electronics

MDPI

Article

VConMC: Enabling Consistency Verification for Distributed Systems Using Implementation-Level Model Checkers and Consistency Oracles

Beom-Heyn Kim

Department of Computer Science and Engineering, Hanyang University, Ansan 15588, Republic of Korea; beomheynkim@hanyang.ac.kr; Tel.: +82-031-400-1036

Abstract: Many cloud services are relying on distributed key-value stores such as ZooKeeper, Cassandra, HBase, etc. However, distributed key-value stores are notoriously difficult to design and implement without any mistakes. Because data consistency is the contract for clients that defines what the correct values to read are for a given history of operations under a specific consistency model, consistency violations can confuse client applications by showing invalid values. As a result, serious consequences such as data loss, data corruption, and unexpected behavior of client applications can occur. Software bugs are one of main reasons why consistency violations may occur. Formal verification techniques may be used to make designs correct and minimize the risks of having bugs in the implementation. However, formal verification is not a panacea due to limitations such as the cost of verification, inability to verify existing implementations, and human errors involved. Implementation-level model checking has been heavily explored by researchers for the past decades to formally verify whether the underlying implementation of distributed systems have bugs or not. Nevertheless, previous proposals are limited because their invariant checking is not versatile enough to check for the wide spectrum of consistency models, from eventual consistency to strong consistency. In this work, consistency oracles are employed for consistency invariant checking that can be used by implementation-level model checkers to formally verify data consistency model implementations of distributed key-value stores. To integrate consistency oracles with implementation-level distributed system model checkers, the partial-order information obtained via API is leveraged to avoid the exhaustive search during consistency invariant checking. Our evaluation results show that, by using the proposed method for consistency invariant checking, our prototype model checker, VConMC, can detect consistency violations caused by several real-world software bugs in a well-known distributed key-value store, ZooKeeper.

Keywords: cloud service; distributed key-value store; model checking; software testing; data consistency

Citation: Kim, B.-H. VConMC: Enabling Consistency Verification for Distributed Systems Using Implementation-Level Model Checkers and Consistency Oracles. *Electronics* **2024**, *13*, 1153. https://doi.org/10.3390/electronics13061153

Academic Editor: Fernando De la Prieta Pintado

Received: 9 February 2024
Revised: 7 March 2024
Accepted: 18 March 2024
Published: 21 March 2024

1. Introduction

Cloud services are becoming more and more popular these days. Among various cloud services, cloud storage services are one of the most predominant ones, as data generated by prolific smart devices have been increasing at an unprecedented pace. Cloud storage services are often built on top of distributed key-value stores, e.g., ZooKeeper, MongoDB, Cassandra, and HBase [1–4]. Distributed key-value stores are essentially distributed systems, which are known to be notoriously difficult to implement without software bugs. Indeed, there are many bugs in real-world open-source distributed systems as reported by developers and researchers [5]. The reason why distributed systems are hard to execute correctly is due to the difficulty of correctly reasoning about concurrent events involved in complex distributed protocols. Additionally, distributed systems must be able to gracefully handle various scenarios of unexpected failures. For instance, leader election protocols

are one of the key components of distributed systems. Once elected, leaders are expected to coordinate the order of distributed events to guarantee the consistency of replicas. In addition, leader election and coordination mechanisms must work flawlessly, even in the presence of unforeseeable failures. Any mistake made in the relevant code may introduce a fork in the history of data, which can possibly lead to data corruption or data loss.

Among concurrency bugs in distributed key-value stores, software bugs causing the violation of data consistency models, so-called consistency bugs, are critical ones that may confuse client applications to make incorrect decisions [5,6]. When consistency bugs manifest, the violation of data consistency models (hereafter, 'consistency model' and 'data consistency model' are interchangeably used) occurs. Imagine a social network application where Alice and Bob post messages to exchange their opinions back and forth to make some decision. This setting is widely known to be a good application scenario to use causal consistency [7]. If some consistency bugs occurred and updates made by Alice and Bob were unintentionally re-ordered, the causality among messages of Alice and Bob might not be kept. Now, the third person, say Carol, who supposes those messages to be displayed in the order that keeps the causal order, may unexpectedly read their messages in the incorrect order and misunderstand the decision made earlier between Alice and Bob. If a similar problem occurs in safety-critical scenarios like traffic control systems where autonomous vehicles exchange their messages about the current positions, velocity, acceleration, etc., then very serious catastrophic failures can happen. Therefore, it is very important to find such consistency bugs in distributed systems before they manifest.

Nevertheless, finding consistency bugs in distributed key-value stores is still an open problem that previous proposals have not yet fully addressed. Finding bugs in distributed systems has been explored by many researchers using various approaches to more effectively test corner cases of *systems-under-test (SUTs)*. However, previous works have not studied how to detect violations of consistency models well when they actually occur. One line of previous works either leverages random testing approaches like fuzzing methods [8–13]. However, fuzzing cannot systematically explore the state space of SUTs; therefore, it may miss some bugs. Although advancement of formal methods may be used to develop formally verified bug-free distributed systems, it still has several limits, such that it takes a lot of time and effort for non-researchers, cannot be directly applied to existing implementations, and a formally verified design does not always imply a correct implementation [14–16]. On the other hand, implementation-level model checking can systematically enumerate the state space of SUTs [17–26] unlike fuzzing and can directly verify the implementation of distributed systems without the limitations of formal method approaches. However, model checking tends to focus on finding general types of concurrency bugs in distributed key-value stores. Thus, it lacks an effective generic invariant checker that can detect the violation of a diverse consistency model.

We propose integrating flexible and interactive consistency oracles with implementation-level model checkers to enhance model-checking approaches with the new capability of detecting violations for various consistency models. We designed and implemented a prototype implementation-level model checker, VConMC (VConMC stands for Verification of Consistency using Model Checker), to demonstrate the effectiveness of the proposed idea. It is an implementation-level model checker equipped with the consistency-specific invariant checker based on consistency oracles [27]. Consistency oracles can tell us what the valid values to read are for the given history of operations (i.e., reads and writes) with respect to consistency models.

The major challenge is to find out how to prepare the inputs for the consistency oracles. Consistency oracles require the history of operations that honor the actual order of operations. From the client perspective, without knowing information associated with the system's internal mechanisms for ordering like clocks, it is very difficult to find out the ordering information of concurrent operations. It is especially challenging if there were concurrent failures during request handling because it affects the timing when operations are actually applied. Instrumenting the source code may help to gain useful insights in the

order of operations, but it requires heavy engineering effort, and the code instrumentation itself may not be sustainable when the source code gets updated. Previous works offering consistency invariant checking, including Jepsen [8], tend to employ exhaustive searches by exploring all possible interleaving of concurrent events. Although the exhaustive approach for consistency invariant checking may work for a small number of concurrent requests and failures, its application to the realistic deployment scenario often faces with the state explosion problem as the scale of the testing environment increases. With this, the state explosion problem inherently embedded in the implementation-level model checking easily becomes more severe.

