NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

HUGO MIGUEL GRILO RODRIGUES
BSc in Computer Science

CAUSAL CONSISTENCY VERIFICATION IN RESTFUL SYSTEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
January, 2023

DEPARTMENT OF
COMPUTER SCIENCE

CAUSAL CONSISTENCY VERIFICATION IN RESTFUL SYSTEMS

HUGO MIGUEL GRILO RODRIGUES
BSc in Computer Science

Adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, NOVA School of Science and Technology

Co-adviser: Filipe Bastos de Freitas
Adjunct Professor, ISEL

Examination Committee

Chair: António Maria Lobo César Alarcão Ravara
Assistant Professor, NOVA School of Science and Technology

Adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, NOVA School of Science and Technology

Member: Paulo Sérgio Almeida
Assistant Professor, University of Minho

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
January, 2023

Causal Consistency Verification in Restful Systems

Copyright © Hugo Miguel Grilo Rodrigues, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.2) [45].

To my mother.

Acknowledgements

Finishing this dissertation concludes a five-year journey that consisted of challenges, learning and joy, but also uncertainties and fear.

I would like to thank my advisors, Prof. Nuno Preguiça and Prof. Filipe Freitas, for guiding me since day one and providing all your knowledge and resources. I really appreciate the patience and motivation they have given me throughout this whole journey. I would also like to thank Prof. Carla Ferreira and Prof. João Leitão for their participation in several meetings, as their opinions and pieces of advice leveraged the development of this project. This work was performed in the context of FCT/MCTES project PTDC/CCI-INF/32081/2017.

Additionally, I would like to thank our institution, NOVA School of Science and Technology, for all the knowledge, opportunities and for preparing me for the future.

Also, I need to thank my family for all the love, support and comprehension. I would not be the person I am today without both my maternal grandparents, my sister and my uncle, who are the pillars of my life.

Finally, I dedicate this work to my mother, who is the brightest star in the sky.

"Act as if what you do makes a difference. It does."
(William James)

Abstract

Replicated systems cannot maintain both availability and (strong) consistency when exposed to network partitions. Strong consistency requires every read to return the last written value, which can lead clients to experience high latency or even timeout errors. Replicated applications usually rely on weak consistency, since clients can perform operations contacting a single replica, leading to decreased latency and increased availability.

Causal consistency is a weak consistency model, however, it is the strongest one for highly available systems. Many applications are switching to this particular consistency model, since it ensures users never observe data items before they observe the ones that influenced their creation.

Verifying if applications satisfy the consistency they claim to provide is no easy task. In this dissertation, we propose an algorithm to verify causal consistency in RESTful applications. Our approach adopts a black box testing, where multiple concurrent clients execute operations in a service and records the log of interactions. This log of interactions is then processed to verify if the results respect causal consistency. The key challenge is to infer causal dependencies among operations executed in different clients without adding any additional metadata to the data maintained by the service. When considering a particular operation, the algorithm builds a new dependency graph that considers one of the possible justifications the operation might have, but if this justification results in failure further ahead in the processing, it is necessary to build another graph considering another justification of that same operation. The algorithm relies on recursion in order to achieve this backtracking behaviour. If the algorithm is able to build a graph containing every operation present in the log, where the chosen justifications remain valid until the end of the processing, it outputs that the execution corresponding to that log satisfies causal consistency. The evaluation confirms that the algorithm is able to detect violations when feeding either small or large logs representing executions of RESTful applications that do not satisfy causal consistency.

Keywords: Distributed Systems, RESTful Applications, Causal Consistency, Vector Clocks, Jepsen, JepREST

Resumo

Os sistemas replicados não podem manter a disponibilidade e a consistência (forte) quando expostos a partições de rede. A consistência forte exige que cada leitura retorne o último valor escrito, o que pode levar os clientes a experienciar alta latência ou até mesmo erros de tempo limite. As aplicações replicados geralmente usam consistência fraca, pois os clientes podem realizar operações contactando uma única réplica, levando a latências baixas e maior disponibilidade.

A consistência causal é um modelo de consistência fraco, mas é o mais forte para sistemas altamente disponíveis. Muitas aplicações usam este modelo, pois garante que os clientes nunca observem dados antes de observar os que influenciaram a sua criação.

Verificar se as aplicações satisfazem a consistência que alegam fornecer não é fácil. Nesta dissertação, propomos um algoritmo para verificar a consistência causal em aplicações RESTful. A nossa abordagem adota um teste de caixa negra, onde vários clientes concurrentes executam operações num serviço, onde as interações são documentadas num ficheiro. Este ficheiro é processado para verificar se os resultados respeitam a consistência causal. O principal desafio é inferir as dependências causais entre as operações executadas em diferentes clientes sem adicionar metadados adicionais aos dados mantidos pelo serviço. Ao considerar uma determinada operação, o algoritmo constrói um novo grafo de dependências que considera uma das possíveis justificações que a operação possa ter, mas se esta justificação resultar em erro mais tarde no processamento, é necessário construir outro grafo considerando outra justificação dessa mesma operação. O algoritmo é recursivo de modo a alcançar esse comportamento de retrocesso. Se o algoritmo conseguir construir um grafo que contém todas as operações presentes no ficheiro, onde as justificações escolhidas permanecem válidas até o final do processamento, indica que a execução correspondente a este ficheiro satisfaz a consistência causal. A avaliação confirma que o algoritmo é capaz de detectar violações ao fornecer ficheiros pequenos ou grandes representando execuções de aplicações RESTful que não satisfazem a consistência causal.

Palavras-chave: Sistemas Distribuídos, Aplicações RESTful, Consistência Causal, Relógios Vetoriais, Jepsen, JepREST

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

This chapter starts by introducing the context of this dissertation, its motivations, the problem it tries to solve. Finally, the proposed solution is presented and the structure of the document is outlined.

1.1 Context

In recent years, developers have relied on developing web and mobile distributed applications to provide services to their clients. A distributed application consists of a set of components connected through a network, which perform computations and communicate with each other to achieve that goal.

Online services are distributed applications used across the world through the Internet. These are usually replicated for providing both dependability and good performance [24], specially for low latency access. Cloud plays a big part in deploying these systems, since it is all about redundancy and fault-tolerance, where it is crucial not to affect the availability of an entire system when some of its components fail [35].

According to the CAP theorem [17], detailed in section 2.3.1, a replicated distributed system cannot hold both availability and (strong) consistency in environments subject to network partitions. As it is impossible to preclude network partitions in a large scale distributed system, a system must choose between availability and consistency, given its requirements.

Stronger consistency models, e.g, linearizability, cause high latency due to blocking cross-replica synchronisation [29], which guarantees that most of clusters hold the most recent updates. Weak consistency provides low latency due to its high availability, since operations can execute contacting a single replica, where there is little synchronisation among replicas [61].

Developers tend to rely on weaker forms of consistency, because they want to provide services with as low latency as possible. It is common for replicated systems to offer weak consistency, since it not feasible to update every single replica spread across the globe before returning to the client. Regarding weak consistency models, eventual consistency is the weakest of them all, whereas causal consistency is the strongest one that is available in the presence of partitions [47].

1.2 Motivation

The components of a system can fail due to software errors and server wrong configuration, and can be exposed to failures, such as server crashes and network partitions [23]. In fact, an issue can imply another, e.g., a software bug can lead to a server crash. If the system does not handle these faults, they lead to consistency anomalies in services or, even worse, catastrophic situations like service unavailability. Detecting these anomalies is not easy, and many times, consistency anomalies are only detected in production after some time. Some of them can be detected using a unique server to probe the service issuing a simple sequence of read and write operations, but in other cases it is necessary to interact with several servers to probe the service.

Designing and developing a distributed system is complex, due to concurrency and the possibility of partial system failures, where some components continue working correctly while others have failed. The greater the complexity of a distributed system, the greater the chance of not handling situations that may end up violating the correctness of the system.

Replicated systems not always provide the consistency guarantees they claim to provide. This fact is supported by Jepsen [36], which has detected consistency issues in more than two dozens of distributed systems, such as: distributed databases, queues, and consensus systems.

Thus, it is necessary to verify and test distributed systems that are replicated and can provide different consistencies.

1.3 Problem

There are solutions that verify the specification of systems using static analysis, which is important. Moreover, it is also crucial to verify their implementation and execution in concurrent environments.

As mentioned earlier, distributed applications can adopt different consistencies. This dissertation tries to address the challenge of verifying whether executions of replicated RESTful applications, following the *CRUD* pattern [64], respect causal consistency, whose concept is detailed in section 2.3.5.

It is not obvious that verifying causality is simpler than verifying linearizability. From an abstract point of view, we can verify linearizability by generating every order of operations performed on a system, and for each order, checking whether it satisfies the results of the operations [40]. At the end, there will be at most one order that satisfies linearizability. As for causality, there may be more than one possible order for the sequence of operations, since every replica has its own order. Different replicas can return different values, however, as long as the operations do not share a causal relationship, we are dealing with a valid configuration of the system. Therefore, there may be an exponential number of configurations of the given system.

Verifying causality in RESTful applications seems to be more complex than verifying it in databases. Let us consider a RESTful application that manages students, where each one is identified by an *id*. Write operations of this application require an additional implicit read before performing the actual write, e.g., an operation that updates a student (PUT) requires that the student with the given *id* exists. In databases, this does not happen, the write is performed without any additional verification. In this particular case, even if the student with the given *id* did not exist in the database, the update would create an entity with that *id*.

1.4 Proposed Solution

In this dissertation, we propose an algorithm that analyses histories of *CRUD* operations corresponding to executions of RESTful applications, classifying them as causal or not, that is, whether or not they satisfy causal consistency. Besides, this algorithm can also detect some violation patterns of a variation of the causal consistency model - causal+ consistency [43] - for the same histories of operations.

The histories of operations fed to the algorithm follow the pattern of histories generated by Jepsen [36] and JepREST [58] after performing a workload of operations over a RESTful application. Every single REST operation in the history file is divided into two points in time: request and response. This means that the operation can take place at any point between those two moments. The request part of an operation contains all the metadata of the request that the client submitted to the application and the same happens with the response part, that is, it contains all the response's metadata retrieved by the application.

The algorithm processes the history file, which converts those metadata structures into one of three types of (processed) operations: write, implicit read and explicit read. A (processed) write corresponds to the actual value writing, i.e., assigns that value to the given entity identifier. An implicit read indicates whether an entity, identified by some field (e.g., an *id*), must exist or not for the corresponding write to take place. If a write operation comprises a POST method, then the implicit read holds the information that the given entity must not exist. However, if we were dealing with a PUT or DELETE methods, the implicit read stores the information that the given must exist. Finally, regarding explicit reads, they correspond to read operations (GET method), storing the entity identifier and the value read.

The vector clock [44] structure is the structure that allows this algorithm to work. Every value that has been either written or read is assigned a vector clock that indicates its version. Every client stores a vector clock, representing its state, and more complex structures that also rely on vector clocks.

The algorithm follows a recursive approach, which goes through each of those processed operations, following the order of their original metadata elements. In the case of implicit or explicit reads, the algorithm tries to justify them either by the corresponding client's knowledge or writes from other clients. In order to get the writes of other clients that might justify the operation of the given client, vector clocks checks must be performed.

If the algorithm is handling an operation that has no justifications or if all justifications of a given operation resulted in failure further ahead, the algorithm backtracks. Once the algorithm backtracks all the way to the first operation whose justifications did not work further ahead, then it classifies the execution corresponding to the history of operations as not causal consistent. If the algorithm was able to create a graph where every justification chosen resulted in success further ahead, i.e., the algorithm was able to treat every operation, then it classifies the history of operations as causal. The graph represents a possible configuration of the RESTful application when considering the execution corresponding to the history of operations.

1.5 Contributions

The project developed contributes to the extension of the JepREST [58] tool. This tool tests and analyses RESTful applications in order to verify whether or not they respect linearizability. Our project took advantage of JepREST and its semantics to verify whether or not replicated RESTful applications satisfy causal consistency. The contributions of this dissertation are the following:

- A detailed definition and explanation of the algorithm that verifies causal consistency in RESTful applications.
- A processing mechanism that converts the data in the history of operations into the data the algorithm consumes.
- An overview of the JepREST's implementation and a detailed explanation of implementation of our algorithm.
- An in-depth experimental evaluation of a prototype of the algorithm that follows the implementation semantics. Its behaviour was assessed through histories written by us and others that have been generated by JepREST after performing workloads on a real RESTful application. The scalability of this prototype was also measured.

1.6 Document Structure

The skeleton of the rest of the document is the following:

- **Chapter 2 - Related Work:** Starts by discussing approaches proposed to test applications and proceeds with a presentation of the specific verification techniques. Then, it introduces consistency models for distributed applications followed by techniques proposed to verify such consistency guarantees. Then, it presents an overview of tools used for testing distributed applications. Finally, it provides a brief conclusion of the chapter.

- **Chapter 3 - Solution Design:** Compares the difficulties of verifying causal consistency between databases and RESTful applications executions and presents in detail the behaviour of the algorithm that verifies causal consistency in RESTful applications. The algorithm's characteristics and the way it processes and manages data are explained through several examples of histories of operations. At the end, its pseudocode is presented.
- **Chapter 4 - Implementation:** Briefly presents the implementation of JepREST's components and describes in detail the implementation of our algorithm.
- **Chapter 5 - Experiments and Results:** Features an in-depth assessment of a prototype of the algorithm that checks the intermediate and final results when feeding histories of operations representing custom executions and executions of a real RESTful application. At the end, it presents the scalability of the prototype.
- **Chapter 6 - Conclusion:** Summarizes what was accomplished in this dissertation and discusses future work.

2 Related Work

This chapter introduces work related with the topic of this dissertation. We start by discussing approaches proposed to test applications and proceed with a presentation of the specific verification techniques. As the goal of this dissertation is to test distributed applications, we then introduce consistency models for distributed applications followed by techniques proposed to verify such consistency guarantees. Then, we present an overview of tools used for testing distributed applications. Finally, we provide a brief conclusion of the chapter.

2.1 Verification of Correctness

The challenge of verifying the correctness of systems is complex. Correctness is based on properties that are either maintained (e.g., system invariants or safety properties) or established during execution (e.g., liveness properties) [1]. This section presents high-level techniques for verifying the correctness of a system.

2.1.1 Knowledge of the Application

Depending on the tester's knowledge of the system, there are two types of tests that can be performed: white-box and black-box.

2.1.1.1 White-Box Testing

White-box testing [52] is often used for verification (i.e., are we building the software right?), where software testers have access to the code, internal structure and design of the system subject to test. This technique is concerned with the internal mechanism of the system, so tests are designed based on the information derived from the source code and internal design, as they mainly focus on control and data flows of a program.

Most often, this approach is adopted only by the developers who built the application because they know how the code is structured, what it is supposed to do and how the components relate between themselves. Some white-box techniques are presented below. However, one will not resort to these techniques. Usually, they reveal to be very inefficient since one would need to analyse every piece of code of each component, which would be a slow and costly process.

Branch Testing [38] This technique aims to write test cases given that each possible possible outcome from conditions, i.e., either true of false outputs from *IF-THEN-ELSE* statements, must be tested at least once. This is done on every control statement, which also includes situations where a decision depends upon previous decisions.

Basis Path Testing [38] Software testers start by drawing an appropriate control flow graph from the source code, which demonstrates the sequence of the different instructions executed. Then, it is necessary to calculate the cyclomatic complexity of it, which defines the number of independent paths and indicates an upper bound for the number of tests required to coverage all program statements. A graph path is considered independent if introduces at least one new set of processing statements or a new condition, which is basically a path with an edge that is not present in any other path. Finally, test cases are designed to force the execution of these independent paths. When executing all test cases, the software tester ensures that all statements in the program have been executed at least once. Therefore, it has the advantage of detecting and reducing redundant tests.

Data Flow Testing [38] This approach looks at how data moves throughout the program and tries to understand how the values assigned to variables can affect its execution. A control flow graph is designed in order to observe how the program variables are defined and used [26].

Tests are designed in a way to pick paths which assure that every data object has been initialized prior to its use, and that all defined objects have been used at least once.

2.1.1.2 Black-Box Testing

Contrary to white-box testing, black-box testing [52] is a testing strategy for examining a system's behaviour, where tests are designed exclusively according to the system's specifications, and so this strategy is often used for validation (i.e., are we building the right software?).

In this strategy, given a system that one wants to test, software testers have no access to its source code, neither any knowledge about its internal structure. Software testers are only aware about the possible inputs that the system is able to consume and the outputs that it should produce given those inputs. Some black-box techniques are presented below:

Random Testing [3] This technique is one the most fundamental and most popular testing method, as well as one of the easiest to implement. In this strategy, the domain input is partitioned across multiple groups (partitions), where each partition holds a single piece of the domain input. Also, what defines the inputs of test is the content of its assigned partition. what each partition holds is what the inputs to a given test will be.

The assignment of partitions to each test is done randomly, usually based on a uniform distribution or according to the operational profile [19].

If the specifications of the system are incomplete, then this may be the only feasible technique that one could rely. Otherwise, it is advantageous to use it as the system would be subject to repeated tests.

Adaptive Random Testing [3] The development of this approach came due to the lack of failure detecting when relying on pure Random Testing over a system. This is because the chances of hitting these failure pattern, i.e., selecting failure-causing inputs as test cases, depends only on the magnitude of the failure rate [19].

Empirical studies have shown that failure-causing inputs tend to form contiguous failure regions. Consequently, non-failure-causing should form continuous non-failure region. Given this, it is important that new test cases should be far away from already executed non-failure-causing test cases. The idea of this technique is to evenly spread test cases across the input domain. Forcing this to happen will enhance the failure detection effectiveness of Random Testing. This makes developers to have more knowledge about the system, since the portions of input domain that were missing in Random Testing will be now taken into account.

2.1.2 Application Running

Depending if we need the system to be running or not, we can analyse it in two ways: static and dynamic.

2.1.2.1 Static

Static analysis [32] consists in analysing software without having it up and running. This analysis is mostly performed on some version of the source code. Usually, this term is applied to the analysis performed by an automated software tool, but there also needs to be some human analysis over the code. Theoretically, static analysis tools can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be difficult [20].

The fact that this approach is independent of code execution, if it proves that the system satisfies a particular property, then every code execution will also satisfy that property.

Even though reviewing an application's code can be done in any phase of the software development, the best option is to do it at an early stage, because detecting and correcting bugs and vulnerabilities late in the software development process can be quite risky and costly.

Tools that statically operate over a system can produce two types of conclusions: false negatives and false positives. False negatives happen when the program contains bugs which the tool does not report, and false positives happen when the tool reports bugs that the program does not contain. While false positives may lead to a long time process until the developer realizes that there is no error after all, false negatives are much more dangerous because they lead to a false sense of security. To this end, a good tool for static analysis is one that, although sometimes shows a false positive, never lets a false negative pass [20].

Tools that statically analyse systems with a single component are precise and efficient. However, that is not case in the context of a large scale system as it would be necessary for developers to write all specifications of the system. This writing process is very complex and time consuming as developers would need to reason about every possible state. Consequently, it is very likely that some possible states are missing, resulting in incomplete specifications about the system, so the correctness of the system is not guaranteed.

Even if one had the complete specifications of the multiple components of a system, with no false positives and, specially, no false negatives, it is not sufficient to do a static analysis over those multiple components to ensure the correctness of global system, since the combination of multiple correct units does not guarantee the correctness of the global environment, as will be seen with more detail in section 2.2.2.

2.1.2.2 Dynamic

Contrary to the static analysis, dynamic analysis [32] is performed by executing programs on a real or virtual processor. Having the system up and running, it is subject to a set of tests that try to verify if properties are satisfied by the system. In order to boost the confidence on the system's correctness, it is necessary that these tests should verify as much properties of the system as possible. This is specially important since one can identify possible inconsistencies of the given results, and eventually resolve them.

This type of analysis seems to be more efficient than the static analysis as it does not include the downsides that static analysis introduce. Still, the disadvantage of dynamic analysis is that the results produced are not generalized for future executions, and although dynamic analysis checks the functional properties of a system's software, static analysis can decrease the amount of testing and debugging necessary for the software to be deemed ready.

2.1.3 Properties of the Application to Analyse

Properties of a system can be either functional or non-functional. The process of analysing each type is different.

2.1.3.1 Functional

One must assert that the system's functional properties, such as behaviour, outputs and flows, are maintained and established during every execution of it.

Although, a distributed system is a rather complex application, which makes its behaviour analysis a quite complex process. For that case, this task is often divided into 3 different steps:

Unit testing [52] The first step is to design and execute unit tests, which aim to verify each unit/component correctness, in isolation, for the sake of asserting that each component provides what it is supposed to provide. Usually, developers rely on this approach for testing small units of code.

Given a complex distributed system, one must test each component separately. To this end, given a particular system's component, a set of unit tests that represent several scenarios is defined to test multiple combinations of inputs that result not only in success, but also in failure. Also, these tests can simulate concurrent based scenarios, which are very common and sometimes very hard to handle in distributed systems.

However, there are many situations where one cannot test a component in isolation, as it has dependencies on other component(s). To solve this, unit tests take advantage of mock-ups to prove intrasystem dependencies and verify interactions of various components [50]. Mocking-up a component is essentially replacing it with an object that simulates the execution of the real component. A very common example is when one wants to test a data repository, which is database dependent. This situation requires mocking the database into an object that does simpler computation and storage processes.

Usually, specially in industries, this type of testing is executed under a dynamic environment, but it can also be performed using techniques that verify the system in a static way.

Integration testing [52] The next step is designing and executing integration tests, which aim to validate that two or more units/components work together properly. This step is considered more complex than the previous one as many more scenarios need to be considered. However, these tests are much closer to the reality of a distributed system, because components communicate with real components, not mock-ups. Besides, integration tests can reuse these same tests without the mock-ups to verify that they run correctly in an actual environment [50]. This way, one can check if the components' functionalities are provided in all combinations of interaction among the integrated components.

As in unit testing, this approach is not only supported by dynamic techniques through the execution of tests, but also supported by static analysis techniques. However, resort to static techniques may not be the best option because new specifications need to be created, since more possible states of the system are taken into account.

Functional testing [52] The last step is designing and executing functional tests to verify every system's components when these interact with each other. In other words, this step is performed over the final system, where one checks if components provide their functionalities when they communicate with each other.

When writing functional tests, one must take into account as much scenarios as we can think of that result in interactions between the system's components, not only when no failures occur but also when in their presence. The complexity of this step is higher when compared to the previous one because all interactions between all components of the system are considered, which results in the introduction of even more scenarios.

As in both previous types of testing, functional testing is supported by both types of analysis techniques: dynamic and static.

2.1.3.2 Non-Functional

Validating software actions is not enough. During the execution of a system, it is necessary to assess its non-functional properties, which are constraints on the manner in which the system implements and delivers its functionality [18].

Non-functional properties are equally important as the functional ones, since they affect client satisfaction and experience [30]. To evaluate these properties, a sequence of experiments is performed to resolve meaningful discrete values and patterns. Typically, the most interesting non-functional properties about a distributed system are:

Latency It is essentially the time interval between sending a request and receiving the respective response. In other words, given a pair of a client and an application server, latency is defined by the time for the request to travel from the client to the application server, plus the time that the application server takes to execute the request and generate a response, plus the time of the response to travel from the application server back to the client. To assess this property, it is conducted a sequence of experiences, increasing the number of clients between each experience until the latency starts to increase considerably.

Performance Verifying the performance of a system implies to determine how many requests an application server can execute per unit of time. A very common technique for doing this assessment is called load testing [37]. *Load* refers to the rate of the incoming requests to the system. Briefly speaking, this technique refers to the practice of accessing the system behaviour under load [13]. In practice, a sequence of experiences is performed, increasing the number of clients between each experience until the number of requests processed by the server stops increasing.

Scalability Verifying the scalability of a system implies to determine the increase in the number of requests the system can execute when the number of servers increases as well. In practice, a sequence of experiences is executed, where one increases the number of clients between each experience until the number requests processed by the server(s) stops increasing. Each experience must be repeated increasing the number of servers in order to resolve the value described in the beginning.

It is evident that this approach, contrary to the functional one, can only be performed using dynamic techniques, since measuring those metrics described above requires the system to be up and running.

2.2 Verification Techniques

This section presents low-level techniques for verifying a system, which can include properties from multiple high-level techniques seen in section 2.1.

2.2.1 Model Checking

Model checking [21, 50] is a formal verification technique that determines if a given system is provably correct using state-space exploration systematically in order to enumerate paths for a given system. This technique is performed without having the system up and running and its execution is shown in Figure 2.1.

Given a tool that is based on model checking, it receives a model, which represents the system's requirements or design, and a property that the system is expected to satisfy, called specification. Once the given model satisfies the given specification, the tool outputs a *yes*, otherwise it generates a counterexample that violates the specification. By ensuring that the model satisfies enough system properties, our confidence regarding the correctness of the model increases [2].

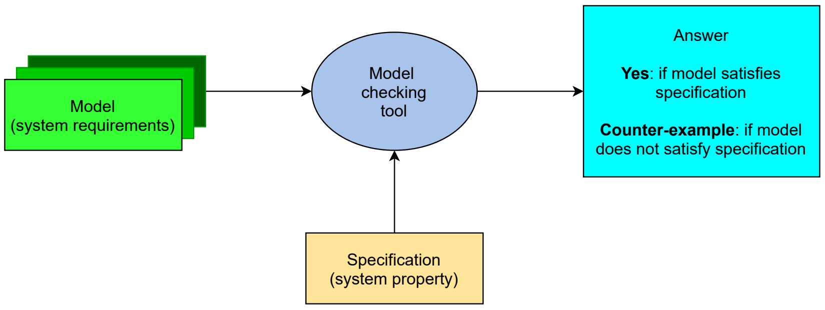
Model
(system requirements)
Model
checking
tool
Answer
Yes: if model satisfies specification
Counter-example: if model does not satisfy specification
Specification
(system property)

Figure 2.1: Functioning of the model checking technique

Supposing that the system has a finite number of states, then this exhaustive analysis can be performed. However, the system in test is usually complex, such as real-life industrial systems. The combinations of inputs, states and failure modes the system can experience cause running a much more time and resource consuming analysis, since the model checker may not finish the verification task.

An approach for this inconvenience is based on restricting the number of states that can be explored by a given threshold. Although, incomplete models will be generated, meaning that eventual bugs over that threshold of states will not be detected [63].

2.2.2 Composition

Composition is a technique that systematically assembles a system from subsystems and components, where tools aim to systematically assembly behaviour models for complex systems from behaviour models for simpler systems and components [10].

The idea behind this technique is based on having provably correct components composed with one another to create provably correct systems. However, this is not the case [50]. Usually, components are tested under different failure modules, and consequently these do not compose. This implies creating a new correctness specification and rerunning the tests to prove that the resulting system is provably correct.

Nevertheless, one would need to create a specification for each combination of components, which does not scale well, specially in microservice based architectures that have recently become popular. A system that relies on a microservice architecture consists of many (from tens to thousands) distinct services, which makes writing correctness specification at this scale not reasonable.

2.2.3 Monitoring

Monitoring [51] is a strategy that gathers metrics from several services or machines that are up and running. These metrics are subject to processing and aggregating, resulting in more valuable metrics, specially regarding their percentiles, which can be visualized into easy to understand dashboards, e.g., plots and histograms, so the user can have a more intuitive idea about how the system has behaved over time. Useful alerts based on these metrics can be set, and are often intended to inform users about an unusual or disruptive event (e.g., peak load) that occurred within the system, which can cause serious problems.

The default or most used metrics if one wants to test a system over time are: service is up or down, requests per second, number of successful and failed requests, cpu load, memory usage, garbage collection, heap size, among others.

When it comes to running any successful service and debugging failures, monitoring the system and detecting errors through it is extremely important. However, this is considered a wholly reactive approach for validating distributed systems, as bugs can be found only once the code has made it into production and thus, clients are affected. Monitoring tools provide visibility into how the system is currently behaving versus how it has behaved in the past. Therefore, monitoring allows us only to observe and should not be the sole means of verifying a distributed system [50].

2.2.4 Canaries

Canarying [50] is another verification technique that requires the application to be up and running and the way it works is shown in Figure 2.2.

Supposing the situation that we have a production cluster running code that is "stable" or "correct" (primary), as far as we know, and came up with a new version of the code (candidate), which we think it will optimize some operations, we want to verify the correctness of this new code. Instead of replacing all nodes (physical or virtual machines) with the candidate code, this approach uses a deployment pattern in which the candidate code is introduced into production clusters, where a few nodes are upgraded with it. The output and metrics from the candidate nodes compared with the primary nodes are the sources of information that one must take into account to decide if more nodes can be upgraded or not. If the candidate nodes prove to behave either equivalent or better than the primary ones, then more nodes are upgraded with the candidate code. Otherwise, if there are significant differences on the behaviour or failures are introduced, canary nodes are rolled back to run the primary code.

