overwrites the value written by the other; the Read anti-dependency, RW, relates two operations if one witnesses a value that is later overwritten by the other. Recalling the previous definition of serializability, the reordering of the operations to obtain an equivalent serial execution cannot exist if there are cyclical dependencies in the dependency graph. Thus, if there is a cycle in the dependency graph of the execution of a program, such program is not serializable.

The identification of dependency cycles in an execution is done by checking the satisfiability of a First-Order-Logic formula. This formula contains variables for each of the dependency, visibility and arbitration constraints in the execution, and is designed such that the assignments to these variables in a satisfying model can be used to reconstruct the anomalous execution of the original program. One of the components of the FOL encoding contains the rules for the consistency model of the underlying database. These can be described using the previously mentioned vis and ar relations between read and write operations. The authors of CLOTHO include the constraints for a few popular consistency models, but CLOTHO itself does not implement any of them. This means that all transactions are executed considering that replicas of the system maintain no synchronization between each other - Eventual Consistency (EC).

The output of this tool is the number of serializability anomalies found in the input program. For each of the discovered anomalies, CLOTHO automatically determines the execution order of queries and values of input arguments required for its manifestation. This information can be viewed in the form of a graphic that also includes dependency and visibility relations between operations in the anomaly.

Although it is not our main focus for this project, another functionality of CLOTHO is the ability to replicate anomalous executions on a concrete environment. A cluster of Docker containers is created in order to run replicated database instances that have been instantiated according to the test configuration - a database initialization file and a set of Java class files annotated with execution orders and input parameters.

The Analyzer module of CLOTHO contains two different components: the front-end compiler that performs the analysis of the input Java program, detecting loops, conditional structures and database accesses, and the Z3 component, which makes use of the Z3 library [25] to determine satisfying assignments to the constructed FOL formula.

In order to supply CLOTHO with different consistency models other than Eventual Consistency, we had to provide the SMT solver Z3 with stronger constraints. We followed the guide presented on the work of Rahmani et al. [9], which proposed the following constraints:

Causal Visibility

$$\Psi_{\mathrm{CV}} = \forall(o_1, o_2, o_3) \cdot \mathsf{vis}(o_1, o_2) \wedge \mathsf{vis}(o_2, o_3) \Rightarrow \mathsf{vis}(o_1, o_3) \tag{3.1}$$

The predicate vis relates two operations o_1 and o_2, $\mathsf{vis}(o_1, o_2)$, if o_1 is able to observe the effects produced by o_2.

The eq. (3.1) states that if an operation o_1 is able to observe the effects caused by operation o_2 and, in turn, operation o_2 observes the effects caused by operation o_3, then o_1 also observes the effects caused by operation o_3. This is, thus, called Causal Visibility.

Causal Consistency

$$\Psi_{\mathrm{CC}} = \forall(o_1, o_2) \cdot \Psi_{\mathrm{CV}} \wedge (\mathsf{st}(o_1, o_2) \Rightarrow \mathsf{vis}(o_1, o_2) \vee \mathsf{vis}(o_2, o_1)) \tag{3.2}$$

The predicate st relates two operations x and y, $\mathsf{st}(x, y)$, if x and y have the same parent transaction, this is, they are both operations of the $\underline{\text{s}}$ame $\underline{\text{t}}$ransaction.

Causal Consistency (eq. (3.2)) states that Causal Visibility must be established and that if two operations o_1 and o_2 are in the same transaction, then either o_1 observes the effects of o_2 or o_2 observes the effects of o_1.

Read Committed

$$\Psi_{\mathrm{RC}} = \forall(o_1, o_2, o_3) \cdot \mathsf{st}(o_1, o_2) \wedge \mathsf{vis}(o_1, o_3) \Rightarrow \mathsf{vis}(o_2, o_3) \tag{3.3}$$

Read Committed (eq. (3.3)) states that if two operations are in the same transaction and if one of them is able to observe the effects of a third, then the second one should also be able to observe the effects of it. This grants some atomicity to the execution environment.