The key observation is that there exists the partial-order exposing API, which enables consistency invariant checking without the exhaustive search through every possible interleaving of concurrent events. Although such an API is not mandatory for key-value stores, it is desirable component that can be easily extended by developers because it can enable consistency invariant checking without making state explosion problems more severe. By exposing the partial-order information to clients, SUTs may reveal a hint of the order of operations. Therefore, we can construct the history of operations, even in the presence of failures during the request handling, only by inspecting the request and response via the API layer of SUTs. With the history of operations and the given consistency models, consistency oracles answer the query requests by providing the collection of valid values that can be read. Then, comparing the answer from the consistency oracles and the actual values read from the SUTs allows us to determine whether there has been any violation of consistency guarantees or not.

To validate the proposed methodology, the prototype tool VConMC is applied to ZooKeeper and demonstrates that the proposed methodology allows us to correctly detect consistency violations. Initially, VConMC uses the systematic and exhaustive state space enumeration of its model checking component to manifest bugs causing consistency violations. Then, when bugs are successfully reproduced, VConMC's consistency invariant checking component which is built on top of consistency oracles can detect violations. As a result, the proposed methodology could effectively reproduce 3 real-world software bugs in ZooKeeper and detect violations without being suffered from exploring various interleaving of client requests.

Contributions

The main contribution in this work is the novel implementation-level model checker integrated with consistency invariant checking based on consistency oracles. There has been no bug finding tool that utilizes systematic and exhaustive state space exploration along with a consistency invariant checker specifically designed for detecting violations of various consistency models while mitigating the state explosion problem during invariant checking. The key enabler is making the observation that there is some client API that offers enough hints on the internal ordering of concurrent operations. Previous tools could not avoid extra heavy engineering effort or additional computation costs because those tools could not leverage the hints on the internal partial-order information exposed by a client API. This work introduces the novel methodology of integrating the invariant checker and the model checker by extensively leveraging such an API and demonstrates its effectiveness.

In short, this work makes the following specific contributions:

1. Making the key observation that there is a client API exposing internal partial-order information;
2. Building the consistency invariant checker by leveraging the aforementioned client API and consistency oracles;
3. Integrating an implementation-level model checker with the consistency invariant checker above;
4. Demonstrating the effectiveness of the proposed methodology by applying it to find real-world consistency bugs in a well-known distributed key-value store.

The rest of the paper is structured as follows: Section 2 provides background information about consistency models. Subsequently, in Section 3, the prototype implementation-level model checker VConMC and consistency oracles are discussed. Then, the proposed methodology for consistency invariant checking follows. More specifically, we elaborate the major challenge involved in consistency invariant checking that does not know any ordering information and the novel methodology of constructing the history of operation using the partial-order information exposed via API through the example case study with ZooKeeper. After that, Section 4 shows the evaluation results demonstrating the effectiveness of VConMC followed by relevant discussion. In Section 5, related works are summarized. Finally, the conclusion is given in Section 6.

2. Consistency Models

Various consistency models have different characteristics suitable for specific application scenarios in terms of data consistency, availability, and latency [28,29]. Terry et al. compiled a selection of consistency models that are supported by many distributed key-value store systems in the context of cloud computing [30]. As this work focuses on distributed key-value store systems for cloud storage services, the aforementioned selection of consistency models is used in this work as a reference. However, it is also possible to extend our proposed methodology to other consistency models. For example, there are consistency models clarified by Higham et al. [31].

Applications relying on the data freshness prefer the strong consistency model to be high availability or short latency. On the other hand, e-commerce applications can tolerate some inconsistent data such as missing items in the shopping cart as customers can easily find and fix this problem when they proceed to purchase. Those e-commerce applications prefer servicing more customers with the least service disruption and with short latency. To meet the needs of the former type of application, which prefers the strictness of data consistency over high availability or short latency, distributed key-value stores employ strong consistency, where concurrent operations are handled as if they are processed by a single logical server. Meanwhile, for the latter type of applications, which favors the high availability or short latency over the strictness of data consistency, the weak consistency models such as eventual consistency have gained large attention from both the industry and academia for the last decades. Moreover, there are various intermediate-level consistency models that are between strong consistency and eventual consistency.

Various consistency models define their own ordering rules of concurrent operations and the timing constraints when previous operations must become visible for subsequent operations. Under strong consistency, distributed key-value stores should provide an illusion that all operations are handled by a single server node. In order to accomplish this, distributed key-value stores order every operation sequentially as there is a total order among all operations, which implies that every client should see the coherent unique global history of operations. Also, each operation must become visible as soon as it is applied. That is, once the write operation is applied on a replica of distributed key-value stores, every subsequent read operation applied afterwards should see the effect of the last write operation. Such a strong consistency guarantee is also known as linearizability [32], which is the most natural consistency model to reason about for client application developers. For instance, under such a strong consistency model, it is obvious to see which value a client should have read via its read operation given the global history of operations applied previously: it would be the latest write made to the key requested by the read operation.

The strong consistency model sacrifices availability or latency at the cost of offering strict data consistency. One way to implement the strong consistency model is by using a two-phase commit protocol that coordinates concurrent operations [33]. The two-phase commit protocol requires additional network and processing delays to ensure all nodes agree on the same update to records. Another way to implement the strong consistency model is by having each client request access to the majority of nodes for writes and reads. Because at least one node is accessed for every write and read, there is at least one node that

has the latest version of records. Therefore, the networking and processing delays involved are much greater compared with those of the single server's case. In addition, if a network partition occurs for some replicas, the cluster may become unavailable for clients, which can only connect to those partitioned server nodes. Thus, clients on the side of the partitioned server nodes must either wait until the network partition is recovered so that the majority of servers become reachable again or retry with different server nodes in the hope of being the one on the majority side. Thus, it is unavoidable that the strong consistency model is the worst choice for applications preferring high availability or short latency.

On the other hand, the eventual consistency model is the weakest form of consistency models, which only guarantees the liveness property [34]. The definition of the eventual consistency model is that, as long as there is no newer write performed, all replicas are eventually going to converge. Thus, the eventual consistency does not restrict operations to keep the rules for the ordering and timing of operations. For example, suppose one client sequentially issued write operations X and Y. Clients may read Y first and read X next, while other clients read X first and then read Y. Unlike the strong consistency model implementation, the eventual consistency model allows each operation to be applied on each server independently without going through the coordination of server nodes using consensus protocols. Therefore, there is no longer additional networking and processing delays involved in processing concurrent operations. Failures do not interrupt operation handling for clients on the partitioned side when the network partition occurs. Thus, under the eventual consistency model, clients can take advantage of high availability or short latency at the cost of giving up strict data consistency.

Moreover, there are various consistency guarantees that are considered to be in the middle of the strong consistency and eventual consistency with respect to the strictness of data consistency. For example, there are several session guarantees such as monotonic reads, consistent prefix, bounded staleness, read-my-write, etc. Each session guarantee has a specific ordering and timing constraint to be met within a session between a client and a distributed system. Monotonic reads guarantee that each read operation within the same session will see a write that is at least as new as the one seen by the last read. Consistent prefix ensures that the prefix of the write sequences within the same session is maintained to be coherent for each read operation. Bounded staleness guarantees that read operations will be able to see the write operations that are no older than the given time bound allows. Read-my-write assures that each read operation sees write operations as recent as the one that the same client issued for the last time within the same session. These intermediate-level consistency guarantees may be used separately or combined to compose a unique consistency model to meet each application scenario.