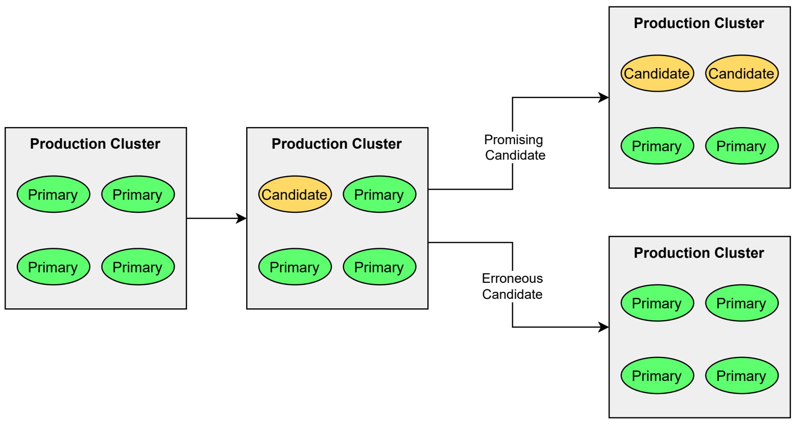

Figure 2.2: Functioning of the *canarying* technique

Even though this technique limits the risk of deploying new code to live clusters, the guarantees it can provide are limited. The only guarantee we have if a canary test passes is that the candidate code performs at least as well as the primary code at a given moment in time. Besides, if the system is not under peak load or a network partition does not occur during the canary test, then no information is obtained regarding the performance of both types of node given these scenarios and thus, we do not know which one handles these situations better.

While canary tests are indeed valuable if one wants to validate whether the candidate code behaves as expected in the common case, it is not enough to verify the system's correctness, specially fault tolerance and redundancy.

2.2.5 Fault-injection

Testing an application through fault-injection [50] consists in causing or introducing faults in the system. These faults can be dropped messages, network partitions, or even the loss of an entire data center. By forcing the injection of faults, engineers are able to observe and measure how the system under test behaves. If no failures are simulated using tools that rely on fault-injection testing, it is not guaranteed that the system is correct because the entire space of failures has not been exercised. Unfortunately, the task of searching the whole space of distinct fault combinations that an infrastructure can test is intractable [1].

This testing technique reveals to be very powerful and useful. A fault-injection test does not necessarily need to be too complex to achieve interesting conclusions about a system. An example that supports this fact happened in October 2014, where Stripe detected a bug in Redis. The basic fault-injection test of running "kill -9" on the primary node of the Redis cluster was enough to cause the loss of all data in that cluster [50].

Fault-injection is becoming very popular and is an area of ongoing research. One of the most interesting current topics regarding this research is *lineage-driven fault injection* [50]. A lineage-driven fault injector, instead of exhaustively exploring the whole failure space as a model checker would, reasons about successful outcomes and what types of failure could occur that would change these. This is extremely useful since it significantly reduces the state space of failures that must be tested to prove the system's correctness.

2.2.6 Chaos Engineering

Pioneered by Netflix, chaos engineering [12, 56] is a discipline that performs a set of experiments to uncover weaknesses of a complex distributed system, which might be compromising its availability and security. In other words, it is based on experimenting a system to build confidence in the system's capability (resiliency) to withstand turbulent conditions in production [54].

A key aspect of chaos engineering, as slightly mention in its definition, is that it relies only on experimentation, rather than testing. Both concepts fall under the *quality assurance* environment, however, they try to obtain different information about a system.

Testing requires that an engineer writing the test(s) knows particular properties about the system that one is looking for. However, humans are not capable of understanding every potential side effect from all interactions of a complex system's components. Running a test holding an assertion, based on existing knowledge, will judge that assertion, usually into either true or false. This means that tests are simply statements about known properties of the system, which do not create new knowledge. While experimentation is based in an exploration of the unknown. It starts by introducing an hypothesis about the system, and if it is refuted, then a new property is discovered, otherwise confidence grows in that hypothesis. Thus, experimentation creates new knowledge, which is the focus of chaos based methods.

In order to properly apply chaos engineering experiments, the experimentation process is based in the following principles [56, 54]:

Build a hypothesis around steady state behaviour: It focus on the way the system is expected to behave, and captures that in a measurement. Interesting metrics that represent this steady state behaviour can be: the overall system's throughput, error rates, latency percentiles, among others. In the context of Netflix services, one of the metrics that they use is based on how many users start streaming a video each second, called *SPS*, which stands for stream starts per second [12]. Netflix engineers use *SPS* as their primary indicator of the overall health of the system. Even though the term "chaos" refers to a sense of unpredictability, one of the key assumptions of chaos engineering is that complex systems exhibit behaviours that are regular enough that they can be predicted. *SPS* is also a very good example of a metric that characterizes the steady state behaviour.

Thus, these systemic behaviour patterns during experiments allow chaos to verify that the system does work, rather than trying to validate how it works.

Vary real-world event: The variables in experiments should reflect real-world events, where they must be prioritized by potential impact and estimated frequency. One should consider events that correspond to failures on hardware (e.g., servers dying), on software (e.g., malformed responses), and also non-failure events (e.g., a spike in traffic) as well. Regarding Netflix services, some examples of inputs that they use in their experiments are [12]: terminate virtual machine instances, inject latency into requests between services, fail an internal service, make an entire Amazon region unavailable, etc.

There are some cases that one needs to simulate that an event occurred instead of simply inject it. For example, Netflix does not actually take an entire Amazon region offline. Instead, they generate actions that try to simulate the occurrence, i.e., client requests are redirected to other Amazon regions, and effects are carefully analysed.

These aspects indicate that any event capable of disrupting the steady state behaviour of a system is a good candidate variable for a chaos experiment.

Run experiments in production: If one is executing experiments on a given environment, then confidence is built in that environment. Usually, a system behaves differently depending on the environment and traffic patterns. Given that the behaviour of utilization can change any time, often in ways that humans cannot predict, experimenting must take place in Production, since sampling real traffic is the only way to reliably capture the request path. More precisely, given a test context (e.g., Staging), it is never possible to fully reproduce all aspects of the system within that context, because there will always be significant differences such as how synthetic clients behave compared to real clients, or DNS configuration issues [12].

Chaos strongly prefers to experiment directly on Production, but there are situations that make sense to start on Staging, and gradually move over to Production.

Automate experiments to run continuously: This principle focuses on the practical implications of working on complex distributed systems. First, it is crucial that a larger set of experiments must be covered compared to what humans can cover manually. In fact, the conditions that could possibly contribute to incidents are so many that they cannot even be planned for. Thus, running experiments manually is labor-intensive and unsustainable. Second, complex systems changes continuously over time. Engineering teams constantly alter the behaviour of existing services, add new services, and change runtime configuration parameters, where any of these modifications can potentially contribute to a new vulnerability, e.g., service interruption [12].

To resolve the first issue, automation provides a means of scale out the search in the solution space of potential vulnerabilities for the ones that could contribute to undesirable systemic outcomes. Regarding the second issue, it is necessary to leverage automation so that one can maintain confidence in results over time [12], since complex systems will change.

Minimize blast radius: As mentioned in the third principle, chaos experiments are executed in Production. If there is not any orchestrated control over these experiments, they have the potential to cause unnecessary effects on the client side. In fact, there must be an allowance for some short-term negative impact because of the unavoidable turbulent conditions. Indeed, it is necessary to reduce the risk to Production traffic by engineering safer ways to run experiments, where the fallout from experiments must be minimized and contained.

Applying these principles over a chaos engineering experiment is extremely important because not only the confidence in the system's increases (hypothesis proves to be true), but also new knowledge about the system is obtained (given a refuted hypothesis). This new knowledge is the new starting point to generate new hypothesis, which is beneficial because confidence is strongly correlated to the number of hypothesis that reveal to be true. Thus, performing chaos engineering experiments that follow the principles above allow engineering teams to innovate quickly at massive scales [54].

2.3 Consistency Models

The term *consistency* refers to the rules, properties and guarantees that a system holds regarding data management (storage and replication). Once performing some *write* operations, the consistency is what dictates the characteristics of the possible outcomes for a given *read* operation.

For a large scale back-end infrastructure to maintain both dependability and good performance, it usually takes advantage of geo-replication [24]. Dependability obligates the system to hold multiple replicas with replicated data, since systems are exposed to catastrophic failures, which need to be tolerated. As for performance gains, they come from the fact that clients are redirected to nearby copies of the data they want to access [29]. Depending on how the data is stored and replicated among the replicas is what will tell which consistency type one is dealing with, i.e., the guarantees for operations upon the system.

This section presents the motivation for having two high-level types of consistency, where each one's overall specification and guarantees are described. Then, multiple consistency models and properties are characterized.

2.3.1 Strong vs Weak

CAP theorem [17] dictates that a replicated storage system needs to choose between either (strong) consistency or availability during failures that partition the network connecting the storage nodes. In other words, an application with replicated storage faces a trade-off between stronger forms of consistency and higher performance properties [46]. There are two high-level types of consistency: strong and weak [29, 46].

If a system provides a strong access to its services, then it is necessary to ensure constant coordination among replicas at different sites. This requires heavier-weight implementations, which leads to increased latency and/or decreased throughput for request execution. However, this prevents most of the anomalies, i.e., unexpected behaviour that is confusing to users. The stronger the consistency, the lower the occurrence of anomalies. The other advantage is that the programming model is simplified.

The motivation for having weaker forms of consistency comes from the fact that many systems must provide low latency and high throughput. Weakly consistent systems provide both of these, as well as high-availability and high-scalability. These advantages arise from the ability to allow read and write operations to be executed with little, or even no, synchronisation among replicas [61], since operations can execute contacting a single replica. Unfortunately, weaker forms of consistency have two primary drawbacks. One is that they allow executions with user-visible anomalies. A common example of this is out-of-order comments on a social network post. Considering three users (Alice, Bob and Charlie) of this application, Alice is the first one to comment on the post, then Bob comments on the same post, after seeing Alice comment, and in the end, Charlie sees Bob's comment appear before Alice's. Furthermore, inconsistencies can occur when only a single user or application is making data modifications [61]. For example, a user could issue a write at one replica, and later issue a read at a different replica. If the two replicas had not synchronised with one another between the two operations, then the user would see inconsistent results. The other drawback is that weaker consistencies increase programming complexity, since developers must reason about and handle all these complex cases.

Regarding non-transactional consistency models, if one makes some tweaks and minor changes to default ones, e.g., linearizability and eventual consistency, where their constraints are either reinforced or relaxed, new models can be generated. In fact, if we keep doing this, then more than fifty different consistency models can be created [62].

2.3.2 Linearizability

Linearizability [34, 46] is the strongest consistency model when it comes to non-transactional systems. Intuitively, linearizability ensures that each operation appears to take effect instantaneously at some point between when the client sends the request and receives the response. This requires blocking user requests waiting for cross-replica synchronisation [29], since every replica must be updated with the requested data modifications before returning to the user.

This model dictates that there exists a total order over all operations in the system, which is consistent with the real-time order of operations. For example, if operation A completes before operation B, then A will be ordered before B. A total order anomaly would occur if the opposite had happened, i.e., B was ordered before A, as it is not concordant with the real-time order. Thus, anomalies are avoided by ensuring that writes take effect in some sequential order that agrees with real time, and that reads always see the results of the most recently completed write. Since this model is very easy to reason about, it also decreases the programming complexity.

As already mentioned, the main problem of this model is the increased latency and decreased throughput, which is why developers are avoiding stronger consistency models.

2.3.3 Serializability

Serializability [14, 57] is a strong consistency model, which is transactional, i.e., its operations (called *transactions*) can involve multiple primitive sub-operations performed in order.

Under this model, transactions appear to have occurred in some total order. In other words, this model guarantees that the execution of a set of transactions over multiple data objects is equivalent to some serial execution (arbitrary total ordering) of the transactions [42]. It guarantees that operations take place atomically, since sub-operations of a transaction do not appear to interleave with sub-operations from other transactions.

Serializability differs from linearizability in three properties [42]. The first two are related to the fact that serialization is transactional, so instead of having a single-operation and single-object ethic, it does multi-operation and multi-object executions. The last one is that serializability does not impose any real-time constraints on the order of transactions. Actually, it does not imply any deterministic order, as it simply requires that some equivalent total order exists.

To achieve serializability's total order of transactional multi-object operations and linearizability's real-time constraints, then one must rely on *strict serializability* [34, 59].

2.3.4 Session Guarantees

A *session* is an abstraction for the sequence of read and write operations performed during the execution of an application. Its intent is to present individual users or applications with a view of the system that is consistent with their own actions, even if they read and write from several potentially inconsistent servers. The goal is for results of operations performed in a session to be consistent with the model of a single centralized server, possibly being read and updated concurrently by multiple clients. To this end, there are 4 guarantees that can be applied independently to the operations belonging to a session.

For simplicity, it is assumed that read operations return the entire sequence of writes. The session guarantees, as defined in [61, 29], are the following:

Read Your Writes: This guarantee is motivated by the fact that users or applications find it confusing if they update the state and then immediately read only to discover that the update appears to be missing [61]. So, this guarantee requires that a read observes all writes previously executed by the same session (client). More formally, being W the set of writes operations made by a client c at a given instant, and S the sequence of write operations returned in a subsequent read operation of c, a *Read Your Writes* anomaly occurs if: $\exists x \in W : x \notin S$.

Monotonic Writes: It requires that writes issued by the same client are observed in the order in which they were issued. This implies that a write is only incorporated into a replica if the replica contains all previous session writes [61]. More formally, being W the set of writes operations made by a client c up to a given instant, and S the sequence of effects of write operations returned in a read operation by any client, a *Monotonic Writes* anomaly occurs if the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

Monotonic Reads: This requires that every write reflected in a read must be also reflected in all subsequent reads issued by the same client. It ensures that reads are made only to replicas that include all writes that were seen by previous reads within the session [61]. More formally, a *Monotonic Reads* anomaly occurs if a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x \in S_1 : x \notin S_2$.

Writes Follow Reads: This requires that a write seen in a read by a given client always precede the subsequent writes that the same client performs. This avoids the strange situation where a client reacts to a write issued by itself or some other client by issuing another write (e.g., after seeing a post, the client adds a comment to it), and subsequently some client observes the second write without observing the first one. More formally, being S_1 a sequence returned by a read invoked by a client c, w a write issued by c after observing S_1, and S_2 a sequence returned by a read issued by any client in the system, a *Writes Follow Reads* anomaly happens if: $w \in S_2 \wedge \exists x \in S_1 : x \notin S_2$.

These guarantees can easily be layered on top of a weakly-consistent replicated data system. However, requesting a guarantee can have an adverse impact on availability, because enforcement of guarantees restricts the set of servers that may be used within a session. Indeed, to make sure that guarantees are met, the servers at which an operation can be executed must be restricted to a subset of available servers that are sufficiently up-to-date. Thus, applications must take a trade-off between availability and the consistency (number of session guarantees applied). For this reason, guarantees can be requested individually on a per-session basis.

2.3.5 Causal Consistency

Causal consistency [11] is considered a weak consistency model, however, it is the strongest one for highly available systems [47]. Briefly speaking, under causal consistency, the system's users can never observe data items before they can observe items that influenced their creation. Suppose the following scenario on a social networking site: Sally cannot find her son Billy, so she posts an update S to her friends: "I think Billy is missing!"; After Sally posts S, Billy calls his mother to let her know that he is at a friend's house, so Sally edits S, resulting in S^*: "False alarm! Billy went out to play!"; Sally's friend James observes S^* and posts status J in response: "What a relief!". If causality is not respected, another user, Henry, could observe effects before their causes, i.e., if Henry observes S and J but not S^*, he might think that James is pleased to hear that Billy's is missing. If the system had respected causality, he could not have seen J without S^*.

Causal consistency comprises Lamport's *happens-before* relationship [41] (denoted →), which is defined as follows:

- If a and b are in the same process, and a comes before b, then $a \longrightarrow b$
- If a is the sending of a message by one process and b is the receipt of it by another process, then $a \longrightarrow b$
- if $a \longrightarrow b$ and $b \longrightarrow c$, then $a \longrightarrow c$
- If $a \nrightarrow b$ and $b \nrightarrow a$, then they are said to be *concurrent*, i.e., $a \parallel b$.

In fact, the four *session* guarantees derive from the properties of this relationship. Thus, causal consistency consists of the four *session* guarantees, which are referred to as *causality semantics*. None of these should be violated if a system relies on causal consistency.

2.3.6 Eventual Consistency

Eventual consistency [46] is the weakest consistency models that one can ask for. Each read or write operation is performed at a single server, and the writes are propagated to other servers in a lazy fashion [61]. Hence, replicas can respond immediately to read operations using their current version of the data, while writes are asynchronously propagated. However, when replicas are not yet synchronised, different replicas may return different results for reads. In other words, a write might not be seen by reads within time t after it committed, which is not allowed in linearizable systems. Thus, this model requires that replicas "eventually" synchronise and agree on a value of an object, i.e., when they all have received the same set of writes, they will have the same value. Indeed, this convergence guarantee is a liveness property, however, it is not a safety guarantee, since a system cannot "violate" eventual consistency at any fixed point in time, and there is always the possibility that it becomes consistent later [11]. At any given time, clients may see a wide range of consistency anomalies, i.e., any subset of the actual writes.

Given its properties, this model is able to provide high availability with low latency and high throughput. However, the programmer will have to deal with strange and complex cases, which leads to increased programming complexity.

2.4 Verification of Consistency

This section presents an overview of what strategies have been done to verify different types of consistency the systems offer and consequently what anomalies they allow, taking into account the multiple consistency models and semantics presented in section 2.3.

2.4.1 Verifying Causal Consistency

Bouajjani et al. [16] introduced a theoretical approach for the problem of checking if a single execution of read-write abstractions, e.g., replicated key-value stores, satisfies causality. This approach detects violations of causal consistency through the occurrence of *bad patterns*.

Three different variations of causal consistency (with respect to its implementation) have been considered: causal consistency (*CC*), causal memory (*CM*) and causal convergence (*CCv*). *CM* and *CCv* are strictly stronger than *CC*. If a site "changes its mind" about the order of operations, then (*CM*) is violated. *CCv* implies that all sites eventually converge to the same state with respect to the operations that are not causally related.

An example of a bad pattern is called *ThinAirRead*, where a client observes a value of an entity, however, no client has written that value to that entity before. This is a bad pattern of every criteria (*CC*, *CM* and *CM*).

They proved that detecting these bad patterns on single executions of read-write abstractions, i.e, verify if they are causally consistent, is NP-complete. Zennou et al. [65] implemented this approach reducing the problem of detecting the existence of those bad patterns to the problem of solving Datalog queries on two different distributed databases. Regarding the results of *CC* and *CCv*, they show that the implementation is more efficient in the case of verifying *CC* and *CCv* compared to the *CM* case.

2.4.2 Verifying Session Guarantees

In order to understand the consistency levels of online service APIs, Freitas et al. [29] propose two black box tests that probe a given system (through its API) in search of occurrences of anomalies that either violate any of the four *session* guarantees, or introduce content or order divergence. Moreover, regarding the latter ones, it calculates for how long it takes for the system to recover from the divergence, i.e., converge back to a single coherent state.

The sequence of events for the first test is shown in Figure 2.3. In this test, each agent performs two consecutive writes and continuously issues reads in the background. Agents have sequential ids and the first write by agent i is performed when it observes the last write of agent $i-1$. For every operation, its invocation and response times, and their output are logged. With this information, it is possible to detect the anomalies that violate any of the four *session* guarantees. For example, if any agent observes the effects of a message M and in a subsequent read by the same agent the effects of M are no longer observed, then a *Monotonic Reads* violation occurs.

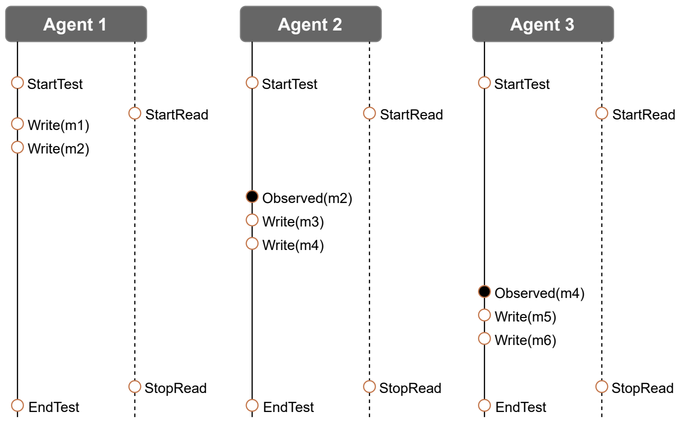

Figure 2.3: Timeline for test 1 with three agents

As for the second test, its timeline is depicted in Figure 2.4. It aims to uncover divergence among the view that different agents have of the system. All agents issue a single write (roughly) simultaneously, and continuously issuing reads in the background. This simultaneity is interesting, becuase it may increase the chances of different writes arriving at different replicas in a different order, and therefore trigger the divergence factor, e.g., if an agent only sees M1, and another agent only sees only M2.

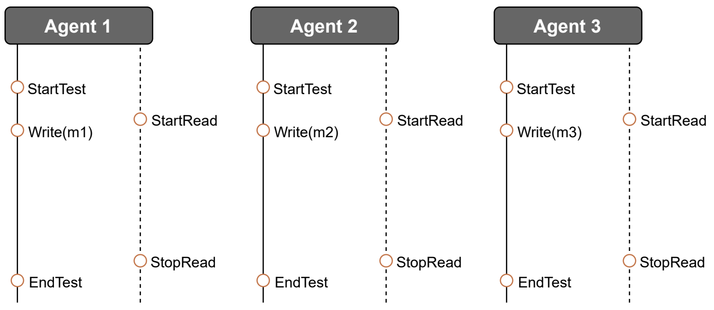

Figure 2.4: Timeline for test 2 with three agents

If a single reader is used, especially when running in the same datacenter as the writer, or even worse, running on the same machine, it is unlikely to discover staleness [15]. Distributed systems usually use some kind of load balancing mechanism. Depending on the intelligence of load balancers, it is likely that all requests from the same IP range are forwarded to the same replica or that there is even a caching layer in between. Therefore, clients must be deployed on different geographically distant locations.

Another relevant study in this context was conducted by Bermbach and Tai [15], where they focused on the consistency guarantees of Amazon S3 under a heavy load of concurrent writes. What differs both strategies is that the former study focuses on understanding the consistency properties offered by service APIs, while the latter one verifies these properties on the storage layer.

Freitas et al. performed this measurement study on three popular platforms: Facebook (Facebook Feed and Facebook Group), Google+, and Blogger. Using three agents, overall results show that all types of anomalies were seen in both Google+ and Facebook Feed, whereas in Facebook Group no violations of *Read Your Writes* and *Order Divergence* were seen. This indicates that engineers chose performance over stronger consistency models for these services. As for Blogger, no anomalies of any type were detected. Thus, Blogger appears to be offering a form of strong consistency.

2.4.3 Causality Semantics

In order to automatically measure the consistency levels and help developers the harmful degree, Tang et al. propose a testing framework called *CausalTester* [60]. This framework measures the consistency of replicated services via causality semantics based on the responses produced. The end-users perspective is that responses from replicated services follow the four *session* guarantees: Read Your Writes (RYW), Monotonic Reads (MR), Monotonic Writes (MW), and Writes Follow Reads (WFR). Indeed, quantifying the consistency with them is a good metric since the harmfulness of violating causality semantics is obvious and easy to explain.

Figure 2.5 shows an overview of this framework's workflow. A replicated data service has *N* replicas hosted on a cloud datacenter. It is remotely invoked by the upper-layer web applications for end-users that are simulated by the benchmark services. End-users access these web applications and observe the behaviour of back-end replicated services. The *CausalTester* adopts twelve test cases as the workloads and the corresponding benchmark services to obtain the responses of replicated services.

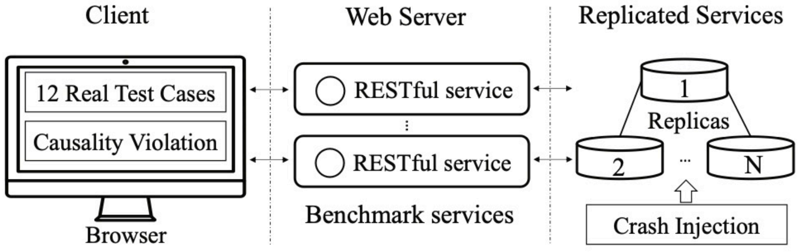

Figure 2.5: The workflow of the testing framework ([60])

The test cases were based on browsing the main pages of three platforms: Twitter, Flickr and Amazon. By analysing the HTML pages, it was perceptible which causality relationships existed on each platform. For example, on Twitter, they observed that a "tweet" and its "comments" shared that type of relationship. They found four typical test cases from each platform (twelve in total) for RYW, MR, MW, WFR, respectively. Thus, these twelve test cases can simulate end-user behaviours during browsing the web pages. These were implemented with sequences of HTTP requests, which are forwarded to the corresponding benchmark services in the web server and then transferred as read and write operations to back-end replicated services. The corresponding benchmark services were developed and deployed as RESTful services.

The REST workload of Yahoo! Cloud Service Benchmark (YCSB) was extended to include and launch the test cases requests and remotely invoke the corresponding RESTful benchmark services. When detecting the causality violations, response messages can be of three types: *normal*, *time-out* and *interruption* messages. The *normal* ones return some specific values of the reads requests. When a crash happens, the responses may include exceptions like *time-out* after a long wait, or *interruption* exceptions. Crash faults were simulated by running "kill -9" to terminate the processes of replica nodes, and restart the processes for crash-recovery.

The framework relies on a particular requesting pattern depending on the causality semantic, which may involve crashes. For example, without considering crashes, when a client performs a write operation on one replica, and tries to read (in the same session) from another replica, if the read returns a version that does not include the effects of its write, then this framework concludes that RYW is violated.

To evaluate this testing framework, three distributed databases were tested: Cassandra (weak consistency), HBase (strong consistency) and Redis (in-memory). For each replicated system, the test cases were executed 100.000 times (with and without crashes on master and replica nodes) and one counted the number of consistency violations for each of the causality semantics. Overall results show that there were detected consistency violations of every causality semantic in every distributed database. Thus, this framework is indeed effective to detect consistency violations for weak consistency and helpful to find consistency-related bugs if the strong consistency is violated.

2.4.4 Principled and Practical Consistency Analysis

Without an understanding of the consistency benefits of intermediate and strong consistency, it is difficult to fully evaluate how they compare to weaker models, and each other. Lu et al. took the first step towards quantifying those benefits by measuring and analysing requests to the social graph at Facebook [46]. Facebook's replicated storage for its social graph is a combination of a two-level cache and a sharded single-master-shard database. A cluster is composed by a group of caches, and within each cluster, *per-object sequential* and *read-after-write consistency* are provided. Across the entire system, eventual consistency is provided. They performed two types of analysis: principled analysis and practical analysis of the consistency.

The principled analysis identifies when the results of the system differ from what is allowed by stronger consistency models, i.e., what anomalies occur in the eventually consistency production system. First, a small random sample of the social graph is logged. Then, offline anomaly checkers run on those logs. An anomaly checker is the tool that returns those results that are disallowed by stronger consistency models. A set of checkers were designed to identify anomalies for three consistency models: *linearizability*, *per-object sequential consistency*, and *read-after-write consistency*. Consistency models provide guarantees by restricting the set of possible executions. These checkers identify when a traced execution violates these restrictions. Each checker does this by maintaining a directed graph, whose vertices represent the state of an object, and whose edges represent the constraints on the ordering between them. They check for anomalies by checking if the state transition order observed by reads is consistent with these constraints. Indeed, running this in real-time would be equivalent to implementing a system with stronger consistency guarantees and running it in parallel with the eventually consistent system. This overhead is avoided by only processing requests well after they have occurred, which allows the storage for the principled analysis trace to be eventually consistent, and provides plenty of time for log entries to arrive.

