Microservice Decomposition for Transactional Causal Consistent Platforms

Madalena Santos
madalenacsantos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2022

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

1. Introduction

The microservice architectural style has been widely adopted for the past years. In opposition to monolithic architectures, microservice architectures decompose an application into a set of small and well-contained services, each having its own cohesive set of responsibilities. This modularization of the system function offers benefits, mainly the ease of development and testing due to the reduction of complexity of services. 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.

Implementing a microservice architecture can bring many advantages, but can also impose additional complexity during the development. Services need to be able to handle faulty behaviour and unavailability of other services. 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.

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 is not trivial. To ease this burden, a number of tools to support this decomposition 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 [6, 13]. 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) [9, 14] in this context.

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[11], a framework that detects serializability violations of Java applications executing on top of weakly consistent distributed databases. CLOTHO discovers serializability anomalies in these executions and translates 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.

2. Background and Related Work

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 [12].

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[12], 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, 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[15]: 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[8]. For these reasons, the consistency models implemented by modern datastores are often weaker than serializability. Bailis *et al.* [2] conducted a survey where 18 off-the-shelf popular database systems were analyzed, and only three of those provided serializability as the default consistency model. Perhaps surprisingly, eight 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[5], 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.

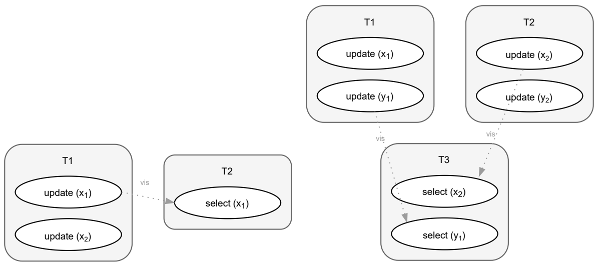
Most microservice architectures favour availability and, therefore, avoid depending on distributed transactions that span multiple services. Instead, transactions can be used internally by each individual microservice, such that functionalities that are executed by the same microservice are isolated 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, programmers need to deal explicitly with the anomalies that may be generated when executing different microservices concurrently, without isolation. In following paragraphs 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 1a, 1b, 3, 2a and 2b, 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 Figure 1a, 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.

Dirty Write Berenson *et al.* [3] 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 Figure 1b.

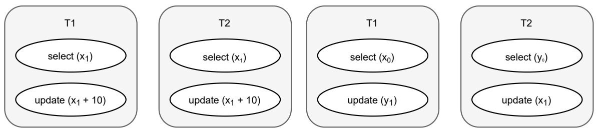


(a) Dirty Read **(b)** Dirty Write

Figure 1: Consistency anomalies: Dirty Read and Dirty Write

Lost Update A lost update anomaly (Figure 2a) 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 (Figure 2b) 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.



(a) Lost Update **(b)** Write Skew

Figure 2: Consistency anomalies: Lost Update and Write Skew

Long Fork A Long Fork anomaly (Figure 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.

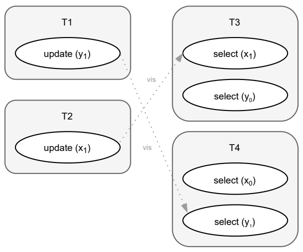


Figure 3: Consistency anomaly: Long Fork

2.4. Transactional Causal Consistency

Causal Consistent memory [1, 7] is a widely studied criterium for distributed systems, as it provides an ideal equilibrium between availability and consistency. It has been proven to be the strongest guarantee compatible with high availability [10, 2]. Causal Consistency 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. Transactional Causal Consistency is the transactional extension to Causal Consistency, guaranteeing the consistency of reads and writes for a set of keys - a transaction - by demanding that all operations are applied on top of the same causal snapshot.

2.5. Decomposing Monolithic Applications into Microservice Compositions

In [13], 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.

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. 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. Verifying Serializability of Applications to Calculate Complexity

While the previously-mentioned work provides essential insights on how to determine boundaries between microservices and contribute with valuable complexity metrics, it only reasons 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, but in this document we will only discuss the most relevant one. The remaining studied frameworks are included in the dissertation.

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

CLOTHO [11] 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.

3. Using CLOTHO to Model Microservices

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 [11] and [13].