3. Materials and Methods

3.1. Problem Statement

Finding consistency bugs require the proposed solution to solve a few challenging issues. First, bugs need to be reliably manifested, which means state space exploration of the system-under-test should be reliable enough to drive the system to reach a target state without observing any significantly non-deterministic behaviors. Second, missing corner cases should be avoided by systematically and exhaustively enumerating possible states of the given system-under-test. Third, when a bug is reproduced, it is essential to detect its consequences using dedicated invariant checkers. More specifically, consistency bugs cause the violation of consistency models, which may be varied from systems to systems. Consistency invariant checkers must adapt flexibly and should be easily extensible. Fourth, consistency invariant checking itself should not contribute to making the state explosion problem of model checking worse. Model checking today already suffers from the huge number of states to explore. Employing consistency invariant checking methodology that adds up the complication cannot be considered as a desirable solution that can be used with model checkers.

Therefore, the proper solution to find consistency bugs must fulfill the following requirements:

1. It must be able to reliably explore state space of the system-under-test;
2. It must be able to systematically and exhaustively explore the state space;
3. It must be able to check for various consistency models;
4. It must be able to mitigate state explosion problem during invariant checking.

The proposed solution can solve the aforementioned problems by satisfying the requirements above by combining each subsystem that can solve each issue in a novel way. First, adding a hook carefully into the system-under-test allows it to tightly control non-deterministic behaviors as much as possible. Second, utilizing the stateless state space search technique allows it to support systematic and exhaustive state enumeration. Third, leveraging consistency oracles allows it to flexibly check for various consistency models. Fourth, using internally used ordering information exposed via a client API helps in inferring the order of operations to be used along with the consistency oracles to perform consistency invariant checking without needing to explore all interleaving of concurrent operations.

More specifically, the first two requirements are fulfilled by the controlled environment component described in Section 3.3 and the model-checking server's state space exploration component presented in Section 3.4, respectively. The third requirement is satisfied by using the consistency oracles component elaborated in Section 3.5 and, finally, meeting the fourth requirement is carried out by the model-checking server's invariant checker component discussed in Section 3.6.

3.2. Architecture Overview

Figure 1 shows the architecture of VConMC, our prototype implementation-level model checker that is specially designed to verify the consistency model implementation of the given distributed key-value store. The given distributed key-value store should run in a controlled environment in a way that concurrent events are executed in a deterministic fashion as orchestrated by the model checker. While the model checker is exploring the state space of the given distributed systems, it systematically explores each path starting from an initial state and ending at one of the leaf states. At the end of every path exploration, the execution results of the clients' read and write requests are verified by checking whether any violation has occurred with respect to the given consistency models using consistency oracles. The check for consistency violation involves the membership test of each read operation to see if the value seen by the read operation is in the set of valid values returned by consistency oracles.

Consistency model checkers should essentially consist of 3 key components: (1) an SUT, which is a distributed key-value store system wrapped by an instrumentation layer to be integrated into the controlled environment for deterministic state space exploration; (2) a model-checking server actually orchestrating the state space exploration of the given SUT and conducting the invariant checks; and (3) a consistency oracle accepting the global history of operations and producing the set of all valid values to read as an answer to a query from the invariant checker module within the model-checking server. In this section, we describe each component of VConMC in more detail.

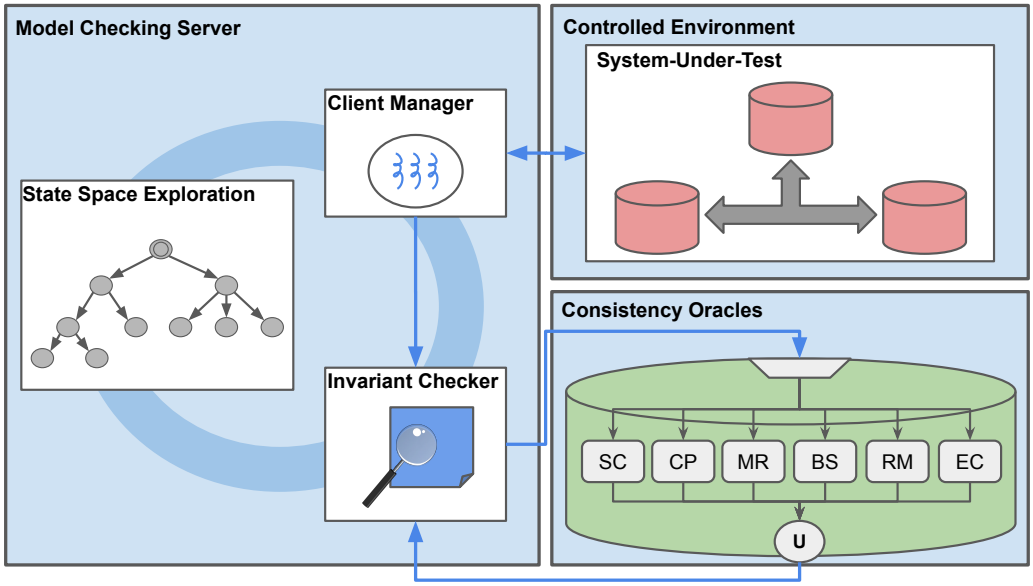

Figure 1. VConMC architecture. VConMC is an implementation-level model checker using consistency oracles as the basis of consistency invariant checking. The model-checking server is composed of three major components such as the state space exploration module, client manager module, and invariant checker module. System-under-test is instrumented to work with the model-checking server within a controlled environment for the verification purpose. Consistency oracles contain several reference consistency modules to produce the set of valid values to read for the given history of operations. The client manager module provides requests and responses sent to and received from the system-under-test for the invariant checker module to interact with consistency oracles during consistency invariant checking.

3.3. Controlled Environment

As a prerequisite, we need to wrap the SUT with an instrumentation layer to tightly control concurrent events and environmental events so that the SUT closely follows the direction of the model checker. Distributed systems heavily utilize concurrency supports of the underlying systems for more efficient performance and higher availability. Such concurrency yields a significant number of concurrent events that can occur with no specific order. In addition, various environmental events such as the stochastic load on the host systems as well as the unexpectedly occurring system crashes or network partition failures can also contribute to the non-deterministic behavior of SUT. Only after suppressing the non-deterministic behavior of the SUT can model checkers tightly control the order of events to systematically drive the SUT into each state.