In contrast, practical analysis is used as a light-weight real-time monitoring tool. As a consequence, it does not trace all operations on a given object, so it does not give insights into how often principled consistency models are violated. Instead, it uses injected reads to track metrics that are designed to mirror the health of the different parts of the replicated storage. It uses the metric $\phi(P)$-consistency, where P represents a set of replicas, which dictates the frequency that injected reads for the same data to all $p \in P$ receive the same response from each p. The usefulness of this metric derives from how it quickly approximates how convergent/divergent different parts of the system are. Increases in network delay, replication delay, misconfiguration, or failures cause a drop in $\phi(P)$-consistency. Furthermore, an increase in the write rate of the system will also decrease it, because there will be more writes in flight at any given time. This metric is composed by two major ones: $\phi(G)$-consistency and $\phi(R_i)$-consistency. The former serves for tracking the health of the overall system, which corresponds to the global set G of all replicas, i.e., all leaf and root cache clusters at Facebook. The latter is for the set of all cache clusters in region R_i. A similar metric is Γ (gamma) [31], which not only captures the frequency of client-observed consistency anomalies, but also determine their severity.

One major finding was that Facebook's social graph is highly consistent, where 99.99% of reads to vertices returned results allowed under all the previous described consistency models. However, there were anomalies under all of those consistency models. Even though these anomalies are rare, this demonstrates that deploying them is beneficial.

2.5 Tools

This section presents popular tools for verifying and analysing systems.

2.5.1 Load Testing Tools

Load testing an application is useful, since we can measure functional and non-functional properties of the system by simulating the way real clients submit requests to the application. Popular tools for load testing are: Artillery [9] and Apache JMeter [6].

For example, in Artillery, a test is characterized by running scenarios containing specific types of requests, where we can assign weights (frequencies) to those scenarios, define the number of requests per second and how they should ramp up, among other configurations. The developer must provide these properties through a YAML script. Artillery gathers the metrics and results it obtained during the execution of the test, with the option of displaying them through plots for better understanding.

2.5.2 EvoMaster

Evomaster [7, 8, 48] is a white-box testing tool for RESTful web APIs. This tool generates system-level test cases by exploiting the source code of the system under test. More specifically, it integrates a search-based technique, where test cases are evolved and evaluated independently, and only at the end of the search, a test suite is constructed by choosing the combination of test cases that maximizes code coverage (source code statements) and HTTP status codes. Each test case contains a sequence of at least one HTTP call.

The tool consists of two main components: the core, and a controller library. The core addresses the generation and evolutions of test cases. The controller library is responsible for obtaining metrics from the system under test, e.g., code coverage, and starting/stopping/resetting the system, e.g., it starts the system before running the generated test suite.

2.5.3 Netflix Simian Army

Since Netflix moved its services to the cloud, they have been focusing on ways to improve availability and reliability over its services. To this end, they have relied on a set of experiments that are based on a combination of chaos engineering and fault-injection techniques.

Netflix's solution is called *Netflix Simian Army* [35] which consists of a set (*army*) of tools, called *simians*, that are responsible for introducing different types of failures and monitoring the cloud environment. These *simians* are the following:

Chaos Monkey was the first *simian* produced by Netflix. This tool randomly disables instances in production, so that one can test the system ability to survive this common type of failure and make sure that customers are not affected. The success and effectiveness of this tool inspired Netflix to create more *simians*.

Latency Monkey introduces artificial delays or network lags in order to simulate service degradation so one can check if upstream services are able to respond appropriately.

Furthermore, by introducing very large delays, one can simulate a downtime of a node or even of an entire service, i.e., service unavailability.

Conformity Monkey is responsible for finding instances that are not in compliance with best-practices, shutting these down. An example of this is that when it finds an instance that do not belong to an auto-scaling group, eventually, it will end up in trouble. These type of instances are shut down, so the owner is given the opportunity to re-launch them properly.

Doctor Monkey relies on running health checks operations and monitoring other external signs of health (e.g., CPU load, and other metrics that have been mentioned in section 2.2.3) on each instance in order to detect the unhealthy ones. When detecting an unhealthy instance, it is removed from service. Eventually, it is terminated once the service owners root-cause the problem.

Janitor Monkey searches for unused resources in the cloud environment and disposes them. This way, it is ensured that it is running without any clutter or waste.

Security Monkey extends Conformity Monkey, focusing on security. It terminates every instance that contain any type of security violation or vulnerabilities, like inappropriate configurations in cloud security groups.

10-18 Monkey is used to detect configuration and run time issues in instances that provide its services to customers in multiple geographic regions, using different languages and character sets.

Chaos Gorilla has the same philosophy as the Chaos Monkey, but it has bigger ambitions. This *simian* simulates an outage of an entire cloud availability zone (i.e., data center). During this brutal situation, one wants to verify that services are able to automatically re-balance to the functional availability zones without manual intervention or user-visible impact.

This tool has revealed to be very useful for Netflix because it tests the resilience of its systems when subject to multiple failures injected by the *simians*. This increases the confidence over the ability of the systems to handle failures that will, eventually, happen in production environments and to minimize their impact to its customers. This confidence is directly proportional to the number of executed *simians* that did not result in any impact on systems or customers.

2.5.4 Jepsen

Jepsen [36] is an open source library that aims to improve the safety of distributed databases, queues, consensus systems, etc. It resorts to fault-injection and black-box testing techniques in order to explore whether the system lives up to its documentation's claims, report new bugs, and suggest recommendations for vendors. In other words, it tries to detect executions where properties claimed by the system are violated. The first step is to obtain the system's invariants, i.e., the properties that the system claims it holds, which is achieved by carrying out a rigorous study of the system's documentation. Then, to verify if these invariants hold during its execution, Jepsen tests are designed.

Figure 2.6 illustrates the test process of Jepsen on a distributed database system, in a Docker environment [53]. By default, a cluster with six containers is built, one of which is the control node and the other five are database nodes. As soon as a test starts, the control node will create a group of worker threads to access the database nodes simultaneously via the SSH protocol, where each worker thread contains its own client. The generator tell the client what operations it should perform against the system. The *nemesis* tells the client what fault it should inject to the system. The programmer needs to specify some Jepsen configurations such as: the operations to be performed, the failure types to inject, among others.

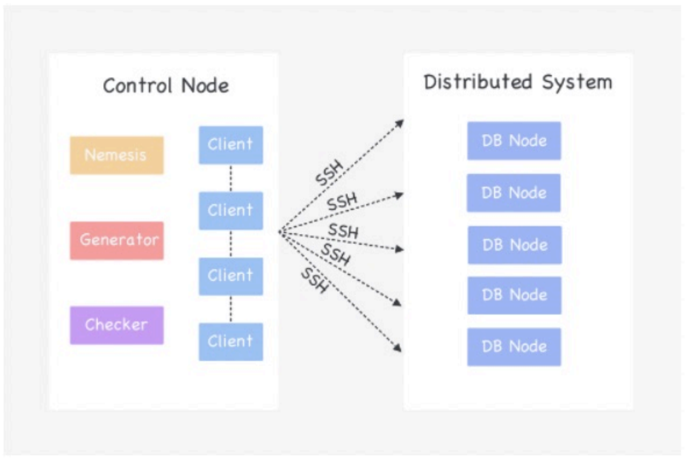

Figure 2.6: Jepsen test process ([53])

For each test performed, a history [22] is generated. A history is a collection operations performed, including their concurrent structure. Jepsen splits each operation in two: its invocation and its completion. For each of these, it stores the following: the worker thread that performed it, the indication whether it was invoked or completed, its type (read or write), its timestamp, and their parameters and return values.

Having the history of a Jepsen test, it will be analysed by the checker which will tell if these concurrent operations represent a valid execution, given the system's invariants. Jepsen includes two checkers: Knossos and Elle. It also allows developers to define their own custom checkers.

Knossos [40] is the default checker of Jepsen and verifies the linearizability of experimentally accessible histories. Given an history of operations, and a model that describes the behaviour of the system when requesting specific operations, Knossos determines whether the history is linearizable or not. We have already seen what a linearizable execution is in section 2.3.2. Knossos tests multiple orders of the same history of operations, until finding one that is indeed linearizable. Otherwise, an error is returned.

Elle [39] is a transactional consistency checker for black-box databases. This checker aims to detect transactional anomalies through client-observed transactions, such as *dirty* *updates*, where a transaction commits a version based on some uncommitted state, and *garbage reads*, where a read observes a value which was never written, among many others. This implies that a history is now a collection of transactions, instead of single operations. Elle looks for a sequence of events in a history which could not possibly have happened in that order, and uses that inference to prove that the history cannot be consistent [27]. It is necessary to build a dependency graph among the different transactions. However, to capture these dependencies, one must resort to histories that satisfy both *traceability* and *recoverability*. These properties require that reads of an object return its entire version history, and there is a unique mapping between versions and transactions. Thus, *read* operations must return every written value so far (ordered), and *write* operations produce unique values. We could not analyse a history composed of *counter* operations, where a write is an increment, and a read returns the version history like $(0, 1, 2, ...)$, because, we might not tell which increment produced a particular version, which violates *recoverability*. The only data type that hold both of those properties is the ordered list, which is the one Elle uses. This dependency graph is actually a sub-graph of every possible history compatible with that observation. Imagining that the database had internally the true history (it has not), i.e., the complete graph, Elle reconstructs a sub-graph of it. Consequently, one might not observe all of the edges.

Elle is more advantageous than Knossos regarding output and performance, due to its cycle detection mechanism. Knossos simply indicates whether a history is linearizable, whereas Elle provides a human-readable explanation for the cycle and why it implies a contradiction. Knossos is often limited to a few hundred operations per history, while Elle can handle hundreds of thousands of operations easily, which is due to the fact that when increasing the history length, Knossos's execution is mildly super-linear, whereas Elle's is linear [27]. Furthermore, when increasing concurrency, Knossos's execution is exponential, whereas Elle's is constant [27].

2.6 Conclusion

This section presents a small conclusion about the related work through Table 2.1, which includes an overview of the main characteristics of each of the tools studied previously. For each tool, it is indicated its verification type, which techniques it is based on, what properties of the system are analysed to determine if the system is correct or not, and how the results are validated to infer the correctness of the system.

Artillery and Apache JMeter are very similar to each other, regarding these properties. Being load testing tools, they rely on dynamic verification, perform black-box tests and analyse the executions of the system. The developers must evaluate the results to determine the system's correctness.

EvoMaster also relies on dynamic verification, since it executes a generated test suite. It performs white-box tests, because it constructs the test cases based on the analysis of

Table 2.1: Properties of the tools studied

Tools	**Verification type**	**Techniques used**	**What it analyses**	**Results validation**
Artillery	Dynamic	BBT	Executions	Developer
Apache JMeter	Dynamic	BBT	Executions	Developer
EvoMaster	Dynamic	WBT	Executions	Automatic
Netflix SA	Dynamic	CE and FI	-	Developer
Jepsen	Dynamic	BBT and FI	Executions	Automatic (LN or SR)

WBT: White-Box Testing, BBT: Black-Box Testing, CE: Chaos Engineering, FI: Fault-injection, LN: Linearizability, SR: Serializability

the system's source code. In the end, it automatically determines the correctness of the system based on its executions against the test cases.

Netflix Simian Army also relies on dynamic verification, and is based on chaos engineering and fault-injection techniques to expose the system to very turbulent conditions. Despite the power of this tool, it does not analyse any properties of the system, as developers must analyse its behaviour, e.g., through monitoring, and determine its correctness.

Jepsen also relies on dynamic verification, and performs black-box tests with the possibility of injecting faults. It analyses the system's executions against the tests and faults injected. Based on them, it automatically determines the system's correctness, verifying whether they are linearizable/serializable, depending on the checker, or not.

3 Solution Design

This chapter starts by comparing the differences between verifying causality in databases and RESTful applications. Then, it presents the design of the algorithm that verifies causal consistency in RESTful applications, explaining the processing mechanism and the algorithm's behaviour. Finally, the pseudocode is presented and analysed.

3.1 Causality Violations Overview

This section presents an overview about the reasoning of detecting causality violations when dealing with simple database operations (reads and writes), and REST operations, pointing their main differences and difficulties.

Regardless the system (database or RESTful application) we aim to test, every single operation has an invocation step and a termination step, which correspond to two different points in time. The invocation step happens when the client issues the operation to the system, and the termination step happens when the client receives an acknowledgment (response) from the system. Thus, the operation can take effect in the system at any point in time in between the invocation and termination points, inclusively.

3.1.1 Simple Database Operations

Suppose a simple database that stores objects identified by an *id* with an integer associated to it. It supports the basic operations of write and read to manage these entities. Figure 3.1 illustrates the timeline of a history of operations for that database, where the left end and right end sides of each box denote the invocation and termination points in time for the current operation. It starts with Client 0 performing a write operation, which assigns the value 1 to x. Right after, Client 1 issues a read of x, which retrieves the value Client 0 wrote (1). Finally, Client 0 issues the same read, which retrieves the information that the entity identified by x does not exist. This history contains a Read Your Writes violation, defined in Section 2.3.4, because Client 0 is not able to read what itself wrote, when no other client wrote to this entity meanwhile. The only way to make this history causally consistent would be for the Client to be able to read the entity x with the value of 1.

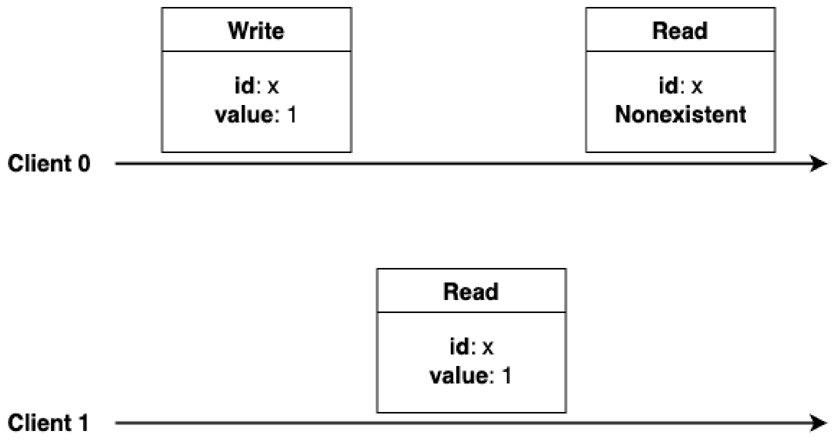

Figure 3.1: Read Your Writes violation of a history of simple database operations

Figure 3.2 shows the timeline of another history of operations. Client 0 performs two write operations consecutively, with the values 1 and 2, respectively. After this, Client 1 issues two read operations consecutively. This history contains a Monotonic Writes violation, defined in Section 2.3.4, since Client 1 observed the writes of Client 1 on a forbidden order, i.e., not the order they were issued. Since the first read of Client 1 reflects the second write operation of Client 0, the subsequent reads of Client 1 must reflect, at least, the value 2. There are several possibilities for the values reflected in the read operation of Client 1 to make this history causally consistent. Considering that the pair of values correspond to the values of the first and second read operations of Client 1, respectively, the possibilities are:

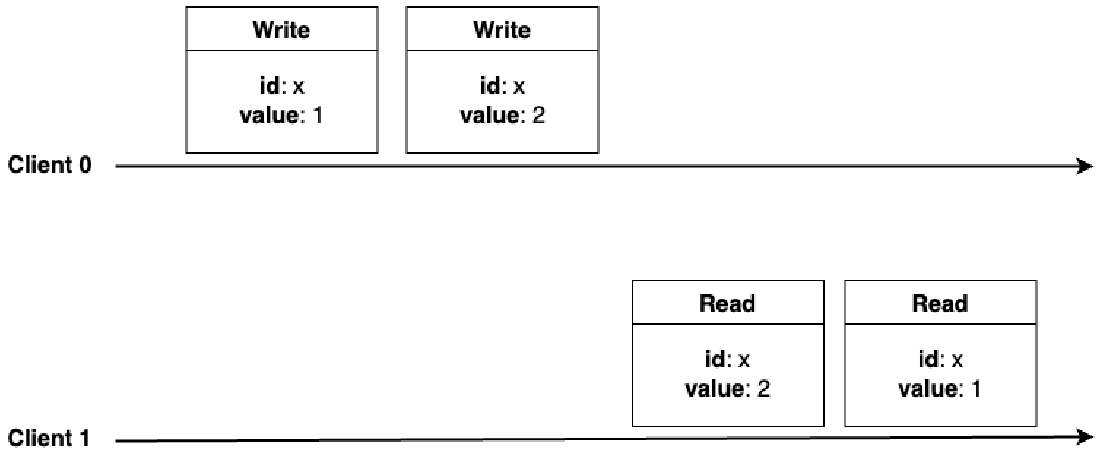

Figure 3.2: Monotonic Writes violation of a history of simple database operations

- *Nonexistent* and *nonexistent*
- *Nonexistent* and 1
- *Nonexistent* and 2

- 1 and 1
- 1 and 2
- 2 and 2

Thus, as far as these simple database operations are concerned, it seems that the read operations are the only points we need to stop and validate if we can justify the value read, i.e., search for the possible dependencies, through write operations whose value written is the same and that the invocation point happened before the termination point of the current read operation.

3.1.2 REST Operations

Now let us consider a simple RESTful application that manages entities equivalent to the ones seen before, i.e., they are identified by an *id* and have an integer value associated. The operations available are creation (POST), read (GET), update (PUT) and delete (DELETE).

Figure 3.3 shows the timeline of a history of REST operations of that RESTful application. Client 0 creates two entities consecutively, where they have the values of 1 and 5, and the *id*s assigned by the application to them are x and y, respectively. After that, Client 1 reads entity y, which retrieves the value of 5, and then creates a new entity x with the value of 2.

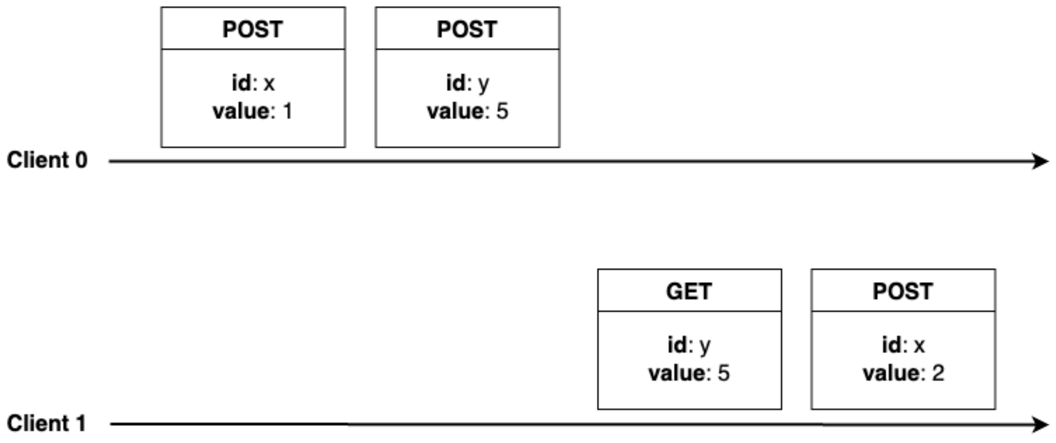

Figure 3.3: Monotonic Reads and Monotonic Writes violations of a history of REST operations

At first sight it seems this history is valid regarding causality. If we ignored the REST semantics, i.e., the POST operations would be simple write operations and the GET operation would be a simple read operation, the only operation we had to validate would be the read. However, since Client 1 is reading an entity that Client 0 previously wrote, there are no violations.

That is not the case when considering these REST operations, because of their semantics. Whenever a client issues a POST operation, the application must ensure that the generated *id* does not exist among the entities that the client is able to observe, before assigning the new value to it. The opposite would happen when dealing with PUT and DELETE operations, where the application must ensure that the entity with the given *id* exists. The two POST operations of Client 0 are valid, since there were no entities with those *id*s. However, since Client 1 observes the second operation of Client 0, this client must be able to observe the existence of *x* as well. Thus, if the POST operation of Client 1 is successful, Monotonic Reads is being violated.

Therefore, it seems that verifying causal consistency in a history of REST operations is more difficult than in a history of simple write and read operations. Everything that is checked on the latter one must also be checked in the former one. However, in REST write operations, there is this additional concept of implicit read operations. Before performing the actual write, these operations are responsible for verifying the existence or nonexistence of the given *id*, depending on the write HTTP method, always taking into account what the client is able to observe at the moment.

3.2 Processing an History of Operations

In this dissertation, we have developed an algorithm that verifies if executions of RESTful applications satisfy causality (causal consistency) or not, based on histories of operations.

The project was built on top of JepREST [58], detailed in section 4.1, so that histories of operations executed in RESTful applications could be logged and analysed.

Let us assume a very simple RESTful application that manages objects, which are identified by an *id* and their value consist of the *id* and a *char*. The *JSON* structure of how these objects are managed on the application is shown in Listing 3.1, where an arbitrary object is depicted.

```
{
    "id": "x",
    "char": "A"
}
```

Listing 3.1: *JSON* representation of the objects managed by the simple RESTful application

Our algorithm assumes that applications include standard REST operation with the normal semantics, as shown in Table 3.1. Users can create an object by sending a *JSON* object like the one specified in Listing 3.1, without the *id* property, with the *id* being generated by the application. The response body of this operation contains the object with the generated *id*. In addition, they can read, update and delete an object, where the *id* of the entity is specified through the URI of the correspondent endpoint. However, if the id specified does not correspond to any object stored, the operation in context fails with the code 404. This will be the RESTful application that this chapter will use to present our solution.

After executing a workload of operations, which is composed by multiple concurrent clients, JepREST generates the file that contains the history of operations performed. A

Table 3.1: REST operations over the objects of the simple RESTful simple application

Operation	HTTP Method	URI	Request Body	HTTP Status	Response Body (on success)
Create	POST	/	{"char": {*char*}}	201	{"id": {id}, "char": {*char*}}
Read	GET	/{*id*}	-	200/404	{"id": {id}, "char": {*char*}}
Update	PUT	/{*id*}	{"char": {*char*}}	200/404	{"id": {id}, "char": {*char*}}
Delete	DELETE	/{*id*}	-	200/404	-

crucial characteristic of the workload that JepREST/Jepsen executes is that clients perform operations in a sequential way, i.e., a client can only perform a new operation when the response of the previous one has been received and reported. The original history file is converted to a *JSON* file. Each operation is recorded as a pair of history entries: request and response. This file records the order of the operations performed by the clients. Whenever a client sends the request to the application, the request's history entry is appended to the file, and the same happens when the response is received. Each possible field of these history entries is specified as follows:

- **type** - Indicates if the current entity corresponds to the request (*invoke*), or response (*ok*).
- **f** - Represents the HTTP method for the current operation.
- **value** - Consists of metadata about the actual request/response:
  - **input** - Consists of information about the request.
    * **json** - Contains the body of the request.
    * **typeOp** - Consists of the Clojure method name of JepREST that submitted the request.
    * **path** - Consists of the *id* property of the object in context, which is not applicable to creation operations.
  - **output** - Consists of information about the response.
    * **status** - Corresponds to the HTTP status code.
    * **body** - Contains the body of the response.
- **process** - Indicates the client that performed the operation.
- **index** - Indicates the index of the current element in the history file.
- **opposite-index** - Indicates the index of the opposite element in the history file. If the current element comprises the request's history entry, this field indicates the *index* of the corresponding response's history entry, and vice versa.

Not every history entry consists of all of these fields. If the current history entry represents the request (*type* field is *inv*), then the *value* field should not contain the *output* field, since this information is only known once the response is retrieved. When we are dealing with a response's history entry, the *value* have both the *input* and *output* fields. In addition, depending on the type of operation we are dealing with (*f* field), the *json* and *path* fields might not occur. If the operation represents a creation, then the history entries will not contain the *path* field, since the *id* of the object is unknown at the request stage. However, for a read operation, the *path* field is needed to specify the id, but the *json* field is discarded, since the request's body remains empty.

Figure 3.4 illustrates the timeline of an example of a history of operations. Client 0 starts by creating an object with the *char* field set to "A", where the application assigns this entity to the *id* "x". Then, it updates it to have the *char* "B". Then Client 1 issues two consecutive reads of that object, where it first sees the value with the *char* set to "B" and set to "A" afterwards. The execution corresponding to this history is not causally consistent, because Monotonic Writes, defined in Section 2.3.4, is being violated and the reason for that is the same as in the Figure 3.2, as the second read of Client 1 must reflect at least the *char B*.

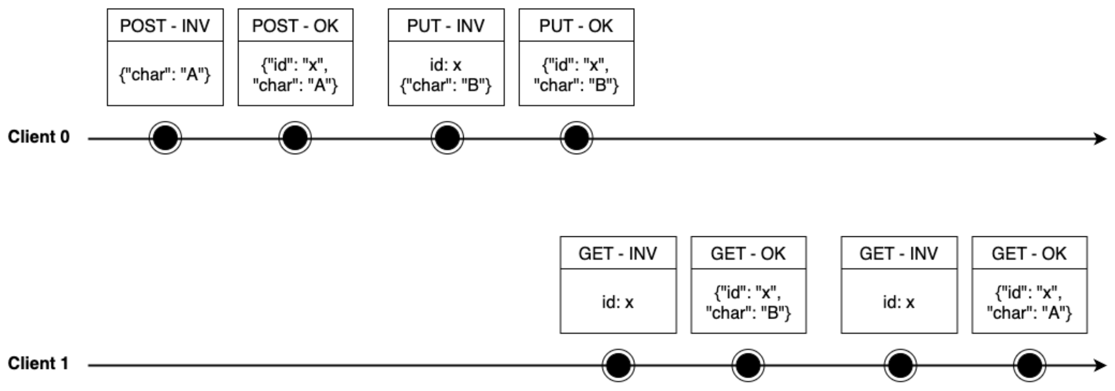

Figure 3.4: Timeline of a history of operations

We can classify an operation as a read or write operation. Operations that ask for the retrieving an object (GET) are considered read operations. Operations that create (POST), update (PUT) or delete (DELETE) an entity are considered write operations. As stated already, every operation starts from a request stage and ends with a response stage, and the history in the operations file includes both.

Depending on the operation type, they are converted into different processed operations, which are the ones that the algorithm operates on. Every processed operation has, at least, to maintain the following data:

- **index** - Index of the current processed operation
- **client** - Client that performed the operation
- **id** - Id of the object associated to the operation

- **value** - Value of the object that was read/written

Next, the mechanism of converting the operations into the processed ones is presented.

3.2.1 Explicit Read operations

As far as read operations are concerned, the only history entry that matters is the response one, as this is the moment when we receive the actual information of the object with the given *id*, and the request's history entry adds nothing to our knowledge. Thus, the invocation of this operation (request's history entry) is discarded and the processed operation will take place at the position of the response's history entry.

The algorithm stores this type of operations as explicit read operations, because the value of the object read is a concrete piece of information, which is assigned to the *value* field of this processed operation. In the case that the read operation resulted in a 404 (nonexistent entity), the *value* field will hold *null*.

3.2.2 Write and Implicit Read operations

Regarding write operations, we need to take into account the REST semantics. The creation (POST) of an object with a given *id* can only happen if this *id* is not associated to any other object, based on what the client can observe. As for an update (PUT) or a delete (DELETE) of an entity with a given *id*, it is necessary for the client to observe the entity with that *id*.

It is necessary to convert this type of operations into separate processed operations: a (processed) write operation and an implicit read operation.

A (processed) write operation represents the act of the client writing something. It stores in the *value* field the value that the client assigned to the given *id*. In case of a delete operation, the field *value* will hold *null*. This processed operation type has two additional fields: *method* and *dependents*. The *method* field holds the REST method of the write operation (POST, PUT or DELETE). The *dependents* field consists of a list that keeps track of the processed operations that depend on this one. This last field will be explained in more detailed in the following sections.

An implicit read operation is responsible for checking for the existence or nonexistence of an *id*. It stores the type of existence check in a *boolean* field called *nonNullableValue*. If the entity must not exist for the write to take place (POST), the field will hold *false*, otherwise *true* (PUT and DELETE). The *value* field of implicit reads is simply discarded, since there is no concrete value to look up.

When the processing mechanism finds a request's history entry in the history of operations file, it can easily access the corresponding response's history entry through the *opposite-index* of the former one (and vice versa), which indicates the *index* of the response's history entry. This is specially useful when dealing with write operations in order to check if they resulted in success (201 or 200) or not (404). If it resulted in failure (404), it is necessary to verify if the given *id* does not really exist. Thus, this pair of history entries will be processed as a single explicit read operation, whose *value* property will hold *null*.

When dealing with successful write operations (201 or 200), at first sight, it makes sense to check for the existence or nonexistence of the given *id* before actually performing the writing of the value, which is, indeed, the behaviour of RESTful applications. In other words, the implicit read operation would take place at the position of the request's history entry and the (processed) write operation would take place at the position of the response's history entry.