In the first stage of the analysis made by CLOTHO [11], 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.

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.

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[4] 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.[11], which proposed the following constraints, for which the logic formulae and description in detail is described in the dissertation document:

- Causal Visibility (CV)
- Causal Consistency (CC)
- Transactional Causal Consistency (TCC)
- Read Committed (RC)
- Repeatable Read (RR)
- Linearizability (LIN)
- Serializability (SER)

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 Section 4.

3.1. 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 Microservices 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 be about maintaining two different versions of the same entity, but making 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 out initial idea to a FaaS system with only one 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 [13] in order to be able to apply it to a FaaS system instead of a Microservices' one.

However, when studying the complexity metric in [13], 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[13], 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, supposing 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 [13] 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. 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 build 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.

3.2. Changes Made to CLOTHO

Consistency Models CLOTHO, by default, assumes an execution environment where no isolation or atomicity guarantees are given. The seven consistency guarantees discussed in Section 3 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.

Distinguishing Labels for Transactions As mentioned in Subsection 3.1, 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.

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 same transaction.

After analyzing the code of the CLOTHO framework, we found a different but similar relation, the sibling relation, that related two operations if they belonged to the same transaction instance.

The two are different: st is more general, since it is true for all operations for which the sibling relation is true, but it is also true for operations for which the 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.

4. Evaluation

4.1. Validating the Implemented Consistency Models

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

4.1.1 Dirty Write

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

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

Listing 1: Dirty Write class abstraction as used in CLOTHO

4.1.2 Dirty Read

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

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

Listing 2: Dirty Read class abstraction as used in CLOTHO

4.1.3 Long Fork

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

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

Listing 3: Long Fork class abstraction as used in CLOTHO

4.1.4 Lost Update

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

Listing 4: Lost Update class abstraction as used in CLOTHO

4.1.5 Write Skew

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

Listing 5: Write Skew class abstraction as used in CLOTHO

The five benchmarks were tested under the different consistency criteria. Table 1 assembles the expected values for the number of anomalies of each benchmark under a certain consistency model, according to academical research.

	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 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

Table 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

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

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. Validating the Novel Complexity Metric

Mono2micro[13] 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:

The intention of this test instance is to mimic the following scenario: the microservice system managing this application has 3 different microservices: the first is responsible for handling domain entity A, the second for domain entity B, and the third service is responsible for checking coherence between these two domain entities. Each service executes, respectively, the first, second and third transactions in Listing 6. 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 were used, as presented in [13]:

$$complexity(d) = \frac{\sum_{f \in F} complexity(f, d)}{\#F} \quad (1)$$

Equation 1: Complexity of a microservice decomposition: the complexity of a decomposition is given by the average of the complexities of its functionalities

$$complexity(f, d) = \sum_{c \in C} complexity(c, f, d) \quad (2)$$

Equation 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

$$complexity(c, f, d) = \# \cup_{a \in c} complexity(a, f, d) \tag{3}$$

Equation 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

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 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 4.2 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 zero. 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, $check\_vars$. This, 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 in Table 3.

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

Table 3: Observed number of serializability anomalies when executing our test program in CLOTHO under each consistency model

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 1 and 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[13], 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 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.

5. 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 task 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.

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 this architectural type.

References

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transac-

tions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, Nov. 2013.

[3] 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*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. Association for Computing Machinery.

[4] 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*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] S. Gilbert and N. Lynch. Perspectives on the CAP theorem. *Computer*, 45(2):30–36, 2012.

[6] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani. *A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures*, pages 41–52. Springer-Verlag, Berlin, Heidelberg, 2009.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[8] J. Lechtenbörger. *Two-Phase Commit Protocol*, pages 3209–3213. Springer US, Boston, MA, 2009.

[9] 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)*, pages 135–150, Savannah, GA, Nov. 2016. USENIX Association.

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

[11] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan. CLOTHO: Directed test generation for weakly consistent database systems. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[12] C. Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018.

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

[14] 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*, Middleware '18, pages 120–133, New York, NY, USA, 2018. Association for Computing Machinery.

[15] G. Vossen. *ACID Properties*, pages 19–21. Springer US, Boston, MA, 2009.