Implementation-level model checkers usually solve this issue in two ways. First, the code of SUT is instrumented in a way that any non-deterministic event about to occur is interposed by the model checker, and the actual execution of it is blocked until the model checker allows it to pass. Instrumentation usually inserts a hook before the line of codes where the non-deterministic event is executed by adding a hook before a non-deterministic event executes; a thread attempting to execute that event traps into the model-checking server and waits for the right timing to fire the interposed event. In this work, instrumentation was performed manually. Adding the hook via automated instrumentation is left as future work. When applying VConMC to ZooKeeper, it took about a couple of weeks to add the necessary hooks through the manual instrumentation process by one person who has moderate knowledge about the system-under-test. Second, the well-defined interface can serve as the interposition layer. For instance, the library wrapping the system calls can be the layer for model checkers to interpose non-deterministic events. Each invocation of system calls is blocked and waits for the model checker to allow it to execute. Hence, the model checker can control the order of system call execution without

modifying the target SUT. However, because some system calls have semantics that can change depending on the input parameters, the implementation of the interposition layer is brittle and more complicated, which requires quite a bit of engineering effort. Thus, VConMC employs the first approach to control the non-determinism of an SUT.

Blocking and giving control over non-deterministic events to the model checker is the key mechanism to systematically steer the SUT to reach each state in a controlled manner. For instance, the line of code that acquires a lock and the line of code that releases the acquired lock guard a critical section to prevent the race condition. However, it is possible that the code may have a concurrency bug that can lead to a deadlock depending on the order of executions. By blocking each concurrent event and explicitly controlling the execution order, the model checker can exhaustively and systematically explore all possible interleaving of acquiring and releasing locks around the critical section.

Figure 2 shows an example that illustrates the scenario where 4 threads concurrently run in parallel and, therefore, a few non-deterministic events occur. Each thread may be running on distributed nodes and communicating through the network. There may be a lock shared by threads running on the same node. For instance, `Thread A` and `Thread B` compete for the lock by trying to successfully execute *lock_acquire* first. In addition, more than two nodes may attempt to send messages to the common destination node at the same time. As an example, both `Thread B` and `Thread C` are concurrently executing *send_msg*. The order of each event is non-deterministic during the execution of the deployed system. `Thread A` may successfully invoke lock_acquire and obtain the exclusive ownership of the shared lock before `Thread B` does. However, it is also possible that `Thread B` successfully acquires the shared lock before `Thread A` does. Moreover, `Thread B` may send a message to `Thread D` before `Thread C` does, or the execution may occur in the reverse order.

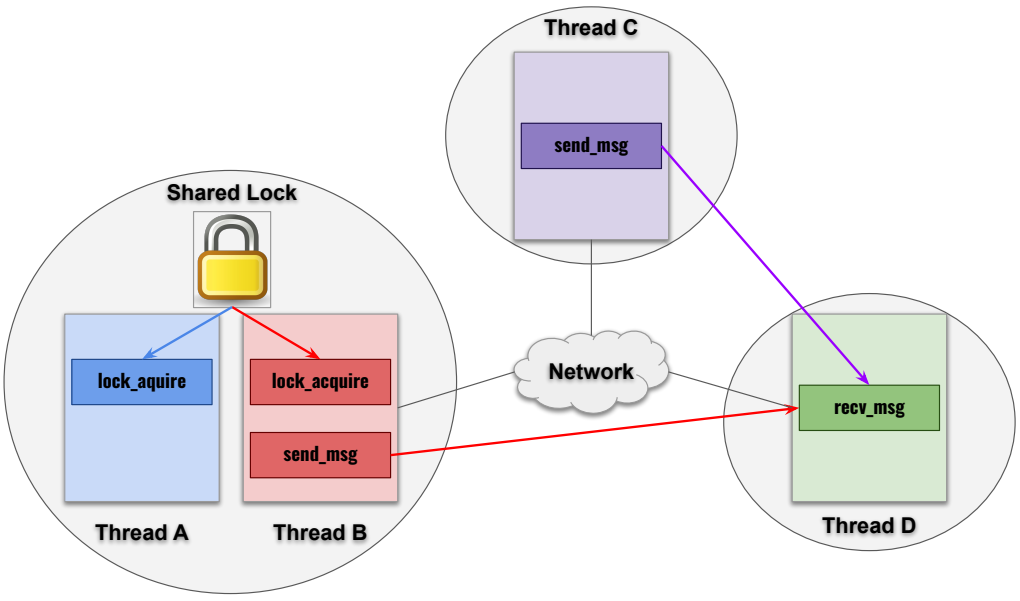

Figure 2. An Example scenario of concurrent events occurring in a non-deterministic order. Different colors are used to indicate the different thread's interaction with resources such as a shared lock or a network.

Meanwhile, Figure 3 describes how the VConMC controls concurrent events for the aforementioned example scenario. To control the non-deterministic order of concurrent events such as *lock_acquire* and *send_msg*, hooks are inserted and activated before the invocation for those events so that those non-deterministic events are interposed and blocked. Then, the model checker receives the interposed events. To gather all concurrent events, the model checker must wait long enough for the SUT to become quiescent. At the quiescent state, no more concurrent events will occur, and the SUT cannot make any progress until the model checker fires off one of the interposed concurrent events.

Subsequently, the model checker determines the order of those interposed concurrent events and accordingly schedules the execution of those events. For example, the model checker will receive an event that is `Thread B`'s attempt to invoke *send_msg* and another event that is `Thread C`'s attempt to invoke *send_msg*. Suppose interposing those two concurrent events are enough to make the SUT to reach the quiescent state. Then, the model checker determines which one of those two events is executed first. Once the order is determined, the model checker orchestrates interposed threads to execute concurrent events in a determined order by letting the first event execute first and the second event follow.

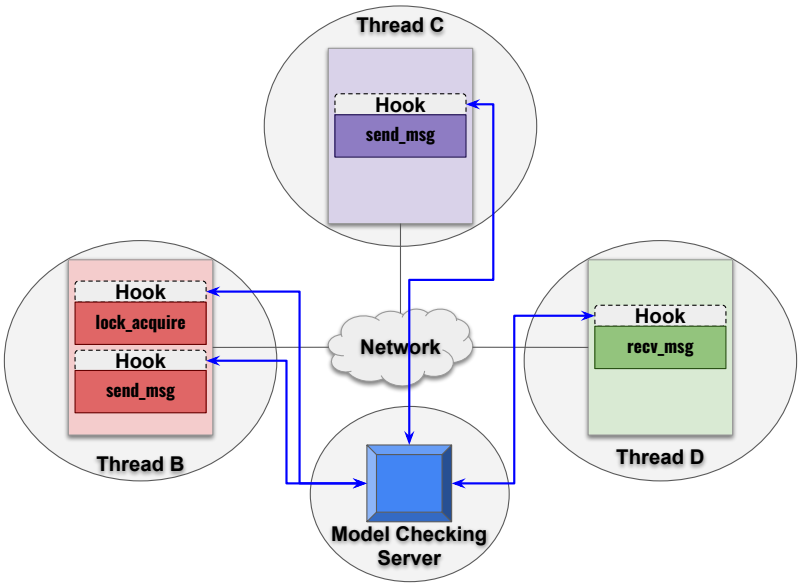

Figure 3. Interposition of concurrent events to execute those events in a deterministic order (Thread A is omitted for the better visibility).