Figure 3.5 demonstrates the timeline of an example of a history of operations without applying the processing mechanism yet. Client 0 issues the creation of a new entity (POST) with the *char* "A", where the application assigned *x* as its *id*. In between the invocation and termination of that write operation of Client 0, Client 1 issued a read operation (GET) of that same *id*, where the response retrieved exactly what Client 0 wrote.

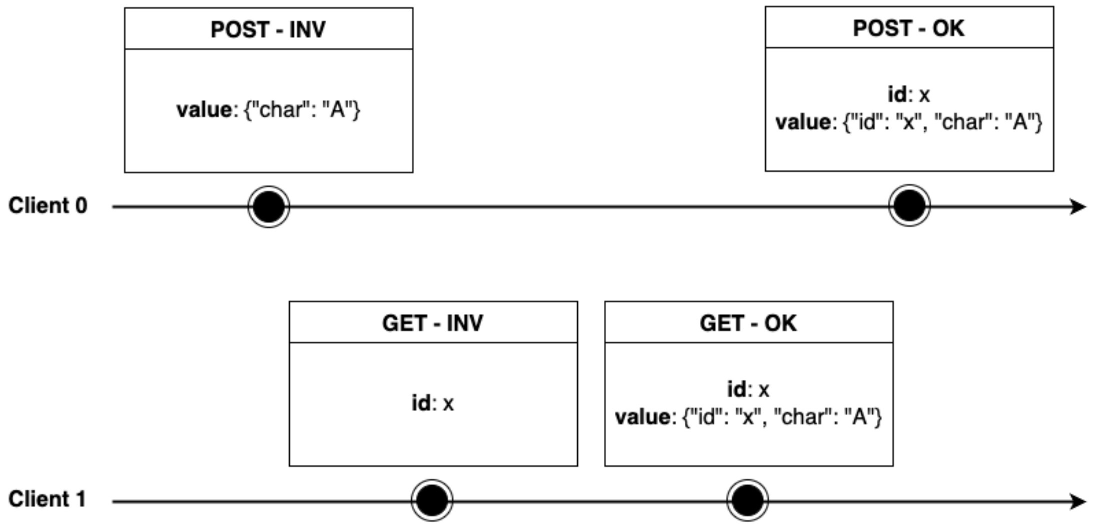

Figure 3.5: Timeline of a history of operations without applying the processing mechanism

In reality, any REST write operation can take place, i.e., it can be successfully written in the RESTful application, at any point in time in between the invocation and termination, inclusively. However, the client only acknowledges that when the response arrives.

Client 0's operation could have happened at any point in time in between its invocation and termination. However, the only write operation that justifies Client 1's read operation is Client 0's. This means that the moment when the write was "established" in the RESTful application was between the invocation of Client 0's write operation and the termination of Client 1's read operation. This property is what makes this history valid, as far as causality is concerned.

Figure 3.6 illustrates the same timeline of the history of operations, however, the processing has been applied following the characteristics discussed above (history entries and processed operations correspondence). Since the operation performed by Client 0 consists of a POST method, the correspondent implicit read operation needs to ensure that this client cannot see an existing entity with that *id*, which is why the *nonNullableValue* property of it holds *false*. The corresponding (processed) write operation represents the moment in time when Client 0 performed the value writing. In between these processed operations, the read operation of Client 1 was processed into a explicit read operation, whose *value* holds what Client 0 wrote.

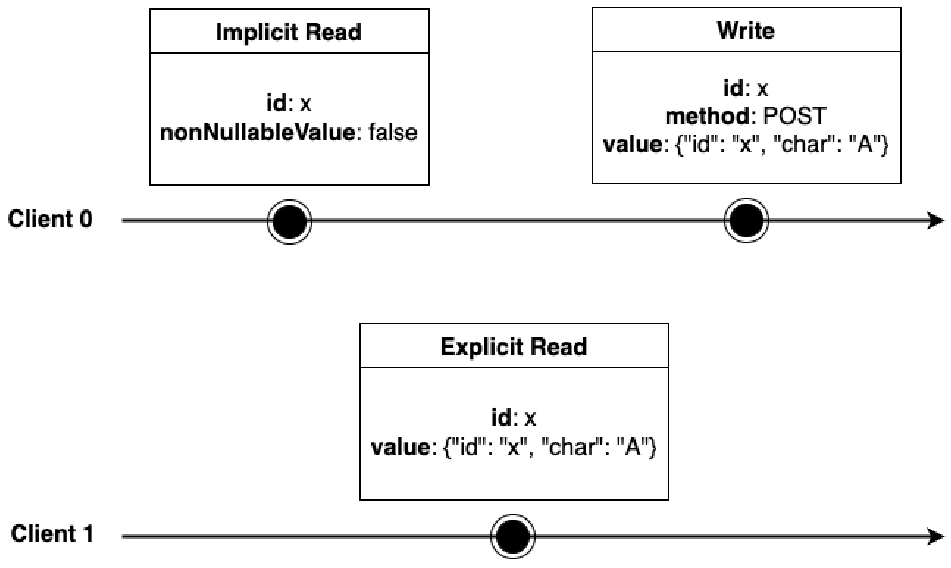

Figure 3.6: Timeline of processed operations assuming the request's history entry is associated to the implicit read operation and the response's history entry is associated to the (processed) write operation

From the algorithm point of view, the point in time when other clients are able to observe the write takes place at the (processed) write operation, which is after the explicit read of Client 1. However, no client is supposed to observe that an entity has some value, when no one has even created that entity before. This is why the processing mechanism must associate the request's history entry to the (processed) write operation and the response's history entry to the implicit read operation, instead. This way, the algorithm makes the write visible to other clients starting from its invocation point. Even though the write can be seen from the moment that the (processed) write operation takes place, it will only be "established" by the implicit read operation after checking for the existence/nonexistence of the *id*. Obviously, the check cannot take into account this write. This behaviour will be studied in more detail further ahead.

RESTful applications might return different information on their responses. In the case of the application we have been dealing with, whenever clients perform POST or PUT operations that end up in success, the responses contain an entity. The most common behaviour is for the response's entity to be concordant with what the client requested, i.e., if the client requested to update the entity with a given *id* to have the *char* to be "Z", the response's body would contain what is depicted in Listing 3.2.

When the response of a write operation returns the created/updated entity, after

```
{
    "id": "x",
    "char": "Z"
}
```

Listing 3.2: Body of the response after a client updated the *char* to be "Z" of the RESTful application

building the (processed) write and implicit read operations, it is useful to also build an explicit read operation with the *value* holding the current write operation response's body. When dealing with a successful delete operation, the final explicit read will contain *null* as its *value*.

Figure 3.7 illustrates the same timeline of the history of operations, with the processing mechanism executed completely. This is the real data the algorithm will analyse in order to check if the RESTful application satisfies causality. These operations are executed by the algorithm following the timeline order. In this case, the first operation executed would be the write of Client 0, then the explicit read of Client 1, then the implicit read of Client 0, and finally the explicit read of Client 0.

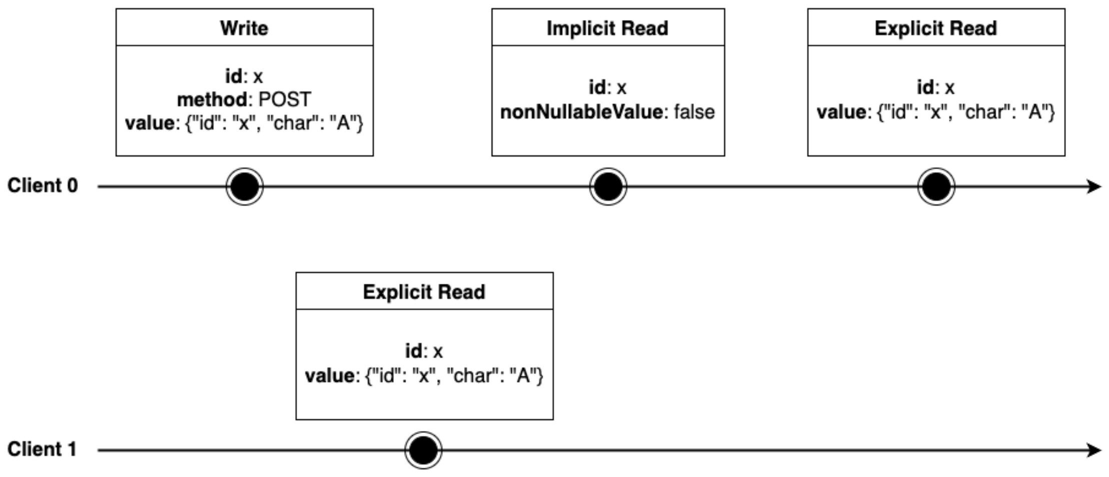

Figure 3.7: Timeline of a history of operations after applying the correct processing mechanism

In the following sections, this additional explicit read operation will not be considered when processing the histories of operations for the sake of space, and because it would not make a difference on the analysis of the histories presented.

3.3 Algorithm

The algorithm tries to determine dependencies among read and write operations that justify the results observed, as far as causal consistency is concerned. Since the algorithm is not aware of the (number of) replicas of the RESTful application, the dependencies have to be inferred from the operations performed by the clients.

3.3.1 Vector Clocks

Having the history of operations file processed, the algorithm is based on going through the processed operations in order, associating vector clocks [49] to multiple pieces of data. In the context of this project, a vector clock consists of a simple array of integers whose size corresponds to the number of clients present in the history of operations. Each position of these vector clocks is associated to a different client and summarizes the operation executed by each client that should be observed - e.g. if position i of the vector has value n, it means that the state observed must reflect the first n operations of client i.

3.3.2 Clients' data knowledge

While processing the operations, the algorithm will maintain for each client two main properties that define its knowledge about the data, which include the vector clock data structure: *state* and *idProperties*.

A vector clock is used to represent the *state* of each client. Suppose we are dealing with a history of operations that contains three clients and at a given moment in time their *states* are [1 0 0], [1 0 0] and [1 0 1], respectively. Regarding the *state* of the first client, the only position that contains a non-zero value is the one that represents itself, which means it has performed a write. The *state* of the second client has a non-zero value at the first position, which corresponds to the first client, meaning that it has read the write of the first client. The same happens for the third client, however, it has performed a write itself (third position) that the other clients have not read it (yet).

The *idProperties* property is a more complex data structure. It consists of a map that stores for each *id* present in the history of operations a list of special objects, called *ValueProperties*, that maintain a *value* and a vector clock. The *value* property corresponds to an entity that the given client has written, or read (explicitly), or might be able to observe further ahead, for the correspondent *id*. The vector clock property corresponds to the *state* of the last client that has either written or read (explicitly) that particular entity, which basically denotes the version of the *value* for the respective *id*. These lists are designed in a way that there are only concurrent *ValueProperties* (vector clock property) for each of the *id*s. Two vector clocks are concurrent if one is neither greater than nor less than the other when doing an element-by-element comparison [44]. Let us suppose we had three *ValueProperties* with the vector clocks of [1 1 0], [0 1 0] and [0 0 2]. The first vector clock is more recent that the second one, which means the second one can be discarded. The remaining ones are concurrent, which means the list in context would only contain the first and last *ValueProperties* objects.

Figure 3.8 illustrates an example of the *idProperties* structure of a client. Regarding x, its list of *ValueProperties* contains two elements, having the vector clocks [1 0 0] and [0 1 0] and values A and B, respectively. Both lists of *ValueProperties* of the *id*s y and z contain a single element.

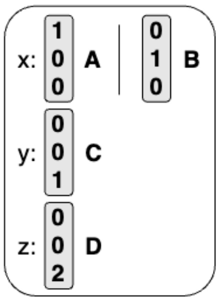

Figure 3.8: Example of the *idProperties* structure of a client

3.3.3 Dependencies and Merge Process

Let us consider only the cases when there are no explicit or implicit read operations between a write and corresponding implicit read of another client that might justify those read operations.

The algorithm tries to validate a read operation (either explicit or implicit) by checking if the client's own knowledge or the writes of other clients can justify it. This is why clients must report not only their knowledge of the data at the moment they are about to perform an operation, but also when their writes get established, which take place at the implicit read operations after checking for the existence/nonexistence of the given *id*, considering what the client might be able to observe.

For the case when a client performs an explicit read, the algorithm first attempts to justify by considering the *idProperties* of the client itself. If the list of *ValueProperties* that corresponds to the *id* of the operation contains an element whose *value* coincides with the entity read, then the client itself is able justify it. Otherwise, it tries to justify it through writes executed by other clients. For each other client, it iterates their established writes, from the most recent to the oldest. If the *state* of the client performing the read is more recent or equal than the one the other client had when establishing the write, then the algorithm stops the iteration for this other client, because it means that the client performing the read already has that knowledge. When that is not the case, the established write justifies the read if both the *id* and the entity of the write coincide with the *id* and the entity of the read.

For the case when a client performs an implicit read operation, the algorithm's behaviour is very similar. The difference is that we do not have a concrete entity to search for. If the *nonNullableValue* field is *false*, the client itself (through its current *idProperties*) or the other clients (through their valid established writes) can justify it if they contain *NULL* entities. However, if the *nonNullableValue* field is *true*, they can justify it if they contain any entities that are different from *NULL*.

Whenever an explicit or implicit read could be justified through the client itself, the state of it will remain the same. For the case of an explicit read, the *ValueProperties* list of the respective *id* will only contain a single element, whose value corresponds to the value read and the vector clock coincides with the new state of the client. As for an implicit read, the *ValueProperties* list of the respective *id* will only keep the elements that satisfy the implicit rule (*nonNullableValue*), i.e., the ones that do not satisfy the rule are removed from the list.

Whenever an explicit or implicit read could be justified through an established write of other client, the resulting state of the client is a merge of its current state and the state of the other client at the moment of establishing the write. The merge process consists of building a new vector clock and assigning to each position the most recent version of both vector clocks at that particular position. If the client had the state [1 1 0] and the other client had the state [3 0 2] when establishing the write, the resulting state for the former client would be [3 1 2]. Both the *idProperties* of the clients are also merged respecting the semantics of the *ValueProperties* semantic. In the case of an explicit read, the list correspondent to the given *id* will contain a single element, whose value coincides with the value read and the vector clock is the same as the client's merged state. As for an implicit read, all lists are treated equally.

Regarding implicit read operations, when the merge process is over, it is responsible for establishing the write for the given client. The state of the client is incremented at the position associated to itself, and the list of *ValueProperties* associated to the *id* that was written an entity to will contain a single element, whose *value* is the entity written and the vector clock is equal to the new client's state. This new client's knowledge of the data is recorded as an established write.

Figure 3.9 represents a portion of the timeline of a history of operations and shows the clients' data knowledge at each operation. Considering each processed operation, the blue box above it represents the state of the client, and the box under it represents the client's *idProperties*, which result from execution of that operation. For the sake of space, the value of the elements of the *ValueProperties* list corresponds to the entity's *char* field. The established writes are represented as the knowledge at the implicit read operations. Let us assume Client 1 had the knowledge specified before executing the explicit read and also that Client 1's explicit read can only be justified by the Client 0's established write. The merge process starts by merging the state of Client 1 and the state Client 0 had when establishing that write, resulting in [2 1]. Finally, the merge of the *idProperties* is performed. The list of *ValueProperties* of x (the *id* read) will only contain the element whose value is what Client 1 read (A) and vector clock is also [2 1]. Regarding the list of y, since the version [1 0] is more recent than the one Client 0 has ([0 0]), the former one is kept. The opposite happens for z, since the version at Client 1 is more recent than Client 0's.

3.3.4 Trace and Backtracking

At any point in time, the algorithm is representing a dependency graph that consists of the operations performed until that time, where the dependencies are related to how the read operations are being justified.

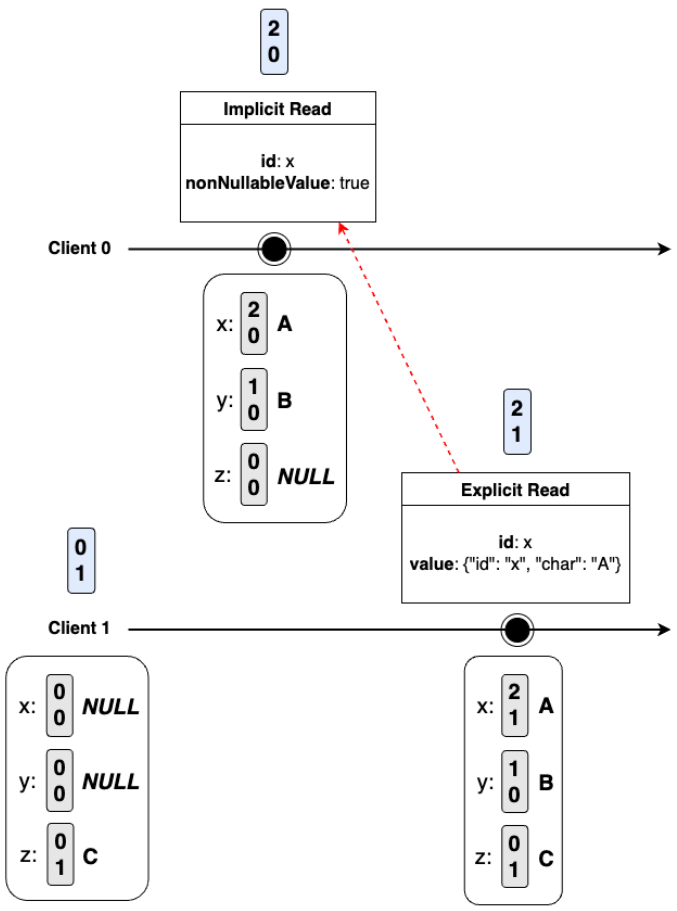

Figure 3.9: Merging process example over the timeline of a portion of a history of operations

Given the processed operations of a history of REST operations, if there is one graph that manages to justify every single explicit and implicit read operation, then the algorithm classifies the history of REST operations as causally consistent.

The algorithm follows a recursive approach, which is based on backtracking. The algorithm may backtrack for two reasons: no justifications available or every justification result in failure further ahead. Whenever a read operation happens to have at least one justification, the algorithm chooses to go forward with one of them (new edge). However, if the next read operation cannot be justified, the algorithm backtracks to the previous one selecting a different dependency (remove old edge, and add a new one), i.e., other possible justification, if there are more of them. The algorithm also keeps track of a global state of the current graph, which includes the clients' data knowledge and a complex data structure that maintains for each *id* of the history of operations a map that records the vector clock that clients assigned to entities they either read or wrote. This way, if a client assigned a particular value of a given *id* to a vector clock, and this global state's structure has that same vector clock pointing to a different value of that same *id*, then the justification chosen for the current operation will not work, and it is necessary to choose another justification, if there are more. This is specially important for the algorithm to ensure that clients converge to the same value given an *id*. Next, all this behaviour is explained in more detail with the help of some examples.

In the beginning, every client starts with the same state and *idProperties*. The state is simply a vector clock full of zeros. Regarding the *idProperties*, for every *id* present in the operations, it is assigned to it a list containing a single element whose *value* is *NULL* and the vector clock is equal to the client's initial state.

Figure 3.10 illustrates the algorithm execution over the processed operations of the history of operations in Figure 3.4. The first implicit read of Client 0 can only be justified by the client itself, since its list of *ValueProperties* of x contains an element that satisfies the operation's property, making it possible to establish its first write. The second implicit read of Client 0 can only be justified by itself as well, so its second write is also established. Regarding the first explicit read of Client 1, the client itself cannot justify it, but the second established write of Client 0 can, since the *id* and entity read correspond to what Client 0 wrote there. This way, both the state and *idProperties* of Client 0 are merged with the knowledge Client 0 had when establishing the write. Finally, the second explicit read of Client 1 cannot be justified neither by itself or Client 1's established writes. The first established write of Client 0 seems to justify, however, the current state of Client 1 is more recent than the one that Client 0 had when establishing the write. Thus, the Client 1's second explicit read has no justifications, meaning that the algorithm backtracks to the previous operation in the timeline, which is the first explicit read of the same client. The algorithm needs to choose another justification for the first explicit read of Client 1, since the second established write of Client 0 resulted in failure, however, there is no more justifications, implying that the algorithm backtracks again to the previous operation in the timeline. The backtrack will eventually reach the beginning of the timeline, because every (implicit and explicit) read operation had only one justification, which resulted in failure, because of recursion. The furthest operation that the algorithm could reach was the second explicit read of Client 1, which is the one that it outputs as the reason the history of operations is not causally consistent.

Figure 3.11 shows a possible execution of the algorithm when iterating through a history of operations. Clients 0 and 2 start by performing concurrent writes, whose implicit reads can only be justified by the clients themselves, respectively, and both writes can be successfully established. Then, Client 2 attempts to update the entity identified by x, whose implicit read can be justified by either itself or Client 0's established write, since both pieces of knowledge contain an entity different than *NULL*. The algorithm chooses to justify through the client itself, and this second write can also be established. After this, Client 1 issues an explicit read which can be justified by either the Client 0's first established write or Client 2's second established write. For the sake of this specific example, let us assume the algorithm chooses to justify it first through the Client 2's second established write. Finally, the only justification for second explicit read of Client 1 would be Client 2's first established write, however, the current state of Client 1 is more recent than the one when Client 2 established that write. Therefore, the algorithm backtracks to

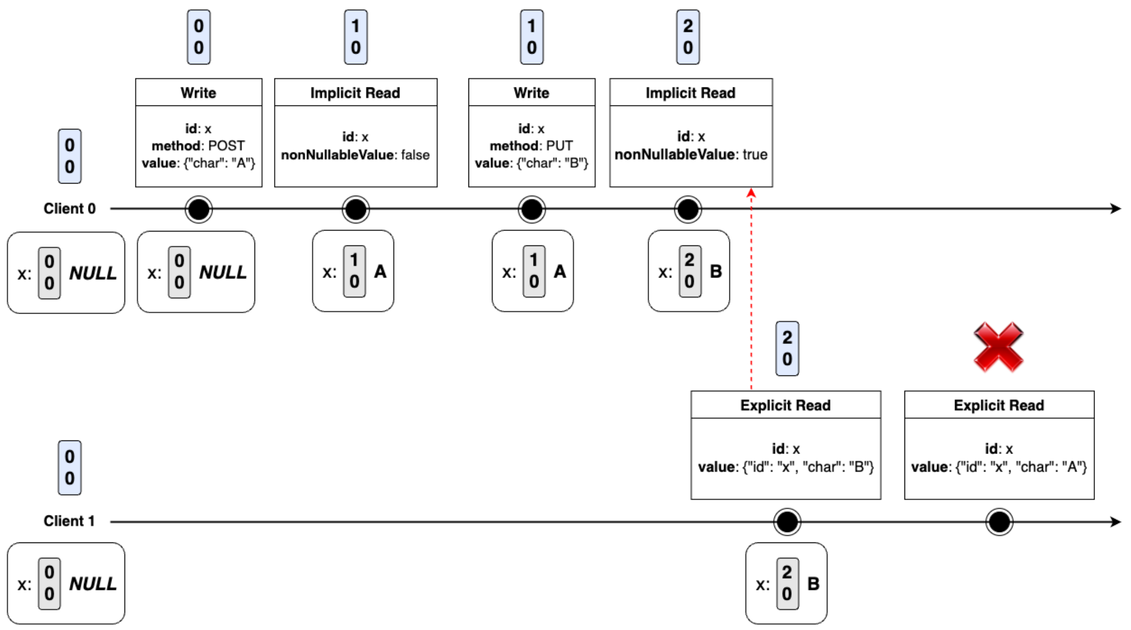

Figure 3.10: Monotonic Writes violation of the timeline of a history of operations considering the algorithm's execution

the previous operation and chooses to justify the value read with the other option available (Client 0's first established write), where Figure 3.12 shows that situation. This way, it is now possible to justify Client 1's last explicit read, and since there are no more operations to go through, the algorithm finishes and classifies this history as causally consistent.

Figure 3.13 shows the situation when the algorithm could justify every read operation, however, the same vector clocks was assigned to two different entities of the same *id*. Both clients start by performing writes whose implicit reads can only be justified by the clients themselves, respectively, and both writes are successfully established. The explicit read of Client 0 can only be justified by Client 1's established write, where the vector clock [1 1] is assigned to the value with the *char* equal to *B*. Finally, the explicit read of Client 1 can only be justified by Client 0's established write, resulting in the same vector clock. However, since the value read is different from the one stored at that vector clock, the algorithm needs to backtrack. Every previous read operation had a single option for justification, i.e., it is not possible to create new branches to the graph, therefore the algorithm goes back to the start point and classifies the execution corresponding to this history as not causally consistent. In fact, this execution satisfies causal consistency, however, it does not satisfy causal+ consistency [43]. Causal+ consistency has all the rules that causal consistency maintains with the addition of the convergence rule, where clients must converge to the same value given an entity. In this case, clients are not converging to the same value of *x*.

Thus, if the algorithm classifies a history of operations as not causally consistent, it is because the corresponding execution violates either causal consistency or causal+ consistency.