Repeatable Read

$$\Psi_{\mathrm{RR}} = \forall(o_1, o_2, o_3) \cdot \mathsf{st}(o_1, o_2) \wedge \mathsf{vis}(o_3, o_1) \Rightarrow \mathsf{vis}(o_3, o_2) \tag{3.4}$$

The Repeatable Read consistency model (eq. (3.4)) obliges that if one operation is able to observe the effects of another operation o_1, then it should also be able to observe the effects of all operations in the same transaction as o_1

Linearizability

$$\Psi_{\text{LIN}} = \mathsf{ar} \subseteq \mathsf{vis} \tag{3.5}$$

The predicate c relates two operations o1 o_1 and o_2, $\mathsf{ar}(o_1, o_2)$, if o_1 is arbitrarily ordered to happen before o_2. Linearizability (eq. (3.5)) states that all database operations are able to observe the effects of operations that were arbitrarily ordered before them. Thus, if the relation $\mathsf{vis}(o_1, o_2)$ is true, so is $\mathsf{ar}(o_1, o_2)$.

Serializability

$$\Psi_{\text{SER}} = \Psi_{\text{RC}} \wedge \Psi_{\text{RR}} \wedge \Psi_{\text{LIN}} \tag{3.6}$$

Serializability, captured by eq. (3.6) is a combination of Read Committed (eq. (3.3)), Repeatable Read (eq. (3.4)) and Linearizability (eq. (3.5)).

Considering that the main focus of our work was, initially, to explore the results of providing Transactional Causal Consistency guarantees to distributed systems, we ought to extend the Causal Consistency model that was implemented in CLOTHO in order to reproduce this. We propose the following isolation guarantee:

Transactional Causal Consistency

$$\Psi_{\text{TCC}} = \forall(o_1, o_2, o_3) \cdot \Psi_{\text{CC}} \wedge (\mathsf{vis}(o_1, o_2) \wedge \mathsf{st}(o_2, o_3) \Rightarrow \mathsf{vis}(o_1, o_3)) \tag{3.7}$$

The line of thought behind the implementation of Transactional Causal Consistency (eq. (3.7)) is that it is an extension of Causal Consistency that considers causal relations on multiple operations over multiple objects. Thus, if one operation o_2 is visible from the point-of-view of operation o_1, then all operations that belong to the same transaction as o_2 should also be visible to o_1.

The above isolation guarantees were implemented by extending the Analyzer module of CLOTHO. All these models were implemented in the context of this work, and were tested making use of the benchmarks developed by the authors of CLOTHO: Dirty Read, Dirty Write, Long Fork, Lost Update and Write Skew.

The reason for the use of these benchmarks for the testing of our own extension to CLOTHO is that it provided a stable and forward manner to validate the correction of the different consistency models, seeing that the results of the execution of the different consistency anomalies under the implemented guarantees is theoretically known. The results and settings of these tests can be seen under Chapter 4

3.3 Obstacles

During the course of this project, a number of obstacles occurred in the adaptation of CLOTHO.

CLOTHO was originally designed to test the correction of programs making use of distributed databases. This means that it assumes a scenario where different replicas execute the same program (e.g. the same code) and eventually, concurrency problems will manifest due to different replicas having to maintain different versions of the same database.

However, Microservices do not work the same. The Microservice architectural model is based on the premise of separation of concerns, which means that each service executes different functionalities, and can make use of different technologies and databases. By principle, two different services will be responsible for two different sets of system domain entities, and concurrency challenges will not arise due to maintaining two different versions of the same entity, but because one needs to make sense of different entities that should look that they were updated atomically.

CLOTHO keeps one copy of the same database in each replica and this ultimately means that we cannot simulate an architecture of Microservices where each service keeps different schemas and entities.

In order to make sense of this problem, we routed towards a different solution, one that would make it possible to use CLOTHO as is to test a distributed system with some level of separation of concerns.

We considered *FaaS* as an alternative to the Microservices architetural style. *Function-as-a-Service* describes an application where storage services are disaggregated from the machines that support function execution. These applications consist of compositions of functions, where each function may run on a separate machine and access remote storage. Programmers can upload arbitrary functions and execute them in the cloud without having to provision or maintain the servers. Because different functions may not run on the same machine, the challenge of maintaining data consistency rises.

In some ways, a parallel between the FaaS and Microservices architectural models can be established: functions in the FaaS model would correspond to different services in the Microservices model. Increasing the number of instances running the same function is equivalent to adding replicas to a service in the Microservice architecture. In this sense, we could adapt our initial idea to be applied to a FaaS system with one single replicated function and calculate the number of anomalies that would arise in this context. Then, we would have to alter the calculations in the complexity metric given by [2] in order to be able to apply it to a FaaS system instead of a Microservices' one.

However, when studying the complexity metric in [2], we understood that this was not feasible, since the complexity value for any system where there is only one cluster of domain entities - which is the case in a FaaS system with a single function - is always zero. This value would not be comparable to the number of anomalies output by CLOTHO.