3.4. Model Checkers

In this work, we employ a stateless state space exploration strategy. Implementation-level model checkers can take either state-based or stateless approaches to explore all possible interleaving non-deterministic events. The state-based exploration approach requires heavy modification of the SUT and the underlying system for the model checker to capture the state information. With a state-based search strategy, the SUT's state can be represented by the content in the memory. Thus, the model checker should be able to create a checkpoint to capture that information at a moment and freely rollback to one of the system snapshots to resume the state exploration. Meanwhile, the stateless exploration does not need such heavy modification but only the lightweight instrumentation of the SUT. The only changes required for the stateless search strategy are to keep track of each interleaving of non-deterministic event that has been explored because the execution order of non-deterministic concurrent events are used as the representation of the SUT's state. For the stateless search strategy, there is no need to take the snapshot of the SUT and the underlying system to roll back to one of intermediate states during the state space exploration. Instead, the stateless model checkers execute the sequence of non-deterministic events to reach one of intermediate states to continue the state space exploration. Therefore, state space exploration consists of path exploration attempts, each of which begins at the initial state and ends at each leaf state where there is no more enabled transition.

Figure 4 describes the stateless state space exploration in VConMC which is built on top of SAMC's framework [24]. The state exploration consists of repeated path exploration. Each path exploration begins at the specific initial state. Then, the model checker sees which events are interposed and regarded as enabled transitions that can be applied to the current state for the model checker to visit one of subsequent states. For example, at the initial state S_1 in Figure 4, there are two enabled events such as `Thread A`'s attempt to

invoke *lock_acquire* and `Thread B`'s attempt to invoke *lock_acquire*. The model checker can determine the order of those enabled events and pick the next event to make a transition to the next state.

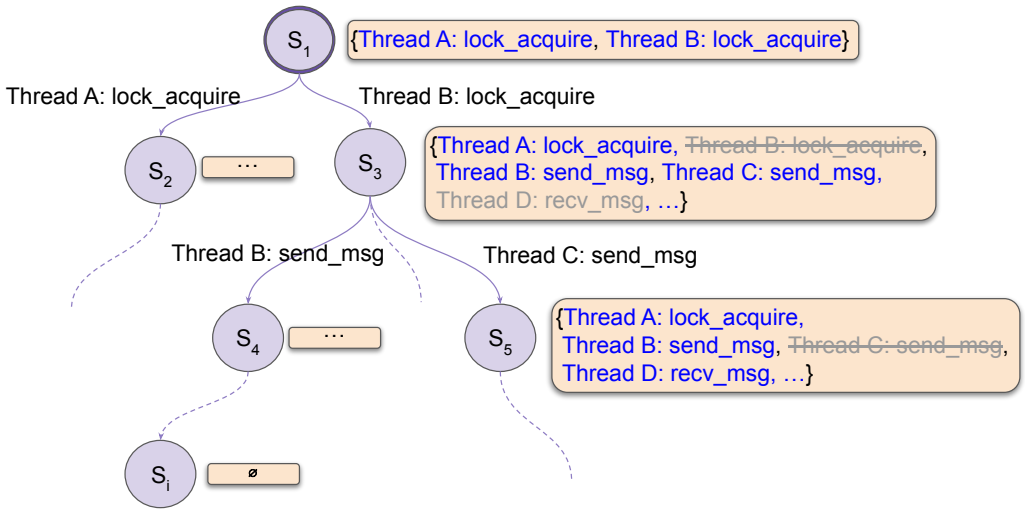

Figure 4. An example of stateless state space exploration. Starting at the initial state S_1, each traversal of a path allows the model checker to visit a series of states until it reaches one of leaf states, such as S_i. Each state in stateless model checking does not actually contain any in-memory snapshot of the SUT's state. Instead, every state in the stateless search strategy is a sequence of non-deterministic events that essentially are transitions between states.

At each state, some transition can become disabled, while some other events are newly enabled. For instance, at S_3 in Figure 4, as `Thread A` succeeds with obtaining a lock, `Thread A` can no longer invoke *lock_acquire*. So, `Thread A`'s attempt to invoke *lock_acquire* should be disabled in subsequent states by removing it from the list of enabled transitions. On the other hand, the model checker can newly interpose `Thread B`'s and `Thread C`'s attempts to send messages to `Thread D` as well as `Thread D`'s attempt to receive messages. Then, those events of `Thread B`, `Thread C`, and `Thread D` are newly appended to the list of enabled transitions. However, without sending any message to `Thread D`, there is no message for `Thread D` to receive. So, `Thread D`'s attempt to invoke *recv_msg* becomes temporarily disabled in S_3. Nevertheless, that *recv_msg* invocation event should not be permanently removed from the list of enabled transitions, as `Thread D`'s *recv_msg* invocation can be executed after `Thread B` or `Thread C` sends a message to `Thread D`. Indeed, once `Thread C` sends a message, `Thread D`'s *recv_msg* becomes enabled in S_5. When a path exploration reaches a leaf state, there is no more enabled transition to traverse, as shown in S_i. Once the model checker reaches one of leaf states, the path exploration is complete and the model checker restarts another path exploration run by rolling back the SUT to the initial state. By systematically and exhaustively exploring each path, VConMC can check if there is no consistency violation for all possible corner cases.

A model-checking server is the key component of VConMC that is responsible for state space exploration. The model-checking server communicates with the interposition layer. Each relevant event interposed on each node is notified to the model-checking server via the network. As previously explained, threads whose events are interposed by hooks are blocked until the model-checking server responds back. All concurrent events collected by the model-checking server are converted into a list of enabled transitions for state space exploration. Then, the model-checking server chooses one of the events from the list of enabled transitions and responds back to the corresponding thread so that the chosen event can be executed.

The algorithm for state space exploration is described in Algorithm 1. At a high level, the model-checking server explores the state space by repeating path exploration runs.

Every path exploration run searches through states by keep executing one of the enabled transitions at each state that has not been previously traversed. Each run is exploring a specific path traversal of the given state space. Note that the selection of transition to execute at Step 5 is performed in a first-in-first-out (FIFO) fashion (better heuristics using hints from the source code using program analysis may be utilized, but this is left out as future work). Once a path exploration run finishes, client operations executed during the path exploration run are analyzed for invariant checking. For each iteration of path exploration runs, the state of the SUT is reset by shutting down all related processes of the SUT and restarting them with the clean state. Additionally, it is possible to record and replay events to bring the SUT to a specific initial state once it is reset to the clean state.

Algorithm 1 VConMC's Stateless State Space Exploration Algorithm

1: A model-checking server initializes the state of a distributed key-value store as an SUT.
2: Starting at the initial state, the model-checking server starts up SUT's processes on each of the distributed nodes.
3: Each node gets blocked when concurrent events are interposed and notifies the model-checking server about the interposed events.
4: Meanwhile, the model-checking server keeps collecting concurrent events until the SUT becomes quiescent.
5: The model-checking server decides the next concurrent event to explore.
6: Every choice at each state is recorded to avoid repeated visits of the same next state.
7: The model-checking server directs a node to execute the chosen next concurrent event.
8: Go back to the step 3 above and repeat until there is no more next event to choose or the depth of the path traversal exceeds the upper bound.
9: The model-checking server analyzes the history of client requests using consistency oracles.
10: Go back to the first step above and repeat until there is no more path to explore.