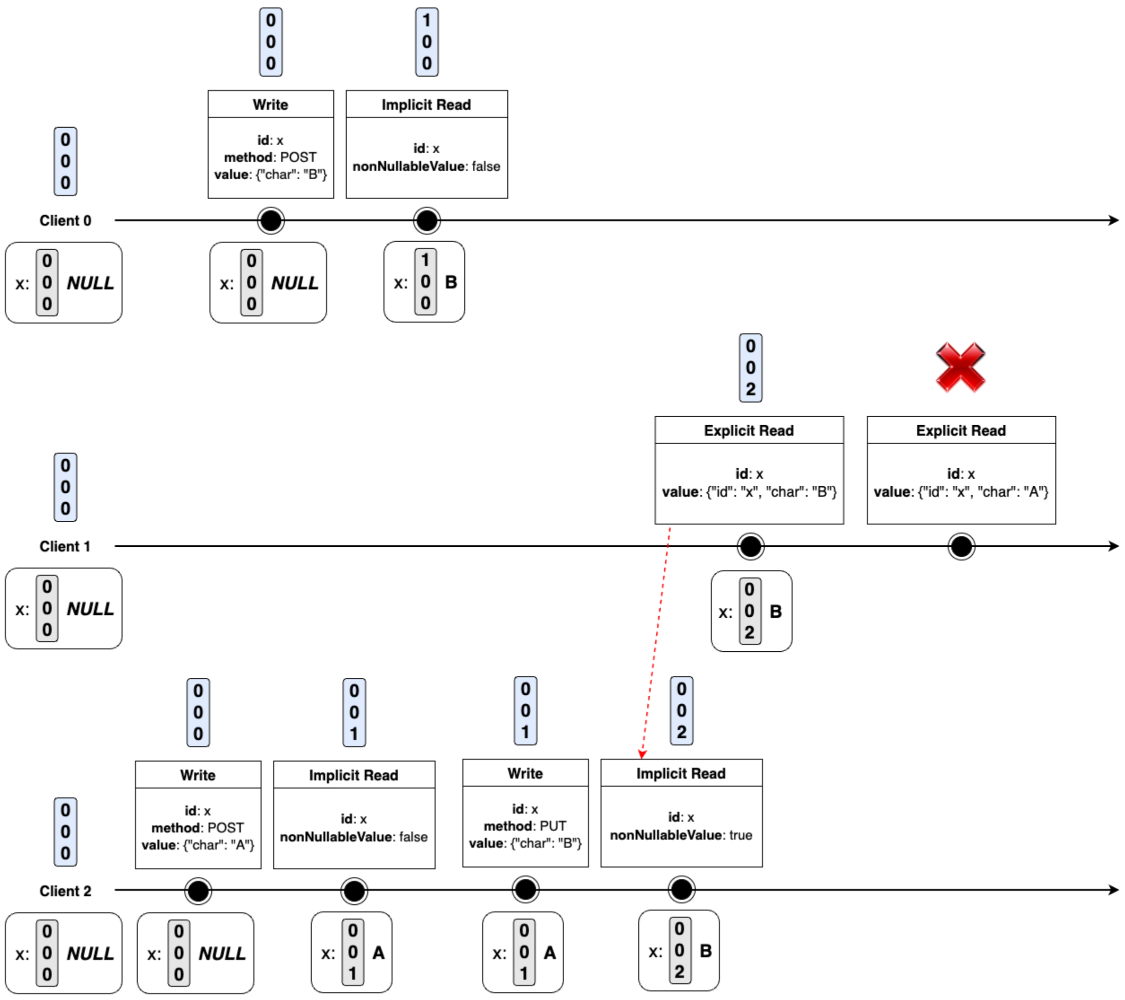

Figure 3.11: Not causally consistent execution of the algorithm over the timeline of a history of operations

3.3.5 Additional Write Vector Clock

There are still certain situations that are not being detected by the algorithm. Figure 3.14 illustrates the algorithm execution over the timeline of operations, which is considered one of those situations. Client 0 performs two consecutive writes over x, assigning to the *char* field the values A and C, respectively. Client 1 issues a write also over x, assigning B to its *char*. Client 1 reads the value of x which is justified only by the first established write of Client 0. Then, it creates the entity y assigning Z to its *char*. Finally, Client 0 reads y, which is justified only by the Client 1's last established write. This history is causally consistent and the algorithm classifies it that way. However, let us assume that Client 0 had issued another reading of x, which returned a value with the *char* set to A. In that case, the history would not be causally consistent, because Read Your Writes would be violated, given that Client 0 wrote C after writing A. However, the algorithm would consider the history causally consistent, where that read operation could be justified by the client itself, since the list of *ValueProperties* of x contains an element whose value (*char* field) is A, which

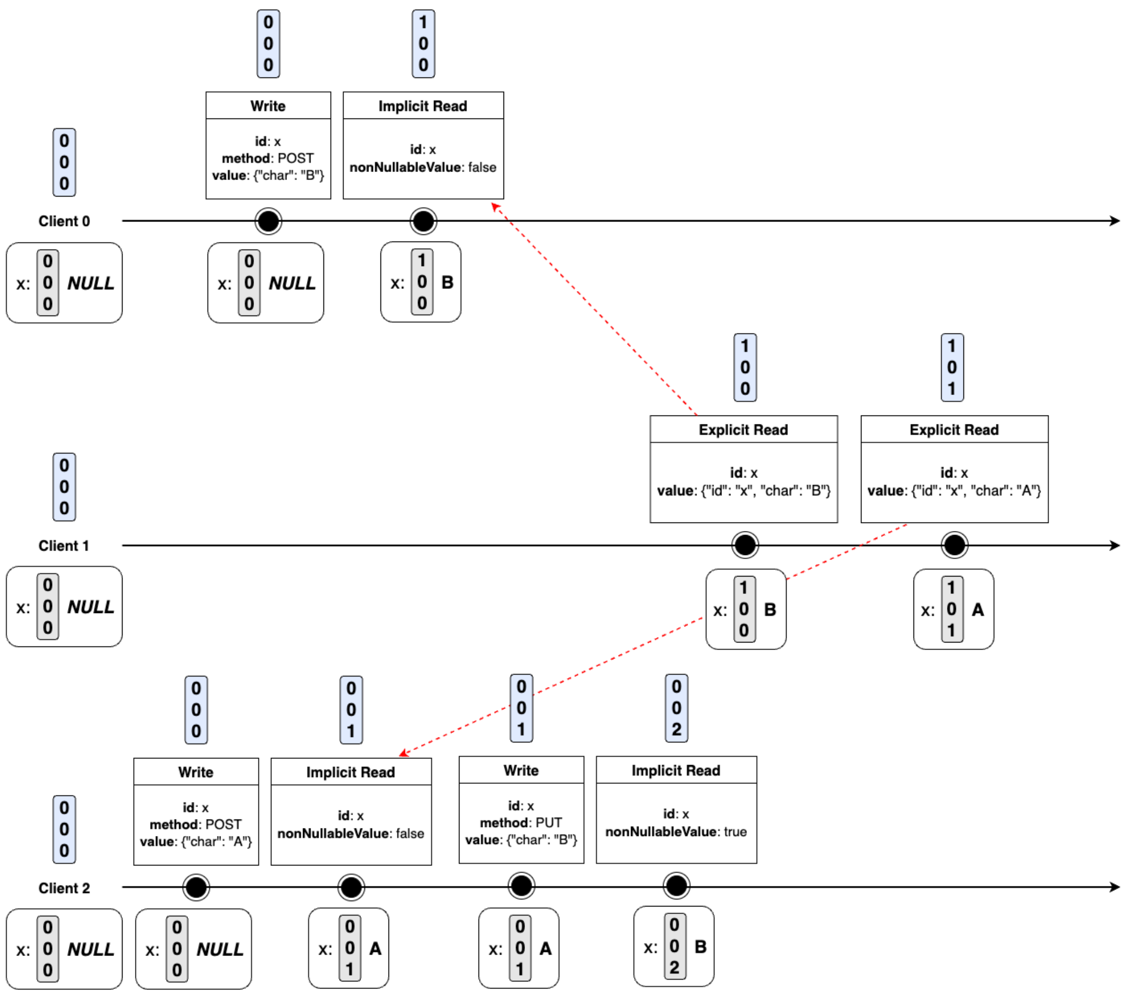

Figure 3.12: Causally consistent execution of the algorithm over the timeline of a history of operations

is pointed out with the red circle.

This situation was created, because Client 1 performed a write operation before reading *x*, which influenced the vector clock of *A*. If no write was performed, its vector clock would be [1 0], which would be discarded in the merging process of the last read of Client 0, since [2 0] is more recent than [1 0]. Therefore, not only it is necessary to merge the *idProperties* by the vector clocks that we have been studying, but also by the write vector clocks. This new vector clock reflects the version when the client established the write for that value. This way, the merging process merges the elements by the usual vector clock and write vector clock (regardless of whether the former ones are concurrent). A new write vector clock only arises upon the establishing of writes, which coincides with the client's state.

Figure 3.15 shows the algorithm execution using write vector clocks over the same timeline of operations. The red vector clock of each *ValueProperties* element represents the new write vector clock. The *ValueProperties* element of *x* with the value *A* and usual

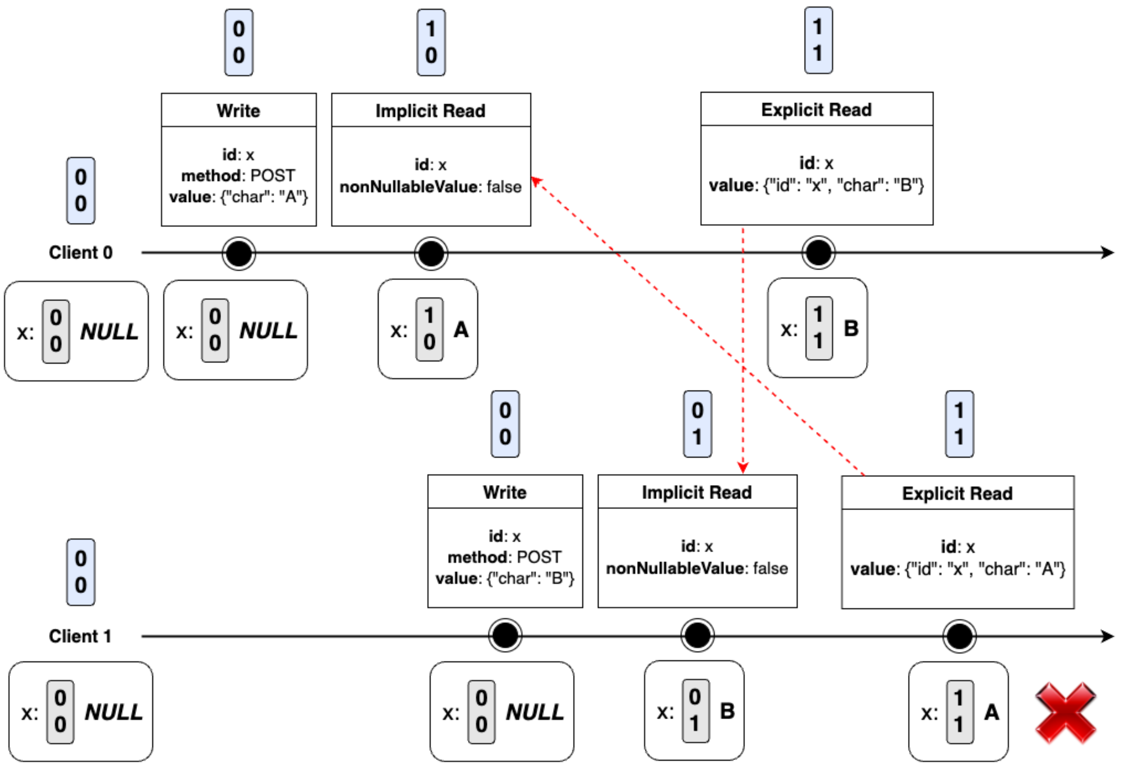

Figure 3.13: Same vector clock pointing to different values of the same *id* violation of the timeline of a history of operations

and write vector clocks [1 1] and [1 0], i.e., the element pointed out with the red circle, is discarded at the read operation of Client 0, because the write vector clock of the *ValueProperties* element of *x* with the value *C* is more recent than this element's, even though the usual vector clocks are concurrent. So, it is guaranteed that Client 0 cannot justify by itself the reading of *x* with the *char* of *A*.

3.3.6 Pending Writes

So far, we have been analysing only timelines of operations, whose operations are justified by established writes, i.e, given an implicit or explicit read operation, these use another client's write as justification whose corresponding implicit read has been justified, making it possible for that client to establish the write. For instance, we have not yet considered the algorithm's behaviour of the timeline in Figure 3.7. Client 1's explicit read depends on Client 0's write that has not been established yet, since the corresponding implicit read operation happens after Client 1's read, meaning that the establishment knowledge of that write has not been created yet. These situations are very common among histories, which result from the concurrency among clients.

Although, it might happen that (implicit and explicit) read operations depend on writes that have not been established yet, i.e, given the write the read operation depends on, this read operation happened before the implicit read of that write, where these read

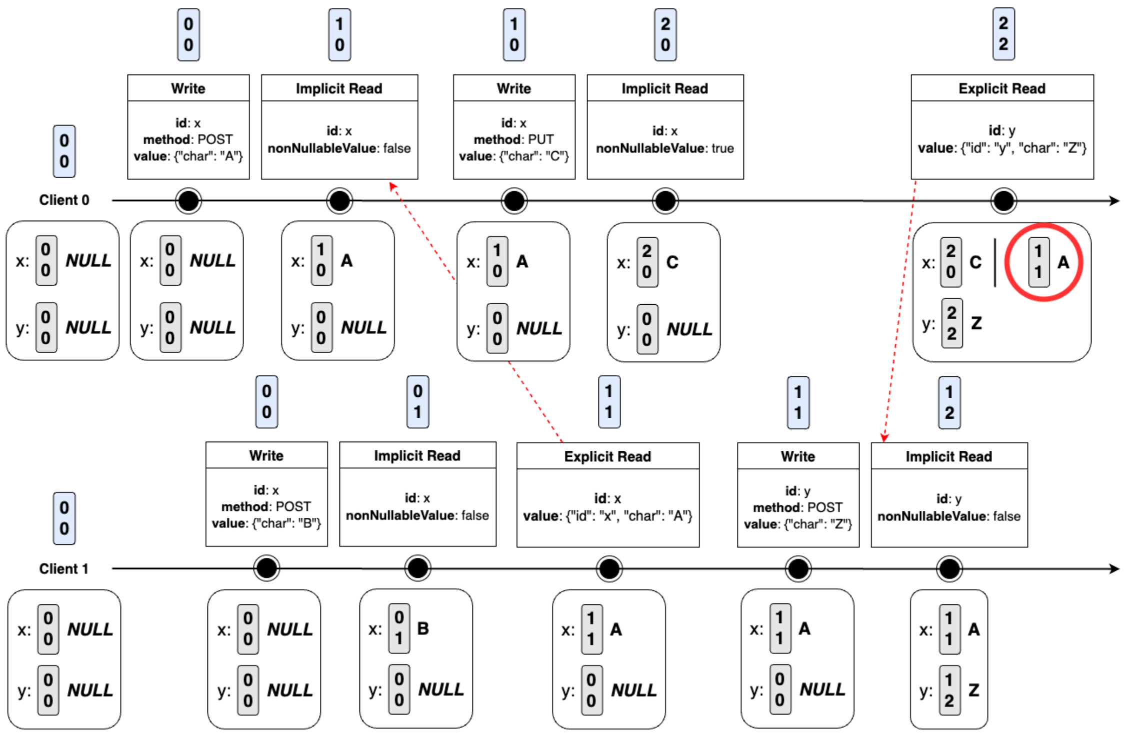

Figure 3.14: Timeline of a history of operations that demonstrates the need to use an additional write vector clock

operations do not have access to the establishment knowledge of the write, since it has not been created yet.

Whenever the algorithm executes a (processed) write operation, this write enters the state of *pending* until reaching the correspondent implicit read operation. The *pending* state means that the client responsible for this write has not established it yet, however, it might be visible for the rest of the clients. If the algorithm tries to justify an operation through a write that is in this state, then this operation can only be performed, when the *pending* write exits this state and becomes established.

An explicit read can be justified by a pending write if the *id*s and values coincide. As for an implicit read to be justified by a pending write, not only the *id*s need to be the same, but also the HTTP methods of the writes in context must be taken into consideration. If the implicit read corresponds to a write that performed a PUT or a DELETE, the only *pending* writes that might justify the existence of the *id* are the ones that comprise a POST, because something had to create it for it to exist. However, if the implicit read corresponds to a write that performed a POST, the only *pending* writes that might justify the nonexistence of it are the ones that have performed a DELETE, and the reason is the exact opposite of the previous one, since something needs to delete an *id* for it not to exist.

Figure 3.16 shows a simple example of this type of situations, which illustrates the execution of the algorithm until the Client 1's explicit read. In this example, an explicit read depends on a *pending* write. In order for the explicit read to take place, it is necessary

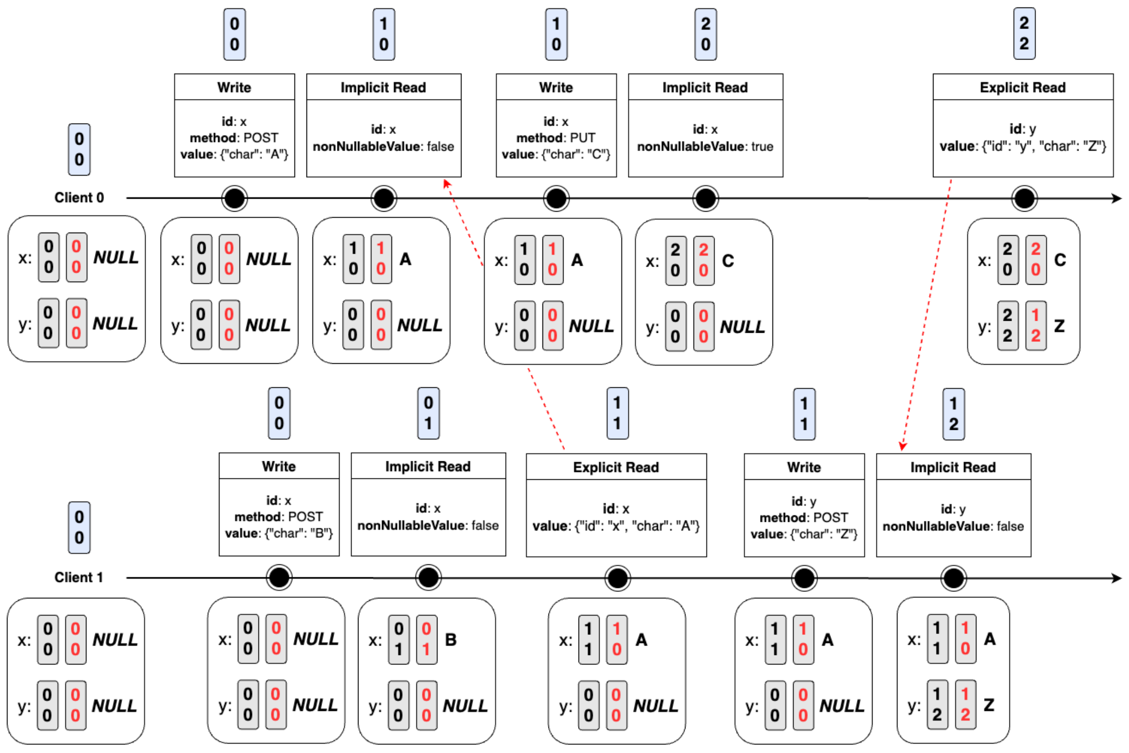

Figure 3.15: Timeline of a history of operations that uses the additional write vector clock

that the write becomes established first. The algorithm sets the state of Client 1's explicit read to be *onStandBy* and associates it as a dependency of the *pending* write, so when it gets established (at the implicit read operation), it can be possible for the explicit read to merge the Client 1's data knowledge with the new established write knowledge. The only justification for the implicit read of Client 0 is the client itself, since its list of *ValueProperties* of *x* contains an element whose value has the *char A*. Figure 3.17 illustrates the final knowledge of both clients after treating the implicit read operation and establishing the corresponding write.

Figure 3.18 illustrates a more complex example of this type of situations, showing the final knowledge of the three clients, i.e, considering the full execution of the algorithm. Client 0 issues a write that comprises a POST operation, Client 1 also issues a write, but comprising a PUT operation, and Client 2 reads *x*, which coincides with the value written by Client 1. Both writes of Clients 0 and 1 are in the *pending* state when the algorithm tries to go through the explicit read of Client 2. The only justification for this read is the *pending* write of Client 1, and this way the algorithm points the read as a dependent of that write. In turn, the implicit read of Client 1 can only be justified by the *pending* write of Client 0, becoming a dependent of it. Finally, the only justification for the implicit read of Client 0 is the client itself, since its list of *ValueProperties* of *x* contains an element whose value is *NULL*. Thus, Client 0's write can be established, and its dependent (Client 1's implicit read) can access that knowledge, merging it and establishing its write. In turn, the dependent of Client 1's write (Client 3's explicit read) can access its establishment

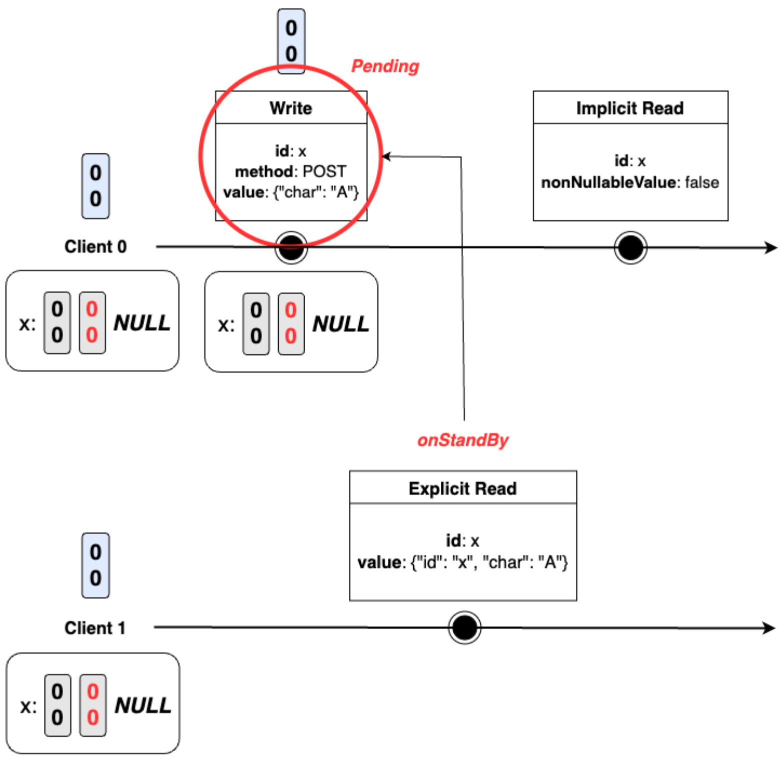

Figure 3.16: Timeline of a history of operations with an explicit read that is justified by a pending write, considering the algorithm's execution until Client 1's explicit read operation

knowledge and merge it.

Whenever a write is established, its dependencies (if they exist), which are explicit reads or implicit reads, must be treated. If a dependency is an implicit read whose corresponding write also contains dependencies, they must also be treated after it establishes the respective write, and so on. The algorithm relies on depth first search to handle this.

Assuming the current operation is an implicit read that depends on a *pending* write and the next operation to be executed is from that same client, both operations will remain in the state of *onStandBy*. The former one is on that state, because it can only be treated after the *pending* write transforms into an established write. The latter one might be able to store the knowledge of the justification, however, it can only be merged when the previous operation (implicit read) gets treated. From now on, whenever the depth first search process is mentioned, we are referring to this process of providing the dependencies of a *pending* write its establishment knowledge, when it gets established, and trying to treat as most *onStandBy* operations as possible. This behaviour will be explained in more detail in the next chapter.

3.3.7 Pseudocode

The algorithm sets up its initial global state consisting of no pending writes and clients having an initial state (array full of zeros), and for each *id* present in the history they store a

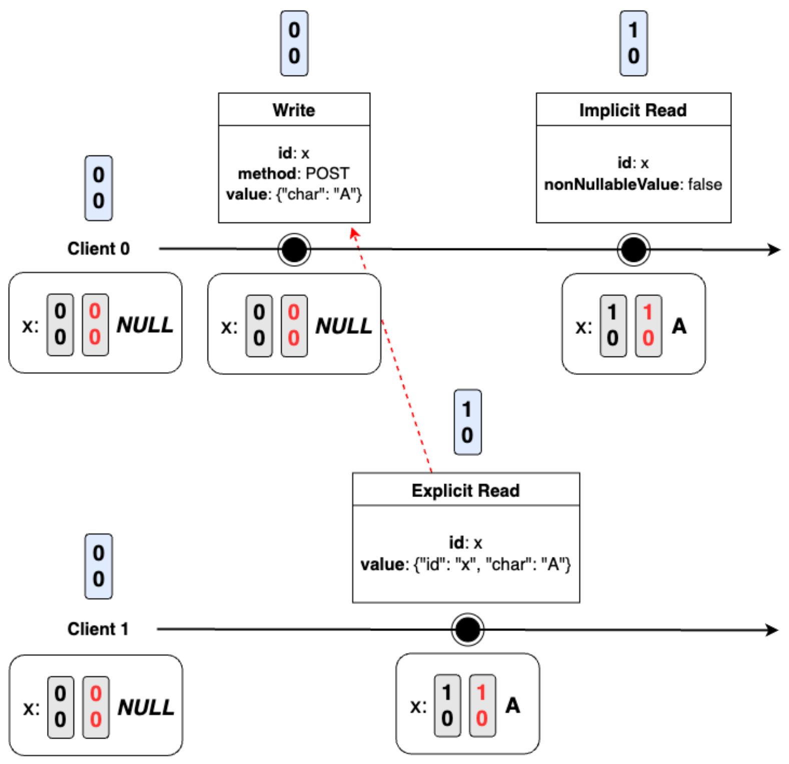

Figure 3.17: Timeline of a history of operations with an explicit read that is justified by a pending write, considering both clients' final knowledge

list of *ValueProperties* containing a single element whose vector clock is equal to the client's initial state and value set to *NULL*.

The pseudocode of the main recursive function, named *check*, is shown in Algorithm 1. It all starts by calling this function with the parameters 0 and the initial global state described above. This function handles the (processed) operation identified by its *operationIndex* with the context of the given *globalState*. Lines 2-3 depict the base case, which happens when the *operationIndex* is equal to the size of the list of operations, meaning that the algorithm have successfully found a graph over these operations that respect causality. Depending on the type of operation (write, implicit read or explicit read), a different function is called to handle the given operation, and lines 4-12 demonstrate that. Any of these functions receive the following parameters: the index of the operation (*operationIndex*), the operation itself, the client that performed the operation and the current global state (*globalState*). Whenever the algorithm chooses a justification for the current operation, these functions will call *check* again with the next operation (*operationIndex* + 1) and a new *globalState* that considers that justification.

The *handleWrite* function handles write operations and its pseudocode is shown in Algorithm 2. The given write enters in the *pending* state (lines 2-3) and the client records this operation as an *onStandBy* write (line 4). When calling the recursive function, *check*, the *globalState* sent contains all this new information (line 5).

The *handleImplicitRead* function handles implicit read operations and its pseudocode

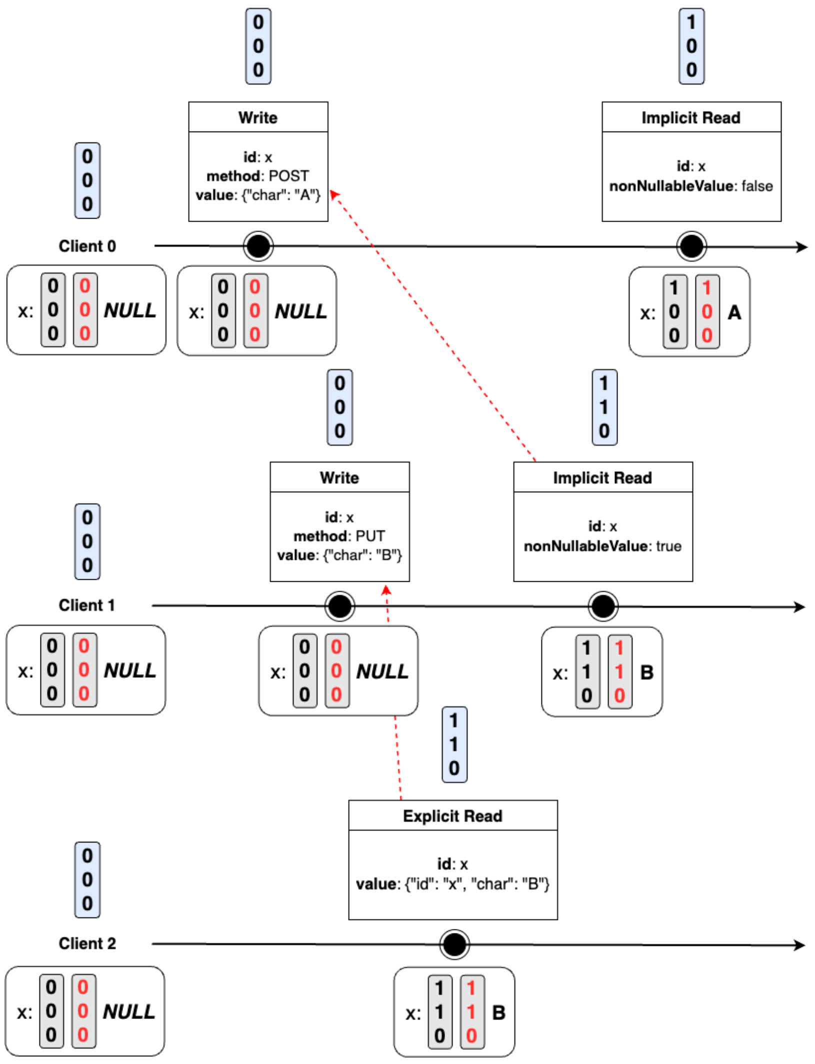

Figure 3.18: Timeline of operations with a write whose correspondent implicit read is justified by a pending write, considering all clients' final knowledge

is shown in Algorithm 3. Since an implicit read has a corresponding write operation, the function makes this write to exit the *pending* state (line 2) and tries to justify the implicit property, which depends on the HTTP method of the write. Whenever the algorithm chooses a justification to apply, it creates a copy of the global state, so that the original global state remains the same in case the justification it chose results in an non causal execution (backtracking). Thus, another justification can be applied starting with the same original global state and not the one that suffered changes from other justification. The algorithm can try to justify the implicit property through the client itself, valid pending writes and valid established writes. First, it tries to justify through the client itself (lines 4-8). When calling the function *completeWriteWithSameClient*, if the client has *onStandBy* operations prior to the correspondent *onStandBy* write, it will remain in the same state until the client's previous operations get treated, returning *true* and an empty map structure, since no new pairs of values and vector clocks have been assigned to any *id*s. Otherwise,

Algorithm 1: Pseudocode of the recursive function

```
1  Function check(operationIndex, globalState):
2      if operatationIndex == operations.size() then
3          return true
4      else
5          operation ← operations.get(operationIndex)
6          client ← globalState.getClient(operation.getClient())
7          if isWrite(operation) then
8              return handleWrite(operationIndex, operation, client, globalState)
9          else if isImplicitRead(operation) then
10             return handleImplicitRead(operationIndex, operation, client,
                 globalState)
11         else
12             return handleExplicitRead(operationIndex, operation, client,
                 globalState)
13         end
14     end
```

Algorithm 2: Pseudocode of the function that handles write operations

```
1  Function handleWrite(operationIndex, operation, client, globalState):
2      pending ← globalState.getPending()
3      pending.put(client.getIndex(), operation)
4      client.addWrite(operation)
5      return check(operationIndex + 1, globalState)
```

it immediately checks if the client is able to justify it. If the client is not able to justify it, or if the depth first search process (after the establishing the write) results in treating any *onStandBy* operation that was supposed to be justified by its client, however, it could not, the first output is retrieved as *false*, otherwise as *true*. The second output consists of the new pairs of values and vector clocks by *id*s that this operation and respective depth first search process generated. It can only proceed to the next operation, once the first output is *true* and the second one is valid, i.e., for a given id, if the global state contains a vector clock that has been generated by this operation, they must point to the same value. If any of these two conditions is not valid or if the recursive call fails, then the algorithm will find the *pending* writes that might justify the implicit property (line 9). For each of these possible justifications (lines 10-16), the algorithm will associate this operation's corresponding write as a dependency of the given *pending* write, proceeding to the next operation. When that *pending* write gets established, this operation's corresponding write is provided with that establishment knowledge through the depth first search process, and it can be treated immediately if there are no previous *onStandBy* operations. The recursive function gets called to treat the next operation. If that results in failure for every valid *pending* write, the algorithm finds the valid established writes for this particular implicit read (line 17). For each of these possible justifications (lines 18-24), the algorithm will feed it immediately to client, however, it can only be merged once the client has no previous *onStandBy* operations. This behaviour and both outputs represent the same ones as when trying to validate through the client itself. It proceeds to the next operation, once the outputs are valid. If any of them is not valid or if the recursive call resulted in failure for every valid established write, then the algorithm has no option but to backtrack (line 25), i.e., inform the previous operation that this one could not be justified, and thus it is necessary to choose another justification for that previous operation, if there are more.