In mono2micro [2], the complexity value is determined based on similarity measures. In a 1-function FaaS system, all domain entities of the system would be accessed by the same function, the only one that exists. This leads to all domain entities being clustered in the same service. Since there is no decomposition, the complexity of decomposing this system is zero.

From a different point of view, let us suppose we would generate bogus domain entity decompositions in this 1-function FaaS system. In order to create decompositions for which the complexity value is non-zero, there would have to different transactions altering different databases.

It is not possible to mimic the partition of domain entities in CLOTHO.

One possible solution to this problem is to limit the interactions between transactions in CLOTHO in pursuance of emulating the system of microservices. The complexity metric in [2] only considers concurrency issues happening in distributed transactions, i.e, that traverse multiple services. To convey this pattern to CLOTHO, we would need to only consider serializability anomalies happening between transactions of different services. One way to implement this is to label the transactions according to the service they would execute on, and modify CLOTHO to only analyze anomalies between transactions that have different labels on them. Anomalies inside each service need not be considered because they execute under Serializability or some other strong consistency model. Only the anomalies in interactions between different services add to the complexity of decomposing a monolith.

Another hindrance to our work was the fact that, in order to use CLOTHO to test applications, these needed to be in the form of Java classes, where each method is a transaction in the system. Not a lot of real-life applications are built this way. Mainly, the programs used for testing during the course of this project were small examples assembled by us and translated into code that could be processed by CLOTHO.

3.4 Implementation

3.4.1 Changes Made to CLOTHO

3.4.1.A Consistency Models

CLOTHO, by default, assumes an execution environment where no isolation or atomicity guarantees are given. The seven consistency guarantees discussed in 3.2 were implemented during the scope of this project. To do so, we created new code on the Z3 component of CLOTHO. This code makes use of the Java binding for Z3.

3.4.1.B Distinguishing Labels for Transactions

As mentioned in 3.3, one aspect to the adaptation of CLOTHO was to figure out how to mimic the distributed nature of Microservices. Each service should execute different local transactions, where serializability (or other equivalent strong consistency criteria) is maintained. To this extent, anomalies found under the interactions of the same transaction are negligible.

With the purpose of limiting the analysis to only transactions executing in different services, we attributed labels to each transaction, signifying the service/cluster that it would be executed on.

Further on, during the analysis of the anomalies in a program, we only allowed CLOTHO to check for anomalies between a pair of transactions that did not belong to the same cluster, i.e, was not attributed the same label.

This was done by skipping loops in the code whenever the label for the transaction was identical.

3.4.1.C Implementation of New Relation Between Operations

During the implementation of the new consistency models, we came across the lack of a relation between operations: the st relation - designating operations that belong to the $\underline{s}$ame $\underline{t}$ransaction.

After analyzing the code of the CLOTHO framework, we found a different but similar relation, the $\mathsf{sibling}$ relation, that relates two operations if they belonge to the same transaction instance.

The two are different: st is more general, since it is true for all operations for which the $\mathsf{sibling}$ relation is true, but it is also true for operations for which the $\mathsf{sibling}$ relation is not true but the class originating the parents of those operations is the same.

For two different instances of the same transaction class, all child operations belong to the same transaction, but the siblings are the pairs of operations inside each transaction instance.

Summary This chapter presents the steps made towards the accomplishments of the goals to our work. It explains the obstacles and conclusions taken during the implementation of extensions of CLOTHO, and the specific modifications made to it. The next chapter addresses the experimental evaluation of our work and presents evidence to prove the pertinency of our work.

4 Evaluation

Contents

4.1 Goals of the Evaluation

The central goal of this evaluation is to determine if our extensions of CLOTHO allow it to lawfully predict the number of serializability anomalies in real weakly-consistent database applications.

The value for the number of serializability anomalies aims to be equivalent to a measure of complexity for such distributed applications. Ideally, our proposed metric should be in line with theoretical conceptions for the complexity of decomposing a monolithic system in several microservices, which means that the higher the number of anomalies, the more difficult it should be to devise such application.

Our metric must also be compared with the one proposed by [2], so as to verify that it agrees with the common notion of complexity in a distributed system, but also to establish if our intention to strengthen previously proposed metrics is successfully carried out.

Finally, the adjustments made to CLOTHO should not impair its performance or functionality.

4.2 Experimental Setting

The evaluation process was structured into different segments: the validation of the implemented consistency criteria and the verification of the newly proposed complexity metric.