At the end of each path exploration run, the invariant checker module of the model-checking server conducts consistency invariant checking as described in Algorithm 2. The invariant checker module interacts with the consistency oracles to conduct consistency invariant checking for the given consistency models (Step 6 and Step 7). The invariant checker module inspects the trace of client's requests and responses by feeding those as inputs into consistency oracles using `Update`($\langle Input \rangle$, $\langle CM \rangle$) API call, where $\langle CM \rangle$ is the parameter used to select consistency guarantees composing the consistency model (Step 7). Further details of consistency oracles' API calls are discussed in Section 3.5. For each read operation, the invariant checking module checks if the value seen by the read is correct under the given specific consistency model using the `Query`($\langle Input \rangle$, $\langle CM \rangle$) API call (Step 6).

Algorithm 2 VConMC's Consistency Invariant Checking using Consistency Oracles

1: An invariant checking module of a model-checking server starts up consistency oracles.
2: Encode the consistency model, $\langle CM \rangle$, as the composition of consistency guarantees.
3: A client's operation is sequentially obtained from the history of operations.
4: The invariant checking module determines whether the obtained client operation is a read operation or a write operation.
5: Convert the client operation into the $\langle Input \rangle$ for the consistency oracles.
6: For the read operation, the invariant checking module invokes `Query`($\langle Input \rangle$, $\langle CM \rangle$) and raises an alarm if the check detects a consistency violation.
7: The invariant checking module feeds the client operation into the consistency oracle by using the `Update`($\langle Input \rangle$, $\langle CM \rangle$) API call.
8: Go back to the first step and repeat the previous steps until there is no more client operation to check.

3.5. Consistency Oracles

In this work, we employ consistency oracles [27] to obtain the set of valid values to read for the given history of operations with respect to a consistency model that may provide the composition of several consistency guarantees. For the scope of this work, consistency violations occur if a read operation sees the value set by a write operation that breaks a guarantee promised by the given consistency model. Therefore, to detect consistency violations, the necessary first step is to know whether the value seen by each read operation is valid or not, which is the very purpose of consistency oracles.

Consistency oracles provide a very simple interface: (1) `Query` and (2) `Update`. More specifically, the interface calls are

$$\texttt{Query}(\langle \texttt{Input} \rangle, \langle \texttt{CM} \rangle)$$

where $\langle Input \rangle$ is the client operation and $\langle CM \rangle$ is the given consistency model, and

$$\texttt{Update}(\langle \texttt{Input} \rangle, \langle \texttt{CM} \rangle)$$

where $\langle Input \rangle$ is the client operation and $\langle CM \rangle$ is the given consistency model. When a client operation is applied by a system-under-test, then it needs to be added to a global history to further analyze the consistency behaviors of the system. `Update` is called by our model checker to update the consistency oracles by appending the given client operation in the consistency oracles' internal global history. It takes a client operation as an input but returns nothing. On the other hand, when a client operation is a read and obtains some value from the system-under-test, the `Query` interface needs to be invoked to determine whether the value read by the operation is valid or not. The `Query` interface will take the read operation as an input and return the set of valid values to read. So, the consistency invariant checking becomes as simple as a set membership test with consistency oracles.

Consistency oracles internally have multiple reference consistency modules to be able to compute all valid values for a consistency model composed of arbitrary consistency guarantees. The internal global history of operations in consistency oracles is shared by each reference module to analyze and produce the set of all valid values to read for the corresponding consistency guarantees. A query request for a given read operation is given with a parameter specifying which consistency guarantees are supposed to be provided by an SUT. Then, the intersection of each set of valid values produced by reference consistency modules is obtained as the final answer of consistency oracles. Each reference consistency module can be therefore regarded as an independent oracle for a specific consistency guarantee. Our prototype consistency oracle employs the careful selection of five consistency guarantees chosen from the practical consistency guarantees compiled by Doug Terry [30], which are strong consistency (SC), monotonic reads (MRs), bounded staleness (BS), read-my-writes (RM), and eventual consistency (EC).

The oracle maintains an append-only global log, which keeps growing as new inputs are appended to its end. The global log, denoted by $\langle GlobalLog \rangle$, can be defined as follows:

$$\langle GlobalLog \rangle = \langle Input_1 \rangle \, \langle Input_2 \rangle \, \ldots \, \langle Input_n \rangle$$

where $\langle Input_i \rangle$ for $i = 1, 2, \ldots, n$ ($n \in \mathbb{N}$) is the representation of each operation performed by clients. The $\langle Input_i \rangle$ is defined as follows:

$$\langle Input_i \rangle = \langle ClientID \rangle \, \langle Key \rangle \, \langle OpType \rangle \, \langle Value \rangle \, \langle Timestamp \rangle$$

where $\langle ClientID \rangle$ is the unique identifier of the client requesting an operation; $\langle Key \rangle$ is the unique identifier of the data entity; $\langle OpType \rangle$ is the type of operation, i.e., whether it is read or write; $\langle Value \rangle$ is the actual data value; and $\langle Timestamp \rangle$ is the time in milliseconds when the operation is requested. Here, $\langle Timestamp \rangle$ can be used when time-based consistency guarantees such as bounded staleness should be checked.

When invoking an interface `Update(`$\langle Input \rangle$`,`$\langle CM \rangle$`)` or `Query(`$\langle Input \rangle$`,`$\langle CM \rangle$`)`, the second parameter $\langle CM \rangle$ encodes a consistency model that is the composition of arbitrary consistency guarantees. To extend consistency oracles to support additional consistency guarantees, corresponding reference consistency modules can be added alongside other reference consistency modules without requiring modification to existing ones.

3.6. Consistency Invariant Checking

In this section, we present how to enable generic consistency invariant checking based on consistency oracles for implementation-level distributed systems model checkers. By leveraging consistency oracles, VConMC can be used for the verification of the various consistency models of the distributed key-value store supporting cloud storage services.

Because consistency models define the ordering rule for client requests reading or writing data from the distributed storage systems to follow, it is critical to infer the order of client requests to check if the SUT does not infringe constraints imposed by those consistency models. If guessing the order of requests is incorrect, consistency violations cannot be caught. For example, suppose two concurrent `PUT(X,1)` and `PUT(X,2)` requests are issued by clients, which are followed by a `GET(X)` request reading 1. Can we conclude that the given history of operations has violated a strong consistency model where every `GET` must read the latest `PUT`? Assuming that due to some network delay causes `PUT(X,2)` applies first and then `PUT(X,1)` applies next, `GET(X)` can read 1 without violating the constraint imposed by the strong consistency model. From the client perspective, it is not obvious to figure out the actual order of concurrent `PUT` requests without support from the server nodes.