Algorithm 3: Pseudocode of the function that handles implicit read operations

```
1  Function handleImplicitRead(operationIndex, operation, client, globalState):
2      writeOperation ← globalState.getPending().remove(client.getIndex())
3      method ← writeOperations.getMethod()
4      globalStateCopy ← copy(globalState)
5      valid, valuesAndVectorClocksById ← globalState-
         Copy.getClient(client.getIndex()).completeWriteWithSameClient(writeOperation,
         globalStateCopy)
6      if valid & isValid(valuesAndVectorClocksById, globalStateCopy) &
         check(operationIndex + 1, globalStateCopy) then
7          return true
8      end
9      clientsWithValidPendingWrites ←
         getClientsWithValidPendingWrites(operation.getId(), method, globalState)
10     foreach otherClientIndex it clientsWithValidPendingWrites do
11         globalStateCopy ← copy(globalState)
12         globalStateCopy.getPending().get(otherClientIndex)
             .addDependent(writeOperation)
13         if check(operationIndex + 1, globalStateCopy) then
14             return true
15         end
16     end
17     validEstablishedWrites ← getValidEstablishedWrites(operation, client,
         globalState)
18     foreach establishedWrite it validEstablishedWrites do
19         globalStateCopy ← copy(globalState)
20         valid, valuesAndVectorClocksById ←
             globalStateCopy.getClient(client.getIndex())
             .completeWriteWithEstablishedWrite(writeOperation, establishedWrite,
             globalStateCopy)
21         if valid & isValid(valuesAndVectorClocksById, globalStateCopy) &
             check(operationIndex + 1, globalStateCopy) then
22             return true
23         end
24     end
25     return false
```

The *handleExplicitRead* function handles explicit read operations and its pseudocode is shown in Algorithm 4. This function has a similar behaviour to the previous one.

In this case, the algorithm does not search for justifications that satisfy the existence or nonexistence of the corresponding *id*, but justifications that comprise the value read from the given *id*. When considering the client itself (lines 4-8) or established writes (lines 17-24) as justifications, if there are no previous *onStandBy* operations at the given client, then this operation can get treated immediately (merged), otherwise it is created a new *OnStandByOperation* object wrapping this explicit read operation, where the justification is stored and only gets treated once all previous *onStandBy* operations get treated. When considering *pending* writes (lines 9-16) as justifications, regardless if the are previous *onStandBy* operations at the given client or not, it is always created an *OnStandByOperation* object wrapping this explicit read operation and gets treated once the *pending* write gets established and when all the previous *onStandBy* operations get treated. Similarly to the previous function, if all the justifications result in failure, the algorithm backtracks (line 25).

3.3.8 Limitations

We found two patterns of histories of operations the algorithm does not classify correctly, which we consider them as the limitations of this algorithm.

Figure 3.19 shows a timeline of a history of operations that violates causality, however, the algorithm cannot detect it. Both clients start by creating two objects - x and y - where Client 0 assigns the values A and D, respectively, and Client 1 assigns the values B and C, respectively. Next, Client 0 creates Z with the value K. Finally, Client 1 executes three sequential reads. Since, the value C won (against D) for y, Client 0 must only be able to observe the value B for x, however, the algorithms considers valid that the user observes A for x.

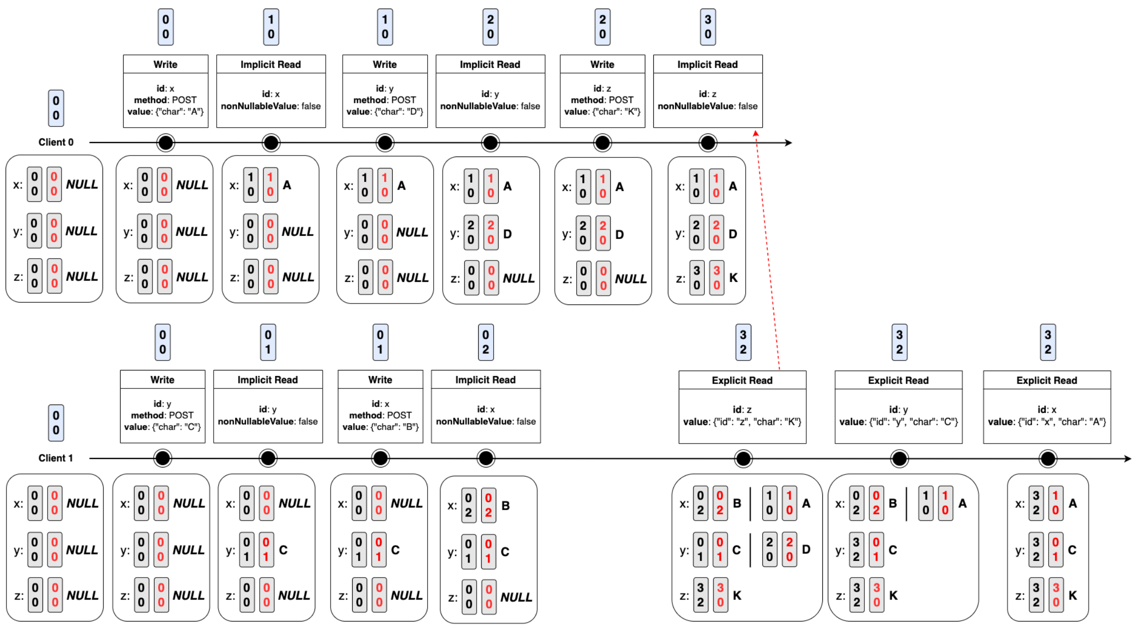

Figure 3.19: Timeline pattern that the algorithm does not verify correctly

Algorithm 4: Pseudocode of the function that handles explicit read operations

```
Function handleExplicitRead(operationIndex, operation, client, globalState):
    id ← operation.getId()
    value ← operation.getValue()
    globalStateCopy ← copy(globalState)
    valid, valuesAndVectorClocksById ← globalStateCopy.getClient(client.getIndex()).addExplicitReadWithSameClient(operation, globalStateCopy)
    if valid & isValid(valuesAndVectorClocksById, globalStateCopy) & check(operationIndex + 1, globalStateCopy) then
        return true
    end
    clientsWithValidPendingWrites ← getClientsWithValidPendingWrites(id, value, globalState)
    foreach otherClientIndex in clientsWithValidPendingWrites do
        globalStateCopy ← copy(globalState)
        valid, valuesAndVectorClocksById ← globalStateCopy.getPending().get(otherClientIndex)
          .addExplicitReadWithPendingWrite(operation, globalStateCopy)
        if valid & isValid(valuesAndVectorClocksById, globalStateCopy) & check(operationIndex + 1, globalStateCopy) then
            return true
        end
    end
    validEstablishedWrites ← getValidEstablishedWrites(operation, client, globalState)
    foreach establishedWrite in validEstablishedWrites do
        globalStateCopy ← copy(globalState)
        valid, valuesAndVectorClocksById ← globalStateCopy.getClient(client.getIndex())
          .addExplicitReadWithEstablishedWrite(operation, establishedWrite, globalStateCopy)
        if valid & isValid(valuesAndVectorClocksById, globalStateCopy) & check(operationIndex + 1, globalStateCopy) then
            return true
        end
    end
    return false
```

As the vector clocks associated to the object x did not change when the y's ones changed, the algorithm considers this behaviour to be valid.

Figure 3.20 shows another timeline of a history of operations that contains a convergence violation (causal+ consistency), but the algorithm is not able to detect it. The three clients start by creating a new entity x, where each assigns a different *char* to it - Client 0 wrote A, Client 1 wrote B and Client 2 wrote C. Then, Clients 1 and 2 create another entity y, assigning the *chars* I and J. After that, Clients 0 and 1 read y having the value with the *char* set to *I*. Finally, Client 0 reads *y* again, but this time the value has the *char* set to *J*. This last operation is the cause of the convergence violation, since regarding *y*, Client 0 sees *I* and then *J*, but Client 2 wrote *J* and then read *I*.

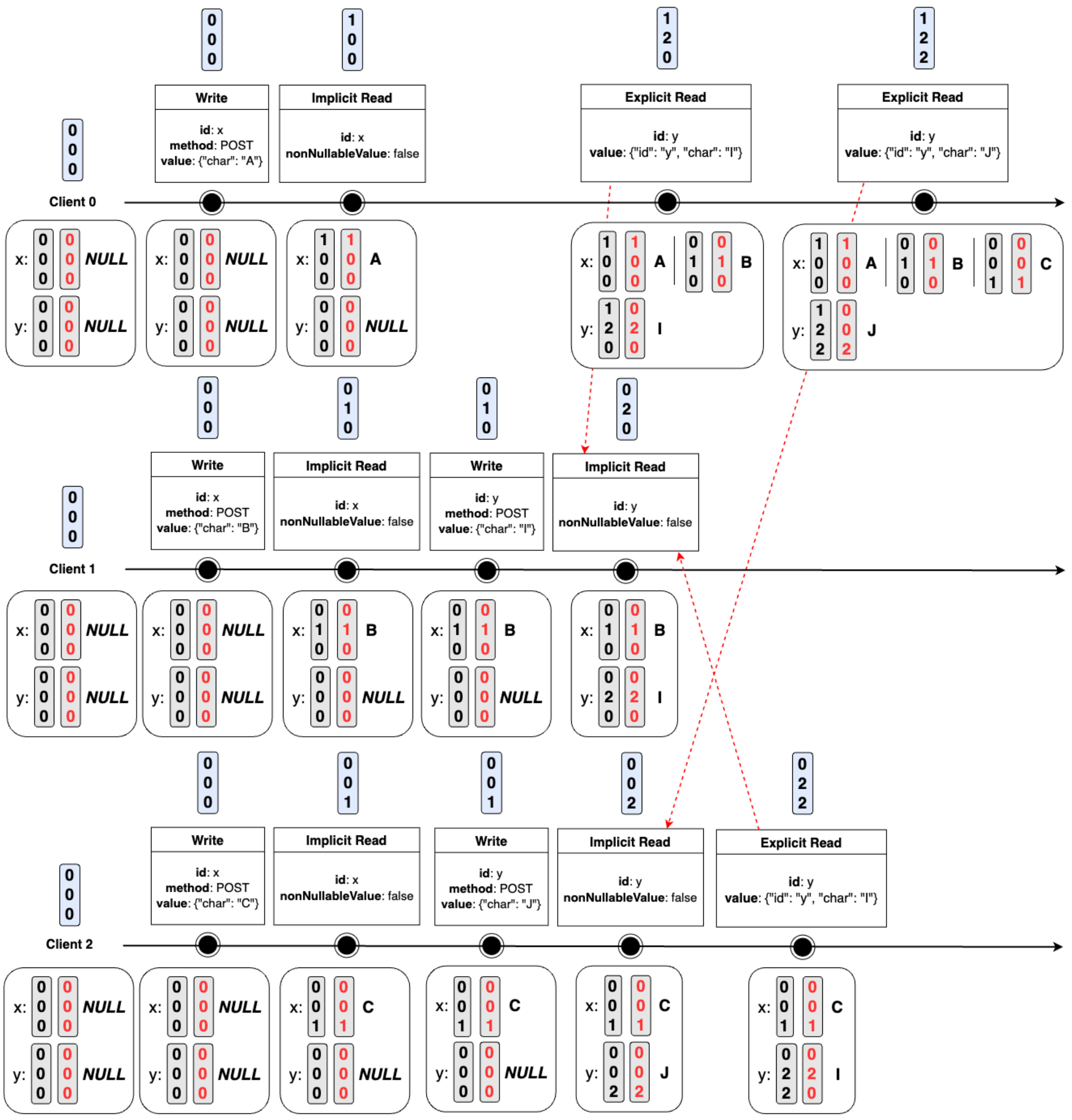

Figure 3.20: Timeline pattern that the algorithm does not verify correctly

The algorithm would detect this violation if the (version) vector clock of the *ValueProperties* element of *y* at Client 0's last explicit read was equal to the (version) vector clock of *ValueProperties* element of *y* at Client 2's last explicit read, since the same (version) vector clock would be pointing to different values (*J* and *I*).

We leave both of these corner cases to be handled in future work.

4 Implementation

This chapter starts by briefly presenting the implementation of the components of JepREST. Finally, it presents in detail the implementation of the algorithm that verifies causal consistency in RESTful applications regarding initial processing, global state and clients.

4.1 JepREST

This section presents an overview of the tool that was extended by this dissertation's project.

JepREST [58] is a tool that intends to simplify the testing process of distributed applications with REST interfaces, based only on their API description (the application itself is seen as a *black-box* system). This tool is responsible for verifying the correctness of a REST application while being subject to functional tests, where multiple clients are concurrently performing requests with the possibility of occurring failures in the system's components. A system is said to have a correct behaviour if its executions are linearizable [34]. This tool was developed on top of Jepsen [36], which does not support the testing of REST applications. JepREST consists of four components:

1. Specification of the application's API and semantics that the developer wants to test
2. Specification of the workloads to be performed on the application
3. Execution of the workloads using Jepsen library
4. Analysis of the results, verifying if the execution corresponds to a correct behaviour

Next, each component is described in more detail.

4.1.1 Application's API and Semantics Specification

JepREST needs to have prior knowledge about the API that the developer wants to test. This knowledge includes information of every resource, and every operation that can be performed on them. The developer needs to provide these information through three types of Java files: a base document, interface document(s) and object document(s). These need to include annotations from OpenAPI, JAX-RS, *javax.validation.constraints* and some custom ones from JepREST.

The base document is an interface that defines the base URI of every service's resource. This interface needs to extend the interface document(s) referring to those resources. An interface document is an interface of a service's resource, which contains information of the operations that can be performed on the resource. An object document is a class that represents an object to be created, where the annotations contained define how its data can be generated. Object documents are used by the interface documents to define the parameters, and body of requests and responses of operations.

JepREST relies on methods and data structures provided by Jepsen in order to verify these REST services. However, Jepsen library is written in *Clojure*, whereas the documents described previously that hold the specification of the API are written in Java. Therefore, a code generated was developed, written in Java, that processes these documents and generate *Clojure* methods that will be executed by the Jepsen library.

Jepsen's "Client" data structure only supports clients that submit simple *read* and *write* requests. Thus, it was necessary to create a new client, "ClientREST", which can submit any REST request, namely POST, GET, PUT, PATCH and DELETE.

Figure 4.1 illustrates the workflow when a "ClientREST" executes a request. A "ClientREST" calls the *Clojure* method of the respective operation, which will invoke the REST API with the request. Then, the API will respond back to the *Clojure* method, which in turn will recreate the response in the Jepsen's response format. Finally, this new response is sent back to the "ClientREST" that submitted the request.

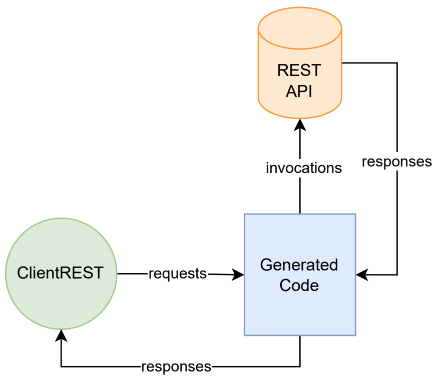

Figure 4.1: "ClientREST" workflow

4.1.2 Workloads Specification

This component complements the previous one, since it defines what workloads the clients can submit to the REST API. Before describing this component, it is important to note that for each operation defined in the documents, the code generator will generate a corresponding *Clojure* method, which will be called whenever a client needs to submit a request corresponding to a operation. Thus, it is necessary to establish a way of choosing the operations parameters, and how to create new *JSON* objects.

Regarding parameters choosing, JepREST will detect every dependency between operations. An operation is dependent on another if it needs to send some parameter that was obtained in the other operation. A list is created for every dependency, where the independent operation appends values to the list, and the dependent will use one of them. If the list is empty, then the dependent one will need to use a random parameter, otherwise a random value from the list is selected to be sent as the parameter. Regarding the creation of *JSON* objects, JepREST will process the object documents in order to check what constraints each field holds. To generate them, two libraries are used: Faker [28] and PETIT [55]. However, JepREST relies mostly on Faker.

Having those methods generated, then this component can execute. The developer needs to provide the workloads through a YAML script. This script contains one or more test scenarios that will be executed on the REST application. A test scenario is composed by a name, a weight, and a sequence of operations. The name represents the scenario's context, whereas the weight dictates its execution frequency. The sequence of operations is defined by using the generated methods names. Listing 4.1 shows an example of this YAML script.

```yaml
scenarios:
  - name: 'Students Management API'
    weight: 100
    flow:
      - createStudentData
      - getStudentData
      - updateStudentData
      - deleteStudentData
```

Listing 4.1: Example of a YAML script provided to JepREST that specifies the scenarios to be executed on a REST API

4.1.3 Workloads Execution

Given the YAML script, this component is responsible for informing the "ClientRest"s which requests they must execute. This process is depicted in Figure 4.2. First, a random number is generated, which determines which test scenario will be submitted, based on the weights, in that instant. Then, the sequence of operations of the chosen scenario is sent to the Jepsen's operations generator. This generator, in turn, sends each of the *Clojure* methods to a "ClientREST" that will submit the request of that method to the REST API. Thus, requests will be submitted by different clients concurrently. This workflow is executed multiple times during the time established for the test process, where, after submitting a scenario, there will be a waiting time before invoking the new chosen scenario.

JepREST supports testing on APIs running on standalone servers and Kubernetes.

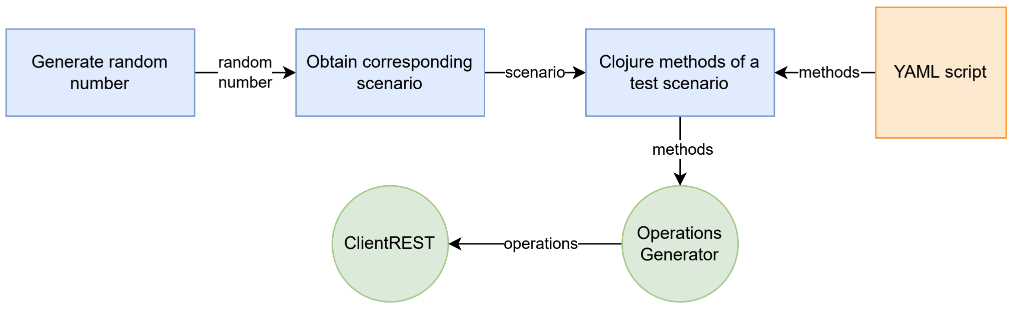

Figure 4.2: Process of informing clients of the requests they must submit

4.1.4 Execution Results Assessment

This component is responsible for verifying the correctness of a REST API. During a test, JepREST generates a file that contains the history of operations, i.e., every submitted request by "ClientREST"s and the responses to these requests. For each operation, this file stores the request type (invocation or a response as "ok" or "fail"), operation type (post, get, put, patch or delete), timestamp, process/client identifier, index (position in history), and additional request/response information (parameters and body).

The history of operations is analysed by this component in order to determine if the execution of the REST API corresponds to a correct behaviour. To this end, one resorts to a Jepsen checker, called Knossos, which defines that the history represents a correct behaviour of the API if it is considered linearizable, given the model defined. The model represents the API state, which expresses the way it is changed given the occurrence of REST operations. Thus, it verifies if it is possible to have a linearization of the operations in which the state of the model matches the responses obtained from the API. If the operations responses are indeed consistency, then Knossos informs the developer that the history is linearizable. Otherwise, it returns an inconsistency problem.

Knossos executes until detecting the first potential linearizability issue. When detecting it, Knossos terminates its history analysis process and returns to the developer information related to the operation that it was not able to linearize.

JepREST created a new model that implements the interface "Model" offered by Knossos. This was necessary simply because Knossos's models can only represent database states that contain simple elements, such as registers. This implies that they cannot represent the state of REST applications, where rather complex elements need to be stored, such as *JSON* objects. Therefore, the developed model aims to define the state of a REST system, managing the multiple *JSON* objects that are used during the system's executions.

4.2 Algorithm

This section presents how the main parts of the algorithm were implemented.

JepREST's checker, Knossos [40], verifies linearizability in RESTful applications, while our algorithm is a brand new checker that verifies causal consistency in those same applications. Our algorithm simply takes advantages of the (Jepsen) semantics and syntax history files generated by JepREST. Thus, these checkers have no correlation with each other whatsoever. Figure 4.3 shows exactly this behaviour.

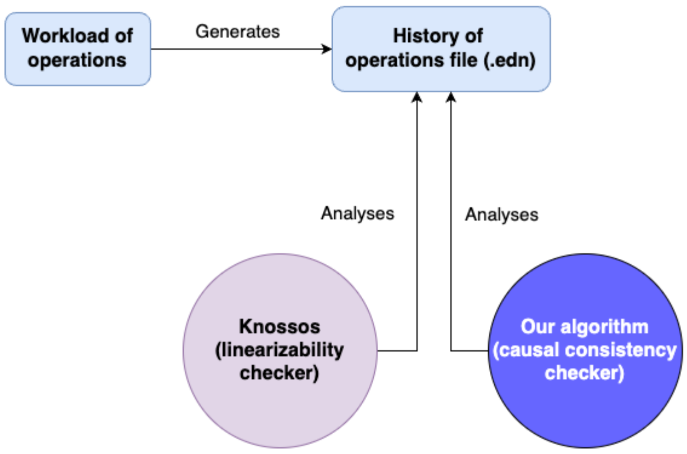
Workload of operations
Generates
History of operations file (.edn)
Analyses
Analyses
Knossos (linearizability checker)
Our algorithm (causal consistency checker)

Figure 4.3: How our algorithm takes advantage of JepREST

The algorithm itself was written in Java. However, it was necessary to do an initial processing of the Jepsen history, which was written in a simple Clojure script.

4.2.1 From Jepsen History to *JSON* History

The file from Jepsen that consists of the history of operations performed on the workload is provided as an *EDN* file. Listing 4.2 illustrates a portion of an example of this file. Instead of directly feeding this file to the Java implementation of the algorithm, we found it advantageous to extract some information out of it and convert this file to a *JSON* file for better compatibility with the Java implementation of the algorithm, and scaling of the whole project.

```
{:type :invoke, :f :post, :value {:input {:json {:char "A"}, :typeOp "createObject"}}, :
    time 1662461060, :process 0, :index 0}
{:type :ok, :f :post, :value {:input {:json {:char "A"}, :typeOp "createObject"}, :
    output {:status 201, :body {:id "x", :char "A"}}}, :time 1662461062, :process 0, :
    index 1}
```

Listing 4.2: Portion of an example of the Jepsen file (*EDN*) that contains the history of operations performed by the workload

As mentioned in Section 3.2.2, the processing mechanism of transforming the history entries of this file needs to know where the corresponding response's history entry of a given request's history entry sits in order to be efficient. A simple Clojure project was created to extract this information and store it in the *opposite-index* field in order to avoid this additional overhead on the Java implementation, and also transform the final data into a *JSON* file, which is the one read by the Java implementation. Listing 4.3 shows the *JSON* portion of the history corresponding to the previously seen *EDN* file portion.

```
[
  {
    "type": "invoke",
    "f": "post",
    "value": {
      "input": {
        "json": {
          "char": "A"
        },
        "typeOp": "createObject"
      }
    },
    "process": 0,
    "index": 0,
    "opposite-index": 1
  },
  {
    "type": "ok",
    "f": "post",
    "value": {
      "input": {
        "json": {
          "char": "A"
        },
        "typeOp": "createObject"
      },
      "output": {
        "status": 201,
        "body": {
          "id": "x",
          "char": "A"
        }
      }
    },
    "process": 0,
    "index": 1,
    "opposite-index": 0
  }
]
```

Listing 4.3: Portion of the *JSON* file that corresponds to the history portion in Listing 4.2

4.2.2 From Raw Operations to Processed Operations

Having access to the new *JSON* file of the history of operations, it is necessary to read its content, which we call *raw* operations, and convert it to the processed operations syntax our algorithm relies on, i.e., write, implicit read and explicit read operations.

Listing 4.4 shows the function that converts raw operations to the corresponding processed operations. First, it obtains the list of raw operations (line 2), where the Gson [33] library was used to deserialize them to Java classes. All the processed operations will be stored in the *processedOperations* list (line 3), and all the operations' *id*s will be stored in the *ids* set (line 5). Clients have sequential identifiers, starting at zero, and the variable *clients* (line 4) will hold the largest of them all, where the total number of clients is $clients + 1$.

Next, we iterate through the raw operations in order to convert them into processed operations (lines 7-27). If the raw operation's *type* is *invoke*, then it is necessary to perform some checks (lines 8-20). If this *invoke* raw operation resulted in failure (lines 12-15), i.e., the correspondent *ok* raw operation's *status* is equal to 404, then the HTTP method (*f* field) of both raw operations is set to GET, regardless of the original one. This is to ensure that the *ok* raw operation is converted into an explicit read operation further ahead. After this, if the *invoke* operation's HTTP method is GET (lines 17-19), this raw operation is discarded and we proceed to the next one.

```
 1 public OperationsInfo processRawOperations() {
 2     List<RawOperation> rawOperations = getRawOperations();
 3     List<Operation> processedOperations = new ArrayList<>(rawOperations.size());
 4     int clients = -1;
 5     Set<String> ids = new HashSet<>();
 6 
 7     for (RawOperation rawOperation : rawOperations) {
 8         if (rawOperation.getType().equals(INVOKE)) {
 9             RawOperation rawResponseOperation = rawOperations.get(
10                 rawOperation.getOppositeIndex());
11             int status = rawResponseOperation.getData().getOutput().getStatus();
12             if (status == 404) { // Set method to be GET
13                 rawOperation.setMethod(GET);
14                 rawResponseOperation.setMethod(GET);
15             }
16 
17             if (!rawOperation.isWriteOperation()) { // Skip invocations of GET operations
18                 continue;
19             }
20         }
21 
22         List<Operation> operations = buildOperation(rawOperation, rawOperations);
23         processedOperations.addAll(operations);
24         Operation operation = operations.get(0);
25         clients = Math.max(clients, operation.getClient());
26         ids.add(operation.getId());
27     }
28 
29     return new OperationsInfo(processedOperations, clients + 1, ids);
30 }
```

Listing 4.4: Convert raw (Jepsen) operations to processed operations

The *buildOperation* function (line 22) receives the raw operation to be converted and the full list of raw operations, and retrieves a list with the new processed operations corresponding to the raw operation. Depending on the *type* and HTTP method (*f* field) of the raw operation, the function's behaviour is the following:

- *invoke* POST and *invoke* PUT - Write operation whose value holds what the client wrote.
- *invoke* DELETE - Write operation whose value holds *NULL*.
- *ok* POST - Implicit read operation whose *nonNullableValue* field is set to *false* and an explicit read operation whose value holds what the client wrote.
- *ok* GET - Explicit read operation whose value holds what the client read.
- *ok* PUT - Implicit read operation whose *nonNullableValue* field is set to *true* and an explicit read operation whose value holds what the client wrote.

- *ok* DELETE - Implicit read operation whose *nonNullableValue* field is set to *true* and an explicit read operation whose value holds *NULL*.

The new processed operations are added to the *processedOperations* list (line 23). Next, the largest client's identifier so far is obtained (line 25), and the *id* of these processed operations is added to the *ids* set (line 26).

Once iterating through every raw operation, this function returns a structure that contains all the new processed operations, and the number of clients and all the *id*s in the history of operations (line 29).