4.2.1 Validating the Implemented Consistency Models

The first step for evaluating our program and its correctness is to test the newly implemented consistency criteria. In order to do so, we compare theoretical values for the number of anomalies in several transactional applications under different consistency levels with the ones calculated by our program.

The following are the benchmarks used to perform the assessments. These are Java classes that implement methods representing different transactions used as input to CLOTHO.

4.2.1.A Dirty Write

```
two_reads(key):
    read(key)
    read(key)

one_write(key, x):
    write(key, x)
```

Listing 4.1: Dirty Write class abstraction as used in CLOTHO

4.2.1.B Dirty Read

```
one_read(key):
    read(key)

two_writes(key, x, y):
    write(key, x)
    write(key, y)
```

Listing 4.2: Dirty Read class abstraction as used in CLOTHO

4.2.1.C Long Fork

```
two_reads(key1, key2):
    read(key1)
    read(key2)

one_write(key, x):
    write(key, x)
```

Listing 4.3: Long Fork class abstraction as used in CLOTHO

4.2.1.D Lost Update

```
one_increment(key, amount):
    x = read(key)
    write(key, x+amount)
```

Listing 4.4: Lost Update class abstraction as used in CLOTHO

4.2.1.E Write Skew

```
one_increment(key1, key2, x):
    read(key1)
    write(key2, x)
```

Listing 4.5: Write Skew class abstraction as used in CLOTHO

	EC	CV	CC	TCC	RC	RR	LIN	SER
Dirty Read	✓	✓	✓	✓	✓	x	x	x
Dirty Write	✓	✓	✓	✓	x	✓	x	x
Long Fork	✓	✓	✓	✓	✓	✓	x	x
Lost Update	✓	✓	✓	✓	✓	✓	✓	x
Write Skew	✓	✓	✓	✓	✓	✓	✓	x

Table 4.1: Expected results for the existence of anomalies in each benchmark for each consistency model. The tick symbol (✓) indicates that the consistency allows some anomalies for the benchmark, whereas the cross symbol (x) implies that the consistency model does not allow any anomalies for such benchmark

	EC	CV	CC	TCC	RC	RR	LIN	SER
Dirty Read	1	1	0	0	0	1	1	0
Dirty Write	3	3	3	0	3	0	1	0
Long Fork	3	3	3	0	3	0	1	0
Lost Update	2	2	2	2	2	2	0	0
Write Skew	2	2	2	2	2	2	0	0

Table 4.2: Observed number of serializability anomalies when executing benchmarks in CLOTHO under each consistency model

The five benchmarks were tested under the different consistency criteria, where **EC** refers to Eventual Consistency, **CV** to Causal Visibility, **CC** to Causal Consistency, **TCC** to Transactional Causal Consistency, **RC** to Read Committed, **RR** to Repeatable Read, **LIN** to Linearizability and **SER** to Serializability. Table 4.1 assembles the expected values for the number of anomalies of each benchmark under a certain consistency model, according to academical research.

Table 4.2 presents the results obtained when executing the modified version of CLOTHO with each of the different benchmark classes as inputs, while tweaking the consistency level by wavering between the implemented consistency models. The value in each cell represent the results observed after a large number of executions of each test instance under each model, to safeguard the reliability of these tests.

4.2.2 Validating the Complexity Metric

Mono2micro [2] collects data from codebases implemented with Spring Boot and the Fenix Framework, and can only be executed using previously developed applications BlendedWorkflow, FenixEdu Academic and LdoD, which are much too complex to be tested by CLOTHO. In order to validate our new complexity metric, we need to test both CLOTHO and mono2micro using the same input application, which entails producing a new test example that is complex enough to manifest different behaviours when analyzed by the two tools.

Instead, we developed a new test scenario small enough to be manually tested by mono2micro, and convert to Java code to be tested by CLOTHO. This test instance consists of three simple transactions:

```
write_A(int key):
    write(var_A, key)

write_B(int key):
    read(var_A)
    write(var_B, key)

check_vars():
    var_A = read(var_A)
    var_B = read(var_B)
    assert_coherence(var_A, var_B)
```

Listing 4.6: Test class abstraction as used in CLOTHO

The intention of this test instance is to mimic the following scenario: the microservice system managing this application consists of two different microservices: the first is responsible for handling domain entity A and the second for domain entity B. Each service executes, respectively, the first and second transactions in listing 4.6, and transaction $check\_vars$ is executed by both services, where the read to domain entity A is made in the first service, and the read to entity B is made in the second service. In respect to transaction $check\_vars$, asserting the coherence of domain entities implies that the serializability of the writes of these variables was respected.