The culprit that makes inferring the order of operations difficult is the distributed processing of concurrent events. Once client requests are issued, those requests travel through a network path, where involved networking and processing delays may vary for a number of reasons: there may be different numbers of processes running on each node on the data path, different hardware specifications and configurations for each server, different numbers of network flows transmitting through a network cable or a router, etc. For instance, the `PUT(X,2)` request may have sat in the buffer for incoming packets on one of the routers because that router may be in the middle of handling an overwhelming amount of network flows, while the `PUT(X,1)` request is handled by another router under much lower load so that the request can be processed right away. Precise estimation of how fast each request can be processed by a specific hop along the network path is very challenging. Moreover, as more client applications send requests, the degree of concurrency imposed by concurrent client requests increases, and failures may occur more frequently during request handling, which make it more complicated to correctly infer the order of operations.

Many distributed systems have the means to coordinate concurrent operations via consensus protocols to enforce some partial order among those operations. For example, Paxos [35] has been widely adapted by many real-world distributed systems [36,37]. ZooKeeper uses its own consensus protocol called ZAB, which stands for ZooKeeper Atomic Broadcast [38]. Raft [39] has been accepted by many distributed systems from the industry, and etcd is one example. MongoDB also employs a variant of the Raft protocol by extending the vanilla Raft to incorporate pull-based replication [40]. The implementation of consensus algorithms employs a logical clock to distinguish each operation reflecting the order of operations being appended in the log. With the logical timestamp, the servers can find out the partial-order information among concurrent operations. As an example, ZooKeeper, the primary open-source distributed key-value system, uses `zxid` as a logical timestamp to serialize each write (i.e., `Put`).

This work utilizes the key observation that there are distributed systems exposing the partial-order information of operations via API by revealing a logical timestamp associated with each client request. For instance, ZooKeeper's API for `Get` and `Put` are `getData` and `setData`. For the invocation of those API calls, ZooKeeper's client stub returns the result of

the API call along with the metadata, including ordering-related information, `zxid`. For a `getData` process, the API call returns the `zxid` of the `setData`, from which the `getData` reads its value. On the other hand, for a `setData`, it returns the `zxid` assigned to a `setData`. That is, the returned `zxid` for a `getData` implies the order of that `getData` relative to the `setData` it saw, and the returned `zxid` for a `setData` implies the order of that `setData` with respect to other `setData` requests. Note that it is not possible to infer the total order of every concurrent request because `zxid` does not distinctively determine the exact order of the `getData`. Nevertheless, the `zxid` values returned by ZooKeeper still let us infer the partial-order information that is enough to be used for consistency invariant checking.

Without knowing the actual order of concurrent events, it is difficult to conduct consistency invariant checking for various consistency models. Consider the state-of-the-art consistency fuzzer, Jepsen [8], which uses the blackbox approach for consistency invariant checking. Suppose that such a blackbox approach for consistency invariant checking is leveraged to test ZooKeeper. Note that ZooKeeper employs a composited consistency model. Its consistency model provides three consistency guarantees: monotonic reads, consistent prefix, and bounded staleness. Figure 5 illustrates the limitation of the the aforementioned method when it checks for a monotonic reads consistency guarantee. The figure shows what can happen when there are multiple concurrent clients writing and reading. Initially, there are three `setData` requests performed by three writers, C1, C2, and C3. As those requests are concurrent, clients would not know in which order `setData` requests are applied without knowing the internal ordering-related information. Meanwhile, each reader has its own session and issues two sequential `getData` requests. Those sessions run concurrently and independently, which mean clients do not know when `getData` requests are actually handled and different `getData` requests across sessions are not co-related.

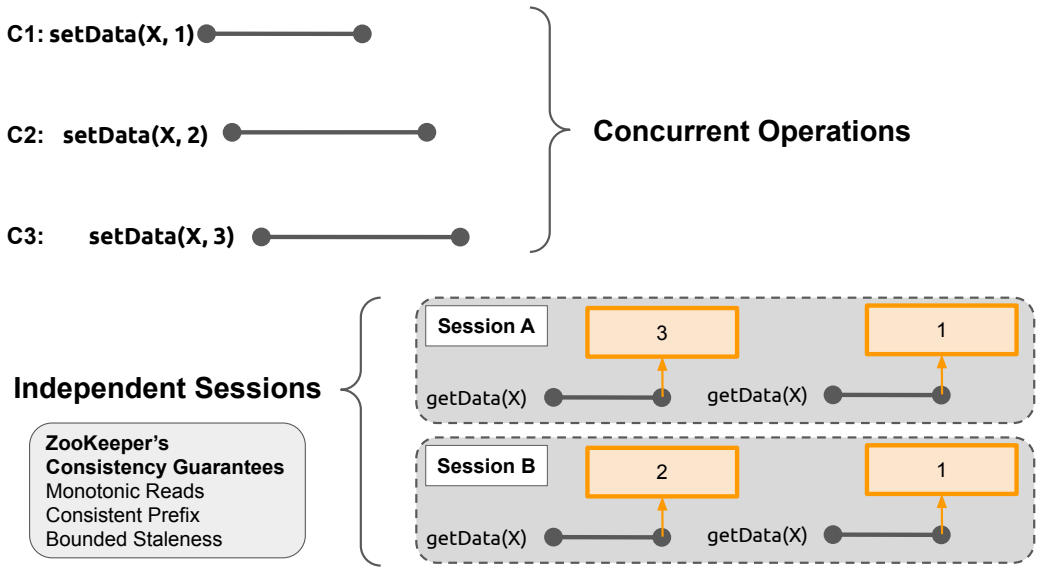

Figure 5. History from the client perspective without knowing the actual order of concurrent operations. Clients C1, C2, and C3 issued `setData` requests, which yield concurrent operations to apply in an arbitrary order via the ZAB protocol. Once those operations are applied, two independent sessions are started, and clients in each session issue a couple of `getData` requests sequentially. In Session A, the first and second `getData` invocations read the value of X that are 3 and 1, respectively. On the other hand, in Session B, the value 2 and 1 are read for the first and second `getData` for the key X, respectively. ZooKeeper's consistency guarantees are specified to involve monotonic reads, consistent prefix, and bounded staleness.

How can one tell if each `getData` request does not violate monotonic reads, one of ZooKeeper's consistency guarantees? The traditional approach like the one used by Jepsen simply performs the exhaustive search through all possible interleaving of concurrent

events. If at least one of those permutations does not violate the given consistency model, it concludes that there was no consistency violation that occurred under the given scenario.

The fundamental problem of having difficulties in consistency invariant checking without knowing the exact ordering information about concurrent operations is that there can be too many possible interleaving scenarios. Figure 6 shows the illustration of applying such an approach to the same scenario depicted by Figure 5. With three concurrent writes, there are 6 possible permutation scenarios. Only when we find the permutation scenario that satisfies the given consistency guarantee can we conclude that there has been no consistency violation. For example, the fourth and sixth permutations in the Figure 6 make sense under the monotonic reads consistency model for both sessions. Thus, after either the fourth or the sixth permutation is checked, we can ensure that the given session histories have not involved any consistency violations. Although the illustrated example with 6 permutation scenarios seems manageable, it is not difficult to see that the number of possible permutation scenarios can exponentially grow as the number of concurrent requests increases. If there were 10 concurrent write operations, then there would have been 3,628,800 possible permutation scenarios! In addition, as failures may occur anytime, the number of permutations to check can quickly become overwhelming. The conventional consistency invariant checking approach may not complete within the reasonable duration due to the computational cost depending on the number of client requests and failure injection.