4.2.3 Global State

The global state manages all the information from a given graph (when choosing specific justifications). That information consists of the properties and current knowledge of each client, every values and vector clocks that have been assigned to the *id* and the *pending* writes.

Listing 4.5 shows the initial part of the Java class that represents the global state. The client objects are stored in a List. A map structure stores for every *id* another map that associates a value that has been either written or read to corresponding vector clock. The *pending* writes are associated to an integer that represents the corresponding client's *id* also using a map structure.

```java
public class GlobalState {

private final List<Client> clients;
private final Map<String, Map<String, Object>> valueByVectorClockById;
private final Map<Integer, WriteOperation> pending;
```

Listing 4.5: Initial part of the Java class that represents the global state

Whenever the algorithm chooses to advance with a particular justification for either an implicit or explicit read, it is necessary to copy the global state and only apply the justification on this copy, otherwise we would lose the original global state, and therefore when other justifications of the same operation were considered, it would not be the original state that was being copied, but the one that considered the previous justification, leading to a invalid behaviour. The new copy of the global state, which includes the chosen justification, is the one that is sent to the recursive function call.

In fact, the copy that the implementation performs is characterized as a deep copy, because the copy should not change the references that the original contains. The new global state object (copy) contains the exact same properties, however, they must have different references. This behaviour was achieved through the usage of Gson [33]. The deep copy process consists of serializing the whole global state object to a *JSON* string and then deserialize it to a new global state object. Thus, it is guaranteed that the original object and the copy do not share any references (both as objects and their properties). Figure 4.4 shows this copy process.

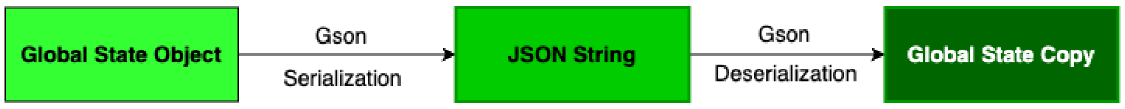

Figure 4.4: Process of copying the global state

4.2.4 Clients

Clients are one of the most important and complex structures of the implementation. A client object stores its *index*, its established writes, its state, its *idProperties* and its operations that are *onStandBy*.

Listing 4.6 shows the initial part of the Java class that represents a client. The *index* is an integer that identifies the client and its established writes are stored in a list of *EstablishedWrite* objects, where each contains the state and the *idProperties* of this client after it established the given write that was in the *pending* state. The client state is represented through a vector clock object, which acts as a simple array of integers. Its *idProperties* are stored in a map structure, where a list of *ValueProperties*, with at least one element, is assigned to each *id* present in the history of operations. A *ValueProperties* object stores the object/value (any type) that has been either written or read, and two vector clock objects (version and write). Finally, the client also stores a list with its *onStandByOperations*, and these elements contain the actual operation, a boolean that indicates if it is supposed to be justified by the client itself, and if not, it is ready to store an *EstablishedWrite* object.

```java
public class Client {

    private final int index;
    private final List<EstablishedWrite> establishedWrites;
    private final VectorClock state;
    private final Map<String, List<ValueProperties>> idProperties;
    private List<OnStandByOperation> onStandByOperations;
```

Listing 4.6: Initial part of the Java class that represents a client

Before executing the algorithm itself, it is necessary to setup these clients' state and *idProperties*. The state will consists of an array whose size is equal to the number of clients in the history of operations. Regarding the *idProperties*, for each *id* present in the history of operations, the corresponding *ValueProperties* list will contain a single *ValueProperties* object whose value is *NULL*, and both vector clocks correspond to the client's initial state.

Whenever the current operation is a write, it is stored not only in the *pending* structure of the global state, but also as an *OnStandByOperation* object in the client's *onStandByOperations* list.

Whenever the current operation is an explicit read, the behaviour of the implementation follows the diagram in Figure 4.5. If its client's *onStandByOperations* list is empty, then this operation can be treated immediately. If it is supposed to be justified by the client itself, this condition gets checked. In the case the client cannot justify it, this information is propagated, and it is necessary to check for another justification of this operation.

Otherwise, this explicit read gets treated, and the *idProperties* are updated. In the case that the justification was an established write, then the merge process happens and both the client's *state* and *idProperties* are updated. In the case a pending write is the justification, this explicit read is converted to an *OnStandByOperation* that gets inserted to the client's *onStandByOperations*, whose *EstablishedWrite* property remains *NULL* until the *pending* write gets established. In the case the client's *onStandByOperations* list is not empty, then this operation will be converted to an *OnStandByOperation*, storing the justification in it and can only get treated once all the previous ones get treated. However, if the justification is a pending write, not only it needs to wait for every previous *onStandBy* operation to get treated, but also wait for the pending write that it depends on to transform into an established write, before getting treated.

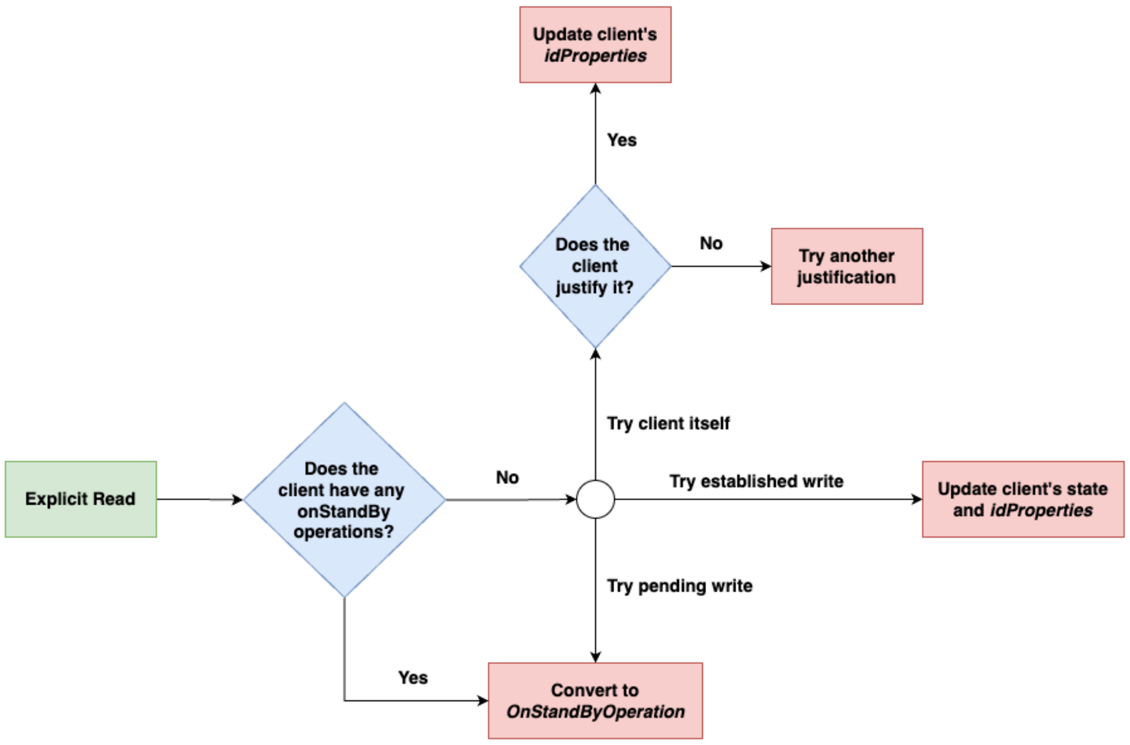
Update client's idProperties
Yes
Does the client justify it?
No
Try another justification
Try client itself
Explicit Read
Does the client have any onStandBy operations?
No
Try established write
Update client's state and idProperties
Try pending write
Yes
Convert to OnStandByOperation

Figure 4.5: Process of treating an explicit read operation on the client side

Whenever the current operation is an implicit read operation, the behaviour of the implementation follows the diagram in Figure 4.6. The corresponding write exits the *pending* state, by removing the entry in the *pending* structure of the global state. As mentioned before, this write is stored in the *onStandByOperations* list of the client. If the client has no *onStandByOperations* (prior to the one corresponding to this write), it follows an initial behaviour similar to the one when dealing with an explicit read, where the difference is that, in the case of using a *pending* write as a justification, it is not created a new *OnStandByOperation*, because the write is already in the *onStandByOperations* list. After updating the properties of the client when either the client itself or an established write justifies the implicit property, the write is established, which will trigger the depth first search process. This process will provide the new *EstablishedWrite* object to the *onStandBy* operations that depend on that write. In the case these dependent operations are the first ones of their corresponding *onStandByOperations* list, every *OnStandByOperation* between this one (inclusive) and the first one that is still waiting for a *pending* write to be established (exclusive) get treated and exit this list. In short, the depth first search process triggers the *onStandByOperations* cleanup, which may also lead to write establishments, triggering the depth first search process again (dependencies), and so on. If the client has *onStandBy* operations prior to the one representing the write, the justification simply gets stored on that write *OnStandByOperation* and only gets treated once all previous *onStandBy* operations get treated. However, if the justification is a pending write, it can only get treated once all the previous *onStandBy* operations get treated and also when the pending write that it depends on transforms into an established write.

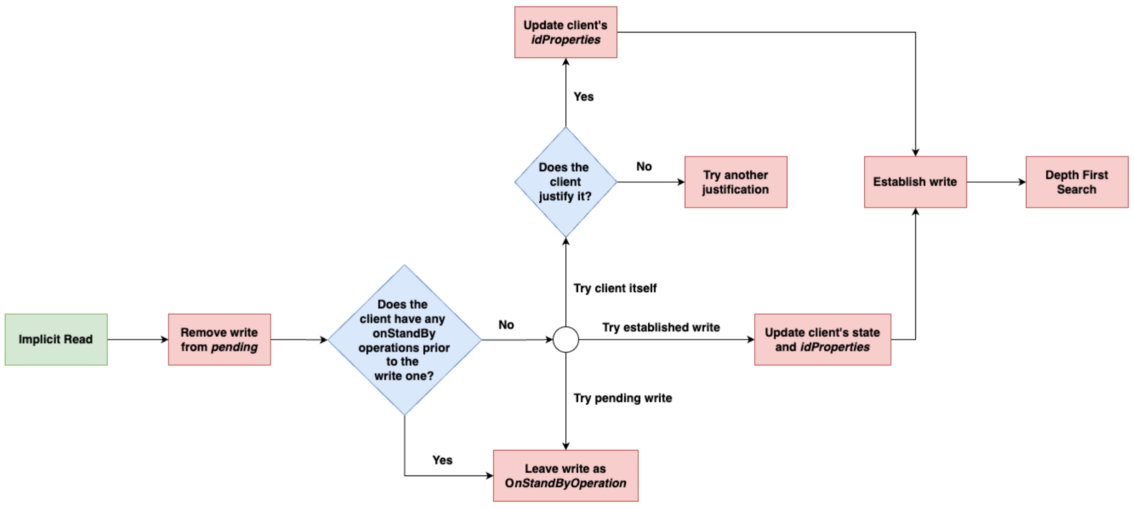
Update client's idProperties
Yes
Does the client justify it?
No
Try another justification
Establish write
Depth First Search
Try client itself
Implicit Read
Remove write from pending
Does the client have any onStandBy operations prior to the write one?
No
Try established write
Update client's state and idProperties
Try pending write
Yes
Leave write as OnStandByOperation

Figure 4.6: Process of treating an implicit read operation on the client side

5 Experiments and Results

This chapter presents an evaluation of the proposed algorithm. The evaluation starts by verifying that the algorithm is able to find causality violations in a set of custom histories designed to illustrate different scenarios of violation. Then, we have experimented with logs from a running server and conclude our experiment with an evaluation of the scalability of the implemented prototype.

5.1 Custom Histories of Operations

This section aims to evaluate the behaviour of the algorithm, not only at the final result (causally consistent or not), but also at specific intermediate results, running our prototype over simple histories of operations created by us. Some of these histories, and corresponding timeline of processed operations, have already been studied conceptually in Chapter 3. For the sake of space, the additional explicit read studied in Section 3.2.2 will not be considered in the timelines we will analyse next.

5.1.1 Read Your Writes

Figure 5.1 illustrates the algorithm execution over a timeline of processed operations. The history comprised by this timeline violates the Read Your Writes guarantee. Client 0 performed a write and Client 1 happened to read it. Finally, Client 0 issues a read on the entity that it created, however, it resulted in a 404. Client 0's explicit read is the only operation that has no justifications. The output is shown in Listing 5.1. It indicates that the corresponding history of operations is not causally consistent, specifying the furthest operation that could not be justified. It is detecting this violation correctly, since after the Client 0's write gets established, Client 0 needs to observe the written value unless there was some operation in some other client that would delete x, which is not the case. As far as the timeline is concerned, the *index* of this operation should be 3, however, the prototype creates an additional explicit read after each successful write operation, which is why its *index* is 4. This situation must be taken into account for the next examples.

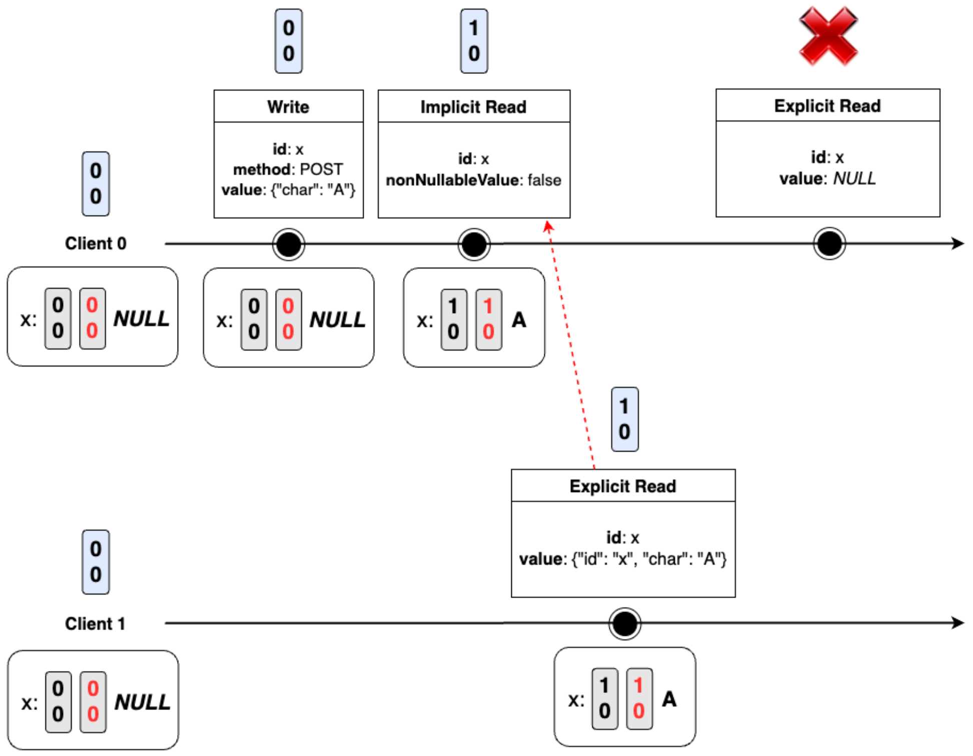

Figure 5.1: Timeline of operations that violates Read Your Writes

```
NOT CAUSAL - Furthest operation that could not be justified was:

index: 4
type: explicit_read
client: 0
id: x
value: null
```

Listing 5.1: Output when feeding the history corresponding to the timeline in Figure 5.1

We also studied the need to use the write vector clock in Section 3.3.5. Figure 3.14 shows the algorithm execution, without write vector clocks, over a timeline that could lead to a Read Your Writes violation if Client 0 had an additional explicit read operation that made it observe *x* with the *char* set to *A*. Figure 3.15 shows the algorithm execution using write vector clocks over the same timeline, which will not consider the *ValueProperties* element of *x* with the *char* set to *A* at Client 0's last explicit read, allowing the algorithm to detect a Read Your Writes violation in the case Client 0 had the additional explicit read described above.

The prototype outputs the established writes and final knowledge of each client when classifying a history of operations as causally consistent. Given that, Client 0's final knowledge output must match the knowledge indicated in Figure 3.15 at Client 0's explicit read. Listing 5.2 shows this portion of the output, which validates the previous requirement.

```
Client 0
    state: [2, 2]
    idProperties: {
        x=[Vector clock: [2, 0] | Write Vector Clock [2, 0] | Value: {char=C}],
        y=[Vector clock: [2, 2] | Write Vector Clock [1, 2] | Value: {char=Z}]
    }
```

Listing 5.2: Portion of the output that corresponds to the final knowledge of Client 0 when feeding the history corresponding to the timeline in Figure 3.15

5.1.2 Monotonic Writes

We have already studied the history of operations of Figure 3.4, which contains a Monotonic Writes violation. The way the algorithm detects the violation is shown in Figure 3.10. When feeding this history to the prototype, we get the output in the Listing 5.3. The prototype is also detecting this violation correctly, since the furthest processed operation it could not justify corresponds to the one we pointed out as not having any justifications.

```
NOT CAUSAL - Furthest operation that could not be justified was:

index: 7
type: explicit_read
client: 1
id: x
value: {char=A}
```

Listing 5.3: Output when feeding the history corresponding to the timeline in Figure 3.10

5.1.3 Monotonic Reads

Figure 5.2 shows the algorithm execution over a timeline of operations that represents a history with a Monotonic Reads violation. Client 0 performs a write and Client 1 reads it. After that, Client 1 performs another read that reads a previous value, which has no justification. As expected, our prototype outputs this operation as the reason the execution corresponding to this history is not causally consistent, i.e., the furthest operation the prototype was able to reach, and Listing 5.4 shows exactly that.

```
NOT CAUSAL - Furthest operation that could not be justified was:

index: 4
type: explicit_read
client: 1
id: x
value: null
```

Listing 5.4: Output when feeding the history corresponding to the timeline in Figure 5.2

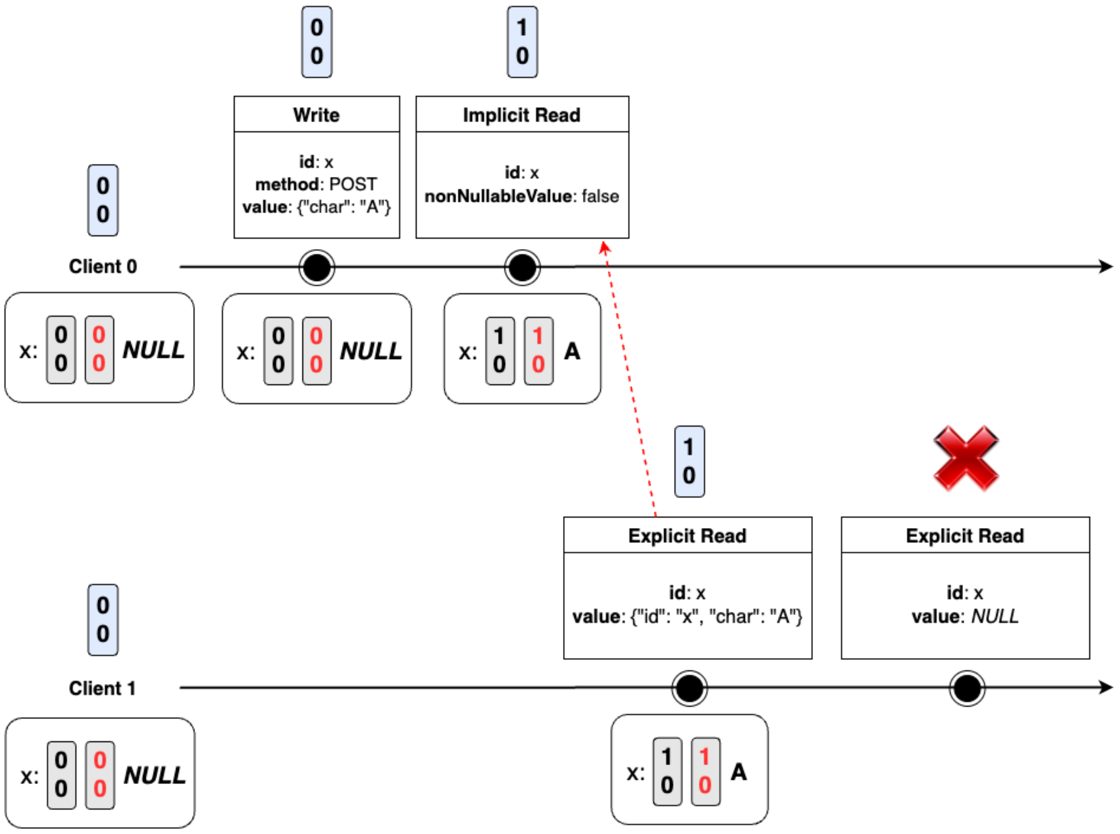

Figure 5.2: Timeline of operations that violates Monotonic Reads

5.1.4 Writes Follow Reads

Figure 5.3 shows the algorithm execution over a timeline of processed operations. The correspondent history contains a Writes Follow Reads violation. Client 0 creates a new entity, *x*, assigning the *char A* to it. Then, Client 1 reads this value and updates its *char* to be *B*. Finally, Client 2 performs two consecutive reads of *x*, observing the *chars B* and *A*, respectively. Client 2's last explicit read violates the Writes Follow Reads, as every replica must apply the write from Client 1 after the write from Client 0, as Client 1 has read the write from Client 0. Given this, Client 2, after reading the write from Client 1 can no longer read the write from client 0. Listing 5.5 shows the output over this specific history, and the furthest operation the prototype was able to justify corresponds to Client 2's last explicit read.

```
NOT CAUSAL - Furthest operation that could not be justified was:

index: 8
type: explicit_read
client: 2
id: x
value: {char=A}
```

Listing 5.5: Output when feeding the history corresponding to the timeline in Figure 5.3

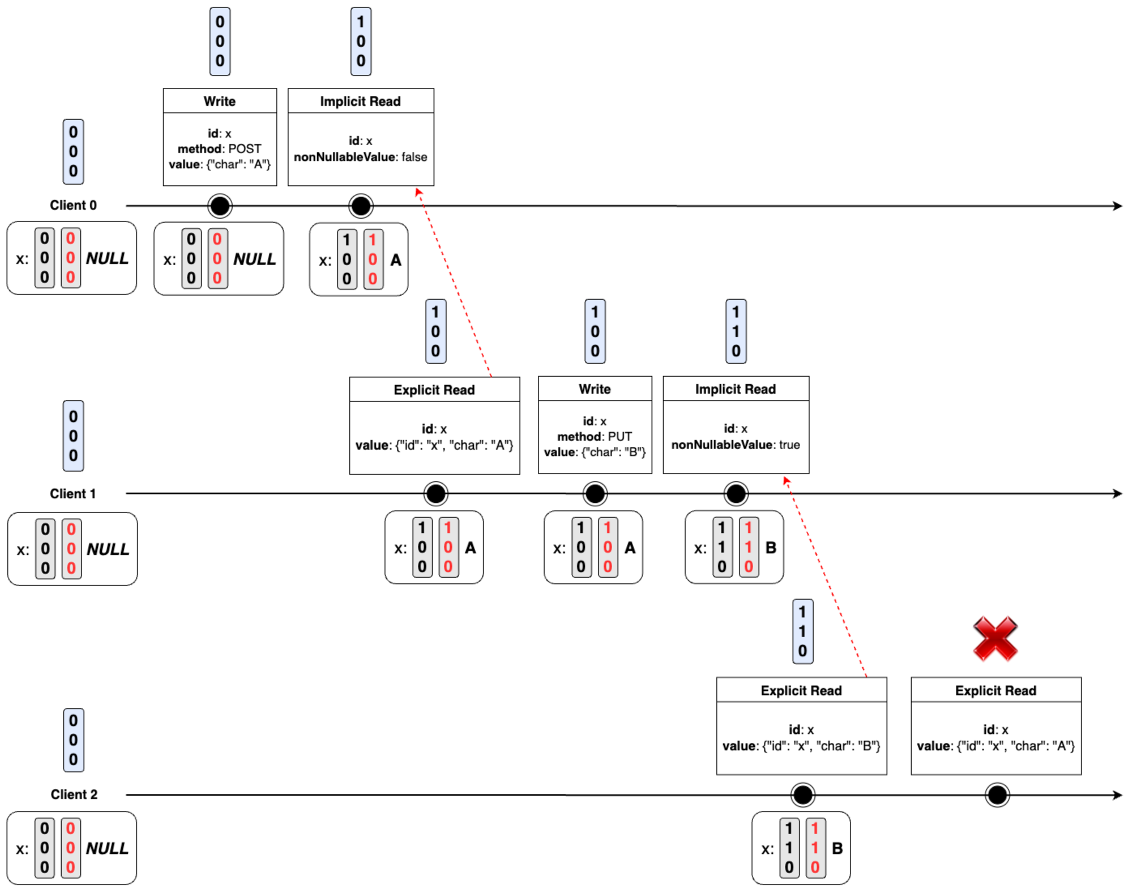

Figure 5.3: Timeline of operations that violates Writes Follow Reads

5.1.5 Multiple Branches Exploration

So far, we have only been submitting to the prototype histories of operations that consist of a single graph, i.e., each processed read operation has at most one justification. In that case, if the prototype has no justifications for a given operation, it will backtrack and consider all possible alternative justifications (dependencies) of each read - if no alternative can justify all operations, the history is classified as violating causal consistency.

We have studied the timeline of operations in Figure 3.11, where it represents a possible set of dependencies that the algorithm created. Client 1's explicit read has two possible justifications, which are the last established writes of Clients 0 and 2. If the algorithm chooses the last established write of Client 2, it will not be able to justify the last explicit read of Client 1, and therefore it needs to backtrack to the previous explicit read of Client 1. If the algorithm chooses the established write of Client 0 (either being the first choice or after backtracking), the last explicit read of Client 1 can now be justified by Client 2's last established write and therefore the history is classified as causally consistent.

Regardless of the prototype behaviour at this situation, the graph that classifies the execution corresponding to this history as causally consistent is the one represented in Figure 3.12. Thus, the output portion that corresponds to the final knowledge of Client

1 must match the knowledge after executing its last explicit read. Listing 5.6 shows that output portion, which indeed validates the requirement above.

```
Client 1
    state: [1, 0, 1]
    idProperties: {
        x=[Vector clock: [1, 0, 1] | Write Vector Clock [0, 0, 1] | Value: {char=A}]
    }
```

Listing 5.6: Portion of the output that corresponds to the final knowledge of Client 1 when feeding the history of operations that corresponds to the timeline of in Figure 3.12

5.1.6 Pending Writes

Figure 3.18 illustrates an execution of the algorithm over a history of operations containing multiple operations that can be justified by pending writes. The operation of Client 0 can only be executed when Client 1 establishes its write. However, Client 1 can only do it once Client 0 establishes its write. The knowledge at the last operation of Client 1 and Client 2 could only be obtained - because the algorithm run depth first search, starting at the moment Client 0 established its write.

Listing 5.7 shows the portion of the output that corresponds to the final knowledge of all clients, which matches the knowledge of clients at their last operation.

```
Client 0
    state: [1, 0, 0]
    idProperties: {
        x=[Vector clock: [1, 0, 0] | Write Vector Clock [1, 0, 0] | Value: {char=A}]
    }

Client 1
    state: [1, 1, 0]
    idProperties: {
        x=[Vector clock: [1, 1, 0] | Write Vector Clock [1, 1, 0] | Value: {char=B}]
    }

Client 2
        state: [1, 1, 0]
        idProperties: {
            x=[Vector clock: [1, 1, 0] | Write Vector Clock [1, 1, 0] | Value: {char=B}]
    }
```

Listing 5.7: Portion of the output that corresponds to the final knowledge of all clients when feeding the history of operations that corresponds to the timeline of in Figure 3.18

5.2 Real RESTful Application

So far, we have been assessing the behaviour of the prototype using histories of operations written by ourselves. However, it is crucial to do it using histories of real RESTful applications that claim either to satisfy causal consistency or not. To this end, a RESTful application was developed, which uses AntidoteDB [5] database.

AntidoteDB claims to ensure causal consistency using vector clocks [4]. Each operation returns a vector clock indicating the time after it has been performed. When executing an operation in Antidote, clients can provide that previous received vector clock to force a minimum time for the snapshot used in the request. This means that if clients always provide these vector clocks, causal consistency is maintained along the application, otherwise it gets jeopardized.

The RESTful application was developed in Java using the Jersey REST framework [25]. In order to access an AntidoteDB cluster, the Java Client 0.3.5 [4] of Antidote was used. This application manages students and maintains the following information about each one: *id*, first name, last name, email, age and phone. The *id* is the only field generated by the application. The REST operations supported by this application are shown in Table 5.1.