Computing the value for decomposition complexity using mono2micro's metric was the next step. The following formulae (eqs. (4.1) to (4.3)) were used, as presented in [2]:

$$complexity(d) = \frac{\sum_{f \in F} complexity(f, d)}{\#F} \tag{4.1}$$

Equation 4.1: Complexity of a microservice decomposition: the complexity of a decomposition is given by the average of the complexities of its functionalities

$$complexity(f, d) = \sum_{c \in C} complexity(c, f, d) \tag{4.2}$$

Equation 4.2: Complexity of a functionality f in decomposition d: the complexity of a functionality is given by the sum of the complexities of accessing each service c in the sequence of accesses made by f. Note that if functionality f is not distributed, its complexity is 0

Finally, the complexity of accessing an entity through functionality f in decomposition d, $complexity(a, f, d)$ is given by the number of other distributed functionalities that access that same entity using a different access mode. Access modes can be either *read* or *write*. Then, the complexity of reading an entity is

$$complexity(c, f, d) = \# \cup_{a \in c} complexity(a, f, d) \tag{4.3}$$

Equation 4.3: Complexity of accessing service c through functionality f in decomposition d: the complexity of accessing a service is the cardinality of the union of the complexities of each entity accessed by f in s

	EC	CV	CC	TCC	RC	RR	LIN	SER
Example Program	3	3	0	0	0	1	1	0

Table 4.3: Observed number of serializability anomalies when executing our test program in CLOTHO under each consistency model

the number of other distributed functionalities that write to it, and the complexity of writing to an entity is the number of other distributed functionalities that read it.

The complexity of the decomposition presented in listing 4.6 can be calculated using the following reasoning. There are two different services: $cluster_A$, dealing with domain entity A, and $cluster_B$, dealing with domain entity B. There are three functionalities (transactions): $write\_A$, which is composed of a write operation to domain entity A; $write\_B$, which is composed of a read operation of domain entity A, followed by a write operation to domain entity B; $check\_vars$, which is composed of a read operation of domain entity A followed by a read operation of domain entity B.

The complexity of the decomposition is given by the average of the values for the complexity of the three functionalities. Functionality $write\_A$ is not distributed - it only accesses service $cluster_A$, and thus, its complexity is 0. Functionality $write\_B$ is a distributed functionality, and accesses service $cluster_A$ and service $cluster_B$. Then, the complexity of functionality $write\_B$ is given by the sum of the complexities of accessing the two services. The complexity of accessing service $cluster_A$ is the cardinality of the union of the complexities of each entity accessed by $write\_B$ in $cluster_A$. Only one access is made by that functionality in that service: a read operation to domain entity A. The complexity of reading A is the number of other distributed functionalities that write to it, which is zero. The complexity of accessing $cluster_B$ in the context of functionality $write\_B$ is 1, since there is one other distributed functionality that reads B, $check\_vars$. Thus, the complexity of functionality $write\_B$ has a value of 1. The complexity of functionality $check\_vars$ can be computed in a similar way, and its value is also 1. Finally, the average of the complexities of the three functionalities has a value of $\frac{2}{3}$.

The following phase for validating our complexity metric is to test the same program using CLOTHO. Although mono2micro only considers Eventual Consistency, CLOTHO is able to determine the number of serializability anomalies when the program is executed under multiple consistency models. We decided to include this strengthening of the underlying consistency model to demonstrate the benefit of improving the level of isolation on microservice systems. The results for the evaluation of such program with CLOTHO are displayed on table 4.3

4.3 Discussion

While analyzing the results of the evaluation presented on the previous sections, the following was concluded: in relation to the implemented consistency criteria, we consider that this development was successful, since all the obtained values conform to what would be expected, as seen in tables 4.1 and 4.2.

Regarding the soundness and relevance of the new complexity metric given by analyzing test input programs using CLOTHO, we consider that it is a pertinent novel way to estimate the expected effort to decompose a monolithic system, as it determines exactly what the key serializability violations in the input programs are, giving programmers thorough evidence of where that effort must be applied in order to develop correct and sound distributed systems. Comparing the metrics produced by this work and the work of mono2micro [2], we observe that, although the numbers on the developed test instance are not significantly different, due to the fact that said instance is not extensive and serves only as a proof-of-concept, our metric and testing tool provide more far-reaching and meaningful hints concerning the points of failure of a microservice system.