Table 5.1: REST operations of the real RESTful application

Operation	**HTTP Method**	**URI**	**Request Body**	**HTTP Status**	**Response Body (on success)**
Create	POST	/	JSON with info	201	JSON with *id* and info
Read	GET	/{*id*}	-	200/404	JSON with *id* and info
Update	PUT	/{*id*}	JSON with info	200/404	JSON with *id* and info
Delete	DELETE	/{*id*}	-	200/404	-

Figure 5.4 illustrates the deployment configuration of this application. It offers two different replicas, where each one is connected to a different Antidote cluster. These clusters are synchronized with each other so that they can replicate data.

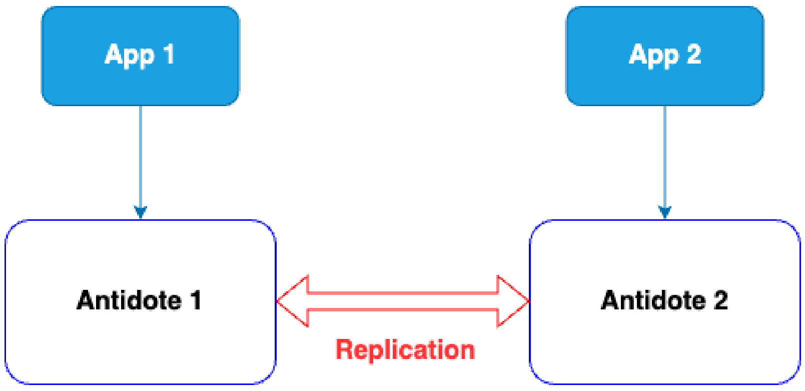

Figure 5.4: Deployment configuration of a real RESTful application using AntidoteDB

With this application built and deployed with the given configuration, we want to assess if it really provides causal executions when clients send the vector clock they received in the previous operation. Likewise, when they do not provide the vector clocks, we expect that some executions do not satisfy causal consistency.

We relied on JepREST to perform workloads of REST operations on this application. We had to adapt JepREST clients based on our requirements. These clients must access replicas randomly, so that they do not always execute requests to the same one, which happens in real applications. The resulting histories of operations will be then analysed by our prototype.

JepREST was used to run several workloads with a different number of REST operations, where the tool's clients neither store or send the data Antidote retrieves when submitting an operation (Antidote's vector clock data). Our prototype classified most of the executions corresponding to these histories as not causally consistent. This result confirms the fact that causal consistency is indeed jeopardized when clients do not provide the Antidote's vector clock data retrieved in their previous operation.

Figure 5.5 illustrates a portion of the timeline of a history of operations that resulted from this assessment, and it shows the initial operations of it. Client 0 creates the entity *9d95* and Client 1 issues a read of the *uafz* that resulted in a 404. Then, Client 0 performs two reads of the entity it created, observing the value it assigned to it and that it does not exist, respectively. Meanwhile, Client 1 attempts to update the value of *9d95*. For the sake of space, the value written by Client 0 at *9d95* is represented by the green ball.

The algorithm cannot find any justification for the last explicit read of Client 0 and since every other read operations have a single justification, it will backtrack all the way to the start and classify the execution corresponding to this history as not causally consistent. Listing 5.8 shows that the reason for it is the last explicit read operation of Client 0.

```
NOT CAUSAL - Furthest operation that could not be justified was:

index: 5
type: explicit_read
client: 0
id: 9d95
value: null
```

Listing 5.8: Output when feeding the history corresponding to the timeline in Figure 5.5

JepREST also performed workloads with a different number of REST operations, where the tool's clients store the data Antidote retrieves when submitting an operation (Antidote's vector clock data) and send it whenever performing a new operation. All executions corresponding to the histories generated by this experiment were classified as causally consistent by our prototype. This does not prove that this RESTful application (the layer above Antidote) is correct as far as causal consistency is concerned. No violations were detected in these specific histories, however, this does not mean that violations may occur on different histories, where clients also take advantage of these vector clocks. We can only conclude that what Antidote claims was verified on our specific tests.

5.2.1 Inserting Violations Inside Generated Histories

We have already validated that the prototype correctly classifies the most simple histories that contain violations. Let us suppose JepREST generated a much larger history of operations of this particular application, which the prototype classified it as causally consistent. If we happen to insert one of those simple not causally consistent histories in

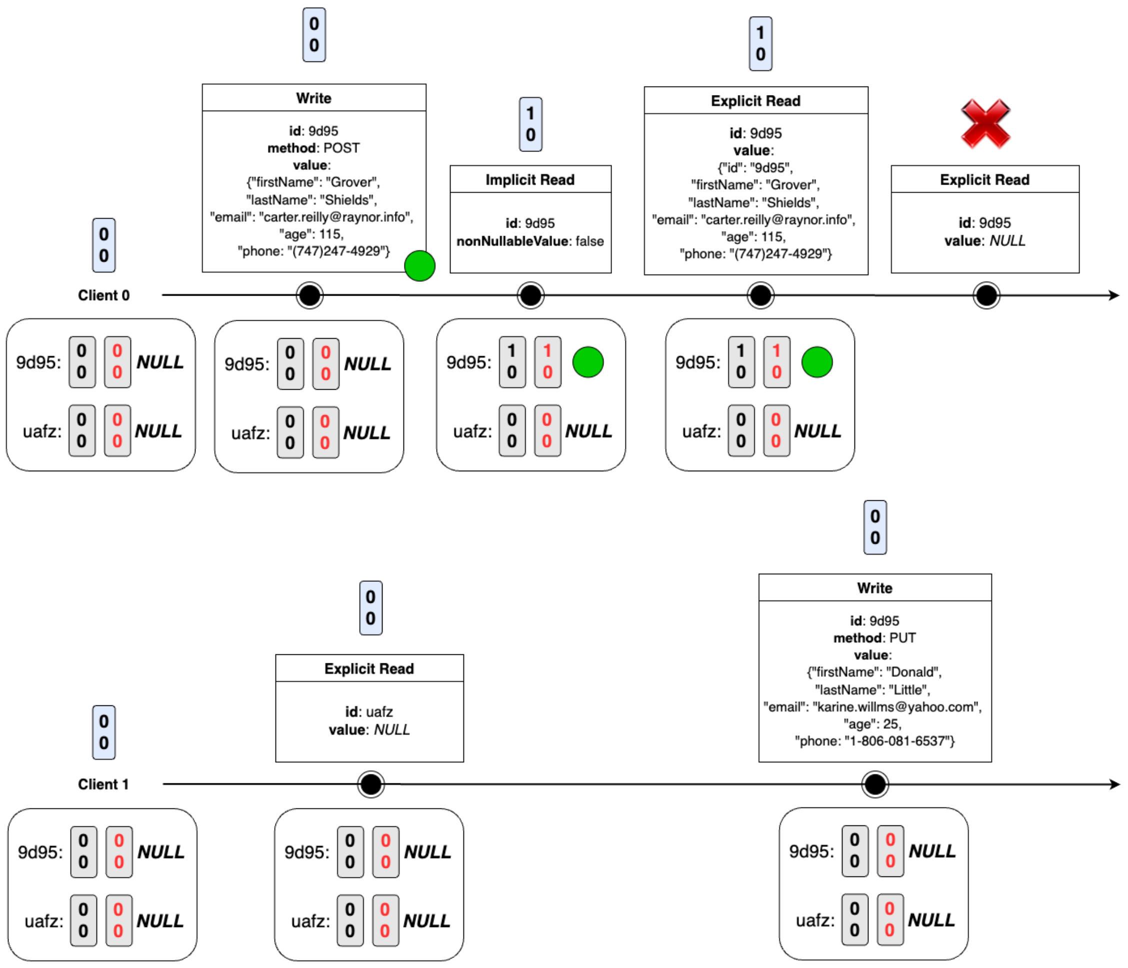
Write
id: 9d95
method: POST
value:
{"firstName": "Grover", "lastName": "Shields", "email": "carter.reilly@raynor.info", "age": 115, "phone: "(747)247-4929"}
Implicit Read
id: 9d95
nonNullableValue: false
Explicit Read
id: 9d95
value:
{"id": "9d95", "firstName": "Grover", "lastName": "Shields", "email": "carter.reilly@raynor.info", "age": 115, "phone: "(747)247-4929"}
Explicit Read
id: 9d95
value: NULL
Client 0
9d95:
uafz:
NULL
Explicit Read
id: uafz
value: NULL
Write
id: 9d95
method: PUT
value:
{"firstName": "Donald", "lastName": "Little", "email": "karine.willms@yahoo.com", "age": 25, "phone: "1-806-081-6537"}
Client 1

Figure 5.5: Portion of the timeline of an history that was generated by JepREST, where clients neither stored or sent the Antidote's vector clock data retrieved in their previous operation

the middle of this larger causally consistent history, we expect the prototype to still detect the violation correctly.

A JepREST workload of over 100 REST operations was performed on the students management application. This means that a history of operations with over 100 invocation and over 100 termination elements was generated. The history whose timeline is illustrated in Figure 3.4 violates Monotonic Writes, and the prototype's behaviour over it has been studied in Section 5.1.2. It was inserted somewhere in the middle of the generated history.

The prototype processed the resulting history file, which contained over 260 processed operations. The output from this experiment is shown in Listing 5.9. The furthest operation reached corresponds to the one that is the reason why the execution corresponding to the smaller history is not causally consistent, which is depicted in the timeline of Figure 3.10. Its *index* (133) shows that this processed operation lies somewhere in the middle of the list of processed operations.

```
NOT CAUSAL - Furthest operation that could not be justified was:

index: 133
type: explicit_read
client: 1
id: x
value: {char=A}
```

Listing 5.9: Output when feeding a large history that contains a violation somewhere in the middle

5.3 Scalability

The scalability of our prototype correlates the number of REST operations to the time it takes the algorithm to validate an history with that number of operations.

The size of histories that violate causality are not a good metric for this type of measurement, because the moment when causality is violated can vary a lot, implying in major execution time differences among different histories with the same number of operations. Considering two histories that happen to violate causal consistency, in one of them the violation might be detected at the very beginning and the violation of the other one might be found only at the end of it. Therefore, the histories of operations for this assessment must not contain any causality violations.

The scalability was measured feeding causally consistent histories, generated in Section 5.2 (student management REST application), to the prototype, taking into account their number of REST operations and the execution time. These histories comprise executions of all four operations provided by the application. In fact, two different histories with the same number of operations that respect causality may have a different number of justifications at each operation, and depending on how and when the algorithm backtracks, their execution times might be slightly different. For this reason, for the same number of operations, different histories were submitted to the algorithm and the final measurement is an average of the execution times of the different histories. These executions were performed on a machine with 32 GB of RAM and two *Intel Xeon E5-2609 v4* CPU. Figure 5.6 illustrates the curve that represents the prototype's scalability. It tends to be exponential and we believe the size of the global state and its constant copying are the reasons for this.

The size of the global state depends on the number of different *id*s present in the history and also on the number of established writes. The larger the number of REST operations, the larger these two properties tend to be. The prototype will be copying global states that might be already large from the beginning (multiple *id*s), and will likely get even larger due to successive write establishments. The global state copying increases a lot because of two factors: the number of REST operations and the number of justifications each operation has. The fact is that the larger the number of REST operations, the higher the probability of having a larger number of justifications at each operation, specially at implicit reads. The algorithm may backtrack several times and since there are more branches to explore, more graphs are being created, meaning that the global state is being copied a lot of times.

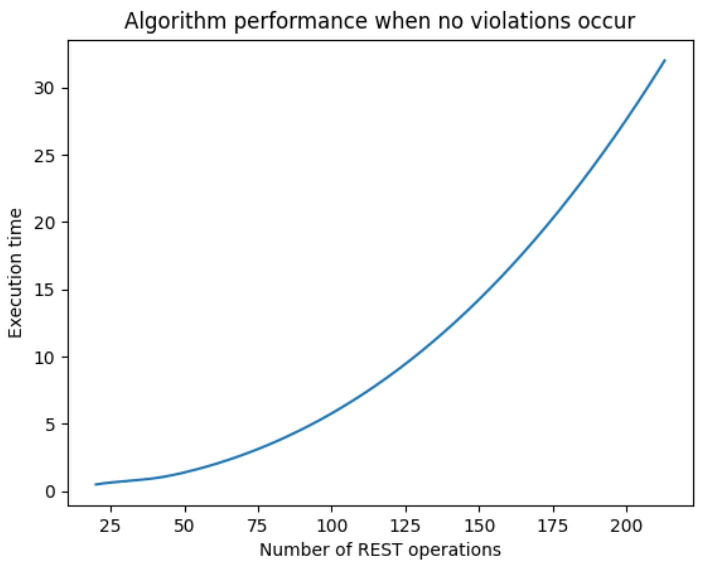

Figure 5.6: Scalability of the algorithm considering only histories of operations without violations

6 Conclusion

This chapter presents the final considerations of this work, as well as the possible improvements that can be applied to it.

6.1 Final Considerations

Many replicated applications rely on weak consistency models to provide their clients low latency and high availability. Many of these applications choose causal consistency as their consistency model, since it is guaranteed that clients observe operations that respect causal relations.

In this dissertation, we have developed an algorithm that verifies whether executions of RESTful applications satisfy causal consistency. These executions are registered in a file, which we call history of operations. This history indicates the order of the requests and responses of the REST operations performed, as well as the data associated to them. JepREST, with the help of Jepsen, performs a workload in the given RESTful application and outputs the corresponding execution history file. This file is then analysed and processed by the proposed algorithm.

The key idea of the algorithm is to try to find a graph of dependencies among operations that justify the results observed. One initial challenge in the processing of REST operations is that besides read and write operations, some operations (e.g., the POST) can be seen an the combination of two other operations (e.g., a write followed by an implicit read).

The algorithm follows a recursive approach, which goes through the operations, following their execution order. At each read operation (either explicit or implicit), it chooses one possible dependency/justification at a time. If the algorithm cannot justify a certain read operation, then it backtracks to the last read operation, choosing another possible justification. If the algorithm backtracks all the way to the beginning, the algorithm classifies the execution corresponding to the history of operations analysed and process as not causal consistent.

The proposed algorithm's is able to verify causal consistency correctly, however, regarding causal+ consistency, there are some histories of operations that the algorithm is not able to verify correctly. Thus, its limitation is that the causal+ consistency verification is not very stable.

Two types of evaluation were performed on the algorithm's prototype. The first one consisted of feeding to the prototype our own histories of operations that included violations of causal consistency, and verify that the prototype was able to detect them correctly. The second one consisted of testing a real distributed RESTful application, which relies on a database that claims to be causally consistent only when clients provide the information retrieved in their last operation. The result of these experiments was the expected - when running the application in a way that respect causality, no violation was found; when not following the rules for respecting causality, our algorithm was able to find causal violations. Regarding the scalability of our prototype, its execution time increases exponentially when increasing the size of histories of operations, which is due to the fact that huge global state structures are copied a lot of times.

6.2 Future Work

We believe there are some interesting functionalities that the algorithm could implement in order to make it as comprehensive and intuitive as possible. These enhancements are the following:

- Handle the corner case patterns that violate causal consistency and causal+ consistency, in which the algorithm does not detect correctly, described in Section 3.3.8.

- Support PATCH operations. This HTTP method is similar to PUT, but the client is able to partially update an entity. The algorithm would need to consider an approach that managed portions of entities and merge them when possible.

- Support operations that retrieve a list of entities (GET all). In the case of the student management RESTful application, detailed in Section 5.2, there could be a GET operation that retrieved all clients. We believe this behaviour could be achieved by considering all combinations of justifications of each operation, instead of just trying each justification in isolation.

- Support write operations that resulted in the 409 HTTP status code (conflict). In the case of the student management RESTful application, the POST (creation) operation could retrieve this status code if the new generated *id* already exists. The algorithm could convert that operation into a single implicit read operation (with no write association), storing the information that this *id* must exist from the given client's point of view.

- Support write operations that resulted in the 500 HTTP status code (internal server error). It seems that the algorithm need to consider the two possible outcomes of this situation: the case that the write was performed on the application and the case that it was not performed.

- Improve the algorithm's scalability/performance, since exponential time is not ideal
- Have the algorithm's prototype to output a visual timeline of the processed operations of the history of operations analysed, including the client's knowledge after performing each processed operation, e.g., Figure 5.5. This way, it would possible to check for intermediate results without having to debug the prototype, making this information more intuitive and easier to access.

Bibliography

[1] P. Alvaro and S. Tymon. “Abstracting the Geniuses away from failure testing”. In: *Communications of the ACM* 61.1 (2018). ISSN: 15577317. DOI: 10.1145/3152483 (cit. on pp. 6, 15).

[2] *An introduction to model checking*. URL: https://www.embedded.com/an-introduction-to-model-checking/. (accessed: 15.12.2021) (cit. on p. 12).

[3] S. Anand et al. “An orchestrated survey of methodologies for automated software test case generation”. In: *Journal of Systems and Software* 86.8 (2013). ISSN: 01641212. DOI: 10.1016/j.jss.2013.02.061 (cit. on pp. 7, 8).

[4] *Antidote Java Client 0.35 API Documentation*. URL: https://www.javadoc.io/doc/eu.antidotedb/antidote-java-client/latest/index.html. (accessed: 27.08.2022) (cit. on p. 78).

[5] *AntidoteDB - A planet scale, highly available, transactional database*. URL: https://www.antidotedb.eu/. (accessed: 27.08.2022) (cit. on p. 77).

[6] *Apache JMeter*. URL: https://jmeter.apache.org. (accessed: 16.02.2022) (cit. on p. 27).

[7] A. Arcuri. “EvoMaster: Evolutionary Multi-context Automated System Test Generation”. In: *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*. 2018. DOI: 10.1109/ICST.2018.00046 (cit. on p. 27).

[8] A. Arcuri. “RESTful API automated test case generation”. In: *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017*. 2017. DOI: 10.1109/QRS.2017.11 (cit. on p. 27).

[9] *Artillery*. URL: https://www.artillery.io. (accessed: 16.02.2022) (cit. on p. 27).

[10] M. A. Austin and J. Johnson. “Compositional approach to distributed system behavior modeling and formal validation of infrastructure operations with finite state automata: Application to viewpoint-driven verification of functionality in

waterways". In: *Systems* 6.1 (2018). ISSN: 20798954. DOI: 10.3390/systems6010002 (cit. on p. 13).

[11] P. Bailis et al. "Bolt-on causal consistency". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2013. DOI: 10.1145/2463676.2465279 (cit. on pp. 21, 22).

[12] A. Basiri et al. "Chaos Engineering". In: *IEEE Software* 33.3 (2016). ISSN: 07407459. DOI: 10.1109/MS.2016.60 (cit. on pp. 15–17).

[13] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, March 1984 (cit. on p. 11).

[14] H. Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *ACM SIGMOD Record* 24.2 (1995). ISSN: N/A. DOI: 10.1145/568271.223785 (cit. on p. 19).

[15] D. Bermbach and S. Tai. "Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior". In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing, MW4SOC 2011 - Co-located with the ACM/IFIP/USENIX 12th International Middleware Conference, Middleware 2011*. 2011. DOI: 10.1145/2093185.2093186 (cit. on pp. 23, 24).

[16] A. Bouajjani et al. "On verifying causal consistency". In: *ACM SIGPLAN Notices* 52.1 (2017). ISSN: 15232867. DOI: 10.1145/3009837.3009888 (cit. on p. 22).

[17] E. A. Brewer. "Towards robust distributed systems (abstract)". In: 2000. DOI: 10.1145/343477.343502 (cit. on pp. 1, 18).

[18] *Chapter 12. Designing for Non-Functional Properties*. URL: https://www.oreilly.com/library/view/software-architecture-foundations/9780470167748/ch12.html. (accessed: 22.12.2022) (cit. on p. 11).

[19] T. Y. Chen, H. Leung, and I. K. Mak. "Adaptive random testing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3321 (2004). ISSN: 16113349. DOI: 10.1007/978-3-540-30502-6{_}23 (cit. on p. 8).

[20] B. Chess and G. Mcgraw. *Static analysis for security*. 2004. DOI: 10.1109/MSP.2004.111 (cit. on pp. 8, 9).

[21] E. M. Clarke, E. A. Emerson, and J. Sifakis. "Model checking: Algorithmic verification and debugging". In: *Communications of the ACM* 52.11 (2009). ISSN: 00010782. DOI: 10.1145/1592761.1592781 (cit. on p. 12).

[22] *Consistency Models*. URL: https://jepsen.io/consistency. (accessed: 01.02.2022) (cit. on p. 29).

[23] J. Cray. "WHY DO COMPUTERS STOP AND WHAT CAN BE DONE ABOUT IT?" In: *Proceedings - Symposium on Reliability in Distributed Software and Database Systems*. 1986 (cit. on p. 2).

[24] G. DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *SOSP'07 - Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. 2007 (cit. on pp. 1, 17).

[25] *Eclipse Jersey*. URL: `https://eclipse-ee4j.github.io/jersey/`. (accessed: 27.08.2022) (cit. on p. 78).

[26] M. Ehmer and F. Khan. "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques". In: *International Journal of Advanced Computer Science and Applications* 3.6 (2012). ISSN: 2158107X. DOI: `10.14569/ijacsa.2012.030603` (cit. on p. 7).

[27] *Elle*. URL: `https://github.com/jepsen-io/elle`. (accessed: 01.02.2022) (cit. on p. 30).

[28] *Faker Documentation*. URL: `https://faker.readthedocs.io/en/master`. (accessed: 07.02.2022) (cit. on p. 63).

[29] F. Freitas et al. "Characterizing the consistency of online services (Practical experience report)". In: *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*. 2016. DOI: `10.1109/DSN.2016.64` (cit. on pp. 1, 17–20, 22, 24).

[30] *Functional Testing Vs Non-Functional Testing: What's the Difference?* URL: `https://www.guru99.com/functional-testing-vs-non-functional-testing.html`. (accessed: 22.12.2022) (cit. on p. 11).

[31] W. Golab et al. "Client-centric benchmarking of eventual consistency for cloud storage systems". In: *Proceedings - International Conference on Distributed Computing Systems*. 2014. DOI: `10.1109/ICDCS.2014.57` (cit. on p. 26).

[32] I. Gomes et al. "An overview on the Static Code Analysis approach in Software Development". In: *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009) (cit. on pp. 8, 9).

[33] *Gson - A Java serialization/deserialization library to convert Java Objects into JSON and back*. URL: `https://github.com/google/gson`. (accessed: 06.09.2022) (cit. on pp. 66, 68).

[34] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990). ISSN: 15584593. DOI: `10.1145/78969.78972` (cit. on pp. 18, 19, 61).

[35] Y. Izrailevsky and A. Tseitlin. "The netflix simian army". In: *The Netflix Tech Blog* (2011) (cit. on pp. 1, 27).

[36] *Jepsen*. URL: `https://jepsen.io/`. (accessed: 01.02.2022) (cit. on pp. 2, 3, 28, 61).

[37] Z. M. Jiang et al. "Automatic identification of load testing problems". In: *IEEE International Conference on Software Maintenance, ICSM*. 2008. DOI: `10.1109/ICSM.2008.4658079` (cit. on p. 11).

[38] M. E. Khan. “Different approaches to white box testing technique for finding errors”. In: *International Journal of Software Engineering and its Applications* 5.3 (2011). ISSN: 17389984. DOI: 10.5121/ijsea.2011.2404 (cit. on p. 7).

[39] K. Kingsbury and P. Alvaro. “Elle: Inferring isolation anomalies from experimental observations”. In: *Proceedings of the VLDB Endowment* 14.3 (2020). ISSN: 21508097. DOI: 10.14778/3430915.3430918 (cit. on p. 29).

[40] *Knossos*. URL: https://github.com/jepsen-io/knossos. (accessed: 01.02.2022) (cit. on pp. 2, 29, 64).

[41] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (1978). ISSN: 15577317. DOI: 10.1145/359545.359563 (cit. on p. 21).

[42] *Linearizability versus Serializability*. URL: http://www.bailis.org/blog/linearizability-versus-serializability. (accessed: 20.01.2022) (cit. on p. 19).

[43] W. Lloyd et al. “Don’t settle for eventual”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP ’11*. New York, New York, USA: ACM Press, 2011, p. 401. ISBN: 9781450309776. DOI: 10.1145/2043556.2043593. URL: http://dl.acm.org/citation.cfm?doid=2043556.2043593 (cit. on pp. 3, 47).

[44] *Logical clocks - Causality and concurrency*. URL: https://people.cs.rutgers.edu/~pxk/417/notes/logical-clocks.html. (accessed: 25.08.2022) (cit. on pp. 3, 42).

[45] J. M. Lourenço. *The NOVAthesis LaTeX Template User’s Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[46] H. Lu et al. “Existential consistency: Measuring and understanding consistency at Facebook”. In: *SOSP 2015 - Proceedings of the 25th ACM Symposium on Operating Systems Principles*. 2015. DOI: 10.1145/2815400.2815426 (cit. on pp. 18, 21, 25).

[47] P. Mahajan, L. Alvisi, and M. Dahlin. “Consistency, availability, and convergence”. In: *University of Texas at Austin TR-11-22 (May)* TR-11-22 (2011) (cit. on pp. 2, 21).

[48] A. Martin-Lopez et al. “Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?” In: *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2021, pp. 231–241 (cit. on p. 27).

[49] F. Mattern. “Virtual Time and Global States of Distributed Systems”. In: *Event London* pages (1989). ISSN: 10980121 (cit. on p. 42).

[50] C. McCaffrey. “The verification of a distributed system”. In: *Communications of the ACM* 59.2 (2016). ISSN: 15577317. DOI: 10.1145/2844108 (cit. on pp. 10, 12–15).

[51] *Monitoring Observability in Distributed Systems*. URL: https://www.linkedin.com/pulse/monitoring-observability-distributed-systems-hossein-samarrokhi. (accessed: 16.12.2021) (cit. on p. 13).

[52] S. Nidhra. "Black Box and White Box Testing Techniques - A Literature Review". In: *International Journal of Embedded Systems and Applications* 2.2 (2012). DOI: 10.5121/ijesa.2012.2204 (cit. on pp. 6, 7, 10).

[53] *Practice Jepsen Test Framework in Nebula Graph*. URL: https://nebula-graph.io/posts/practice-jepsen-test-framework-in-nebula-graph. (accessed: 01.02.2022) (cit. on p. 29).

[54] *Principles of chaos engineering*. URL: https://principlesofchaos.org. (accessed: 12.01.2022) (cit. on pp. 15, 17).

[55] A. Ribeiro. "Invariant-Driven Automated Testing". MA thesis. Universidade Nova de Lisboa, 2021 (cit. on p. 63).

[56] C. Rosenthal and N. Jones. *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media, 2020. Chap. 3 (cit. on p. 15).

[57] *Serializability*. URL: https://jepsen.io/consistency/models/serializable. (accessed: 20.01.2022) (cit. on p. 19).

[58] S. Simões. "JepREST: Sistema para teste funcional de aplicações REST distribuidas". MA thesis. Universidade Nova de Lisboa, 2021 (cit. on pp. 3, 4, 35, 61).

[59] *Strict Serializability*. URL: https://jepsen.io/consistency/models/strict-serializable. (accessed: 20.01.2022) (cit. on p. 19).

[60] Y. Tang et al. "CausalTester: Measuring the Consistency of Replicated Services via Causality Semantics". In: *2021 IEEE 30th Asian Test Symposium (ATS)*. 2021, pp. 49–54. DOI: 10.1109/ATS52891.2021.00021 (cit. on p. 24).

[61] D. B. Terry et al. "Session guarantees for weakly consistent replicated data". In: *Parallel and Distributed Information Systems - Proceedings of the International Conference*. 1994. DOI: 10.1109/pdis.1994.331722 (cit. on pp. 1, 18, 20, 21).

[62] P. Viotti and M. Vukolić. "Consistency in non-transactional distributed storage systems". In: *ACM Computing Surveys* 49.1 (2016). ISSN: 15577341. DOI: 10.1145/2926965 (cit. on p. 18).

[63] K. Vorobyov and P. Krishna. "177 Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches". In: *Proc. SSV* (2010) (cit. on p. 12).

[64] *What is CRUD?* URL: https://www.codecademy.com/article/what-is-crud. (accessed: 29.12.2022) (cit. on p. 2).

[65] R. Zennou et al. "Checking causal consistency of distributed databases". In: *Computing* (2021). ISSN: 14365057. DOI: 10.1007/s00607-021-00911-3 (cit. on p. 22).

2023 Causal Consistency Verification in Restful Systems Hugo Rodrigues

NOVA SCHOOL OF SCIENCE & TECHNOLOGY