Adding to this, our metric also considers different consistency models, which means it is more versatile in regards to the set of programs and systems that it can be applied to. Mono2micro overlooks the existence of stronger consistency criteria, and always assumes Eventual Consistency. Using CLOTHO also allows us to study the direct improvement that strengthening the consistency model of the underlying system can have on the simplification of the development of the transactional programs on top of these systems.

Summary In this chapter, we have presented the experimental evaluation of our work, describing the conditions under which these tests were conducted in order to analyze its relevance and correctness. The results show that our implementation is in conformance to theoretical expectations and proves to be relevant due to the new information added to previously developed metrics. The next chapter concludes this document by making the final conclusions and discussing possible ideas for future work.

5 Conclusion

Contents

5.1 Conclusions

Partitioning a monolithic application into different services is not an easy task: although the decomposition of responsibilities provides some benefits, it also interferes with the effort of reasoning about system logic. This work provides some insight into the intricacy of monolith decomposition, by improving previously developed metrics to assess the complexity of this development, and adding important information to the points of failure of the resulting systems. The results of this project proved to be relevant and meaningful in the context of our research and this document presents as a guide for the interpretation of complexity, points of failure and decomposition in distributed systems, mainly in the microservice architectural model.

5.2 Limitations and Future Work

This project was limited by a few obstacles, namely in the usage of tools that were not completely appropriate to microservice systems. A possible path for future development is to further extend our work to better adapt to a microservice architectural style, mainly by adding to CLOTHO the capability to segregate responsibilities in different services.

Bibliography

[1] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, *A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 41–52. [Online]. Available: https://doi.org/10.1007/978-3-642-01247-1_5

[2] N. Santos and A. Rito Silva, "A complexity metric for microservices architecture migration," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

[3] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "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, Nov. 2016, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lu

[4] A. Z. Tomsic, M. Bravo, and M. Shapiro, "Distributed transactional reads: The strong, the quick, the fresh & the impossible," in *Proceedings of the 19th International Middleware Conference*, ser. Middleware '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 120–133. [Online]. Available: https://doi.org/10.1145/3274808.3274818

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, Nov. 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[6] J. F. Almeida and A. R. Silva, "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)*, vol. 12292 LNCS, pp. 39–54, 2020.

[7] M. K. Aguilera, J. B. Leners, R. Kotla, and M. Walfish, "Yesquel: Scalable sql storage for web applications," in *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ser. ICDCN '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2684464.2684504

[8] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management*

of Data, ser. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 83–97. [Online]. Available: https://doi.org/10.1145/3318464.3389710

[9] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan, “CLOTHO: Directed test generation for weakly consistent database systems,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360543

[10] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018. [Online]. Available: https://books.google.pt/books?id=UeK1swEACAAJ

[11] G. Vossen, *ACID Properties*. Boston, MA: Springer US, 2009, pp. 19–21. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_831

[12] J. Lechtenbörger, *Two-Phase Commit Protocol*. Boston, MA: Springer US, 2009, pp. 3209–3213. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_2

[13] S. Gilbert and N. Lynch, “Perspectives on the CAP theorem,” *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[14] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ANSI SQL isolation levels,” in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 1–10. [Online]. Available: https://doi.org/10.1145/223784.223785

[15] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, “Session guarantees for weakly consistent replicated data,” in *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, 1994, pp. 140–149.

[16] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995. [Online]. Available: https://doi.org/10.1007/BF01784241

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

[18] P. Mahajan, L. Alvisi, and M. Dahlin, “Consistency, availability, and convergence,” 05 2012.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 401–416. [Online]. Available: https://doi.org/10.1145/2043556.2043593

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

[21] S. M. Beillahi, A. Bouajjani, and C. Enea, “Robustness against transactional causal consistency,” *CoRR*, vol. abs/1906.12095, 2019. [Online]. Available: http://arxiv.org/abs/1906.12095

[22] O. Lahav and U. Boker, “Decidable verification under a causally consistent shared memory,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 211–226. [Online]. Available: https://doi.org/10.1145/3385412.3385966

[23] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, “Static serializability analysis for causal consistency,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 90–104. [Online]. Available: https://doi.org/10.1145/3192366.3192415

[24] K. Nagar and S. Jagannathan, “Automated detection of serializability violations under weak consistency,” *CoRR*, vol. abs/1806.08416, 2018. [Online]. Available: http://arxiv.org/abs/1806.08416

[25] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’08/ETAPS’08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.