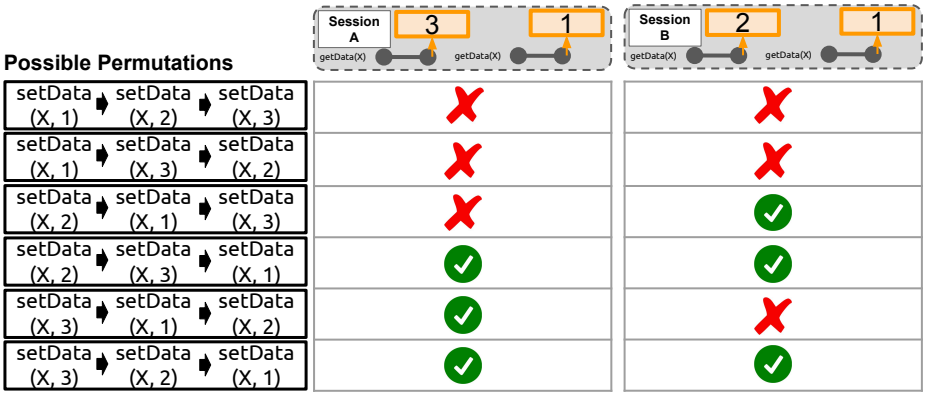

Figure 6. All possible permutations of concurrent operations without knowing the actual order of concurrent operations. Under each scenario, the given history of operations from the client perspective may be determined to be correct or incorrect with respect to monotonic reads, which is one of ZooKeeper's consistency guarantees.

If the partial-order information was provided via API, the consistency invariant checking would not have suffered from the aforementioned problem. For instance, partial-order information reflected in the zxid allows us to directly infer the possible permutation that conforms to the actual order without the exhaustive search. Figure 7 shows that zxid was returned by each `setData` and `getData`. The actual order of concurrent operations that are revealed by the zxid values is as follows: `setData(X,3)`, `setData(X,1)` and then `setData(X,2)`. Session A shows that the first `getData` reads 3 written by the `setData` of zxid 1004 and the second `getData` reads 1 written by the `setData` of zxid 1005. Thus, each getData does not read the stale value and, therefore, the observed history conforms to the given consistency guarantee, monotonic reads. Meanwhile, Session B shows that the first `getData` reads 2 written by the `setData` of zxid 2001, but the second `getData` reads 1 written by the `setData` of zxid 1005. Hence, Session B's history is showing the consistency violation for the given consistency model, monotonic reads.

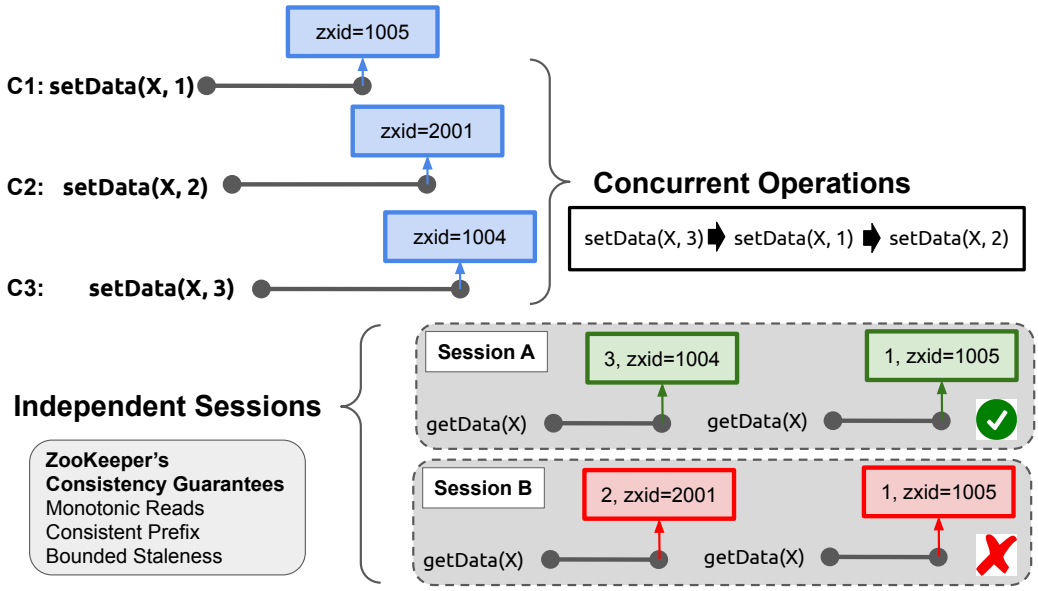

Figure 7. History from the client perspective when knowing the order of concurrent operations allows us to correctly determine consistency guarantees. Logical timestamp values, which are zxid values, are shown in the figure for each ZooKeeper API request. Also, each session is marked with either valid or not valid using green check marks and red cross marks, respectively. Session A is permitted under the monotonic reads consistency model, while Session B is the violation of that consistency model.

To conduct the consistency invariant checking for a session guarantee such as a monotonic reads consistency guarantee, we need to construct the history of operation for each session in a way that the order of operations should honor the partial-order information revealed via API. For example, in ZooKeeper, zxid is returned for each `getData` and `setData`, and it can be used to construct the history of operation without violating any partial order determined by zxid. In ZooKeeper, zxid determines the partial order for every `setData`, while it does not say anything about each `getData`. Meanwhile, within a session, each `getData` is issued sequentially, which gives us another set of partial-order information to keep. Therefore, we construct the history of operation for each session in the following steps: (1) During the model-checking process, `setData` requests should be recorded along with their zxid, request, and response time; (2) During the model-checking process, `getData` requests of each specific session should be recorded along with their request and response time; (3) For each session, the session's history of operation should be constructed by appending `setData` in a way that the order determined by zxid is maintained; (4) For each `getData` of the session, insert it into the position where all previous `setData` requests have the response time earlier than its request time and all previous `getData` requests are those ones issued before it within the same session. The output of the procedure above is the history of operation that can be fed into the consistency oracles. Here, the required assumption is that each write should write a unique value.

3.7. Implementation

A prototype system is built and employed to find bugs in a real-world key-value store system. We implemented consistency oracle as 0.4K lines of code in Java. It embeds five read-consistency guarantees such as eventual consistency, monotonic reads, bounded staleness, strong consistency, and read-my-write. We currently do not support the consistent prefix guarantee as only the systems that update data entirely are considered. On the other hand, model checker was also implemented in Java by modifying SAMC. The verifier component using our consistency oracle was newly added. In total, we introduced 1.1K lines of new code on top of SAMC's.

SAMC's design can be broken into three components: (1) an interposition layer; (2) a model-checking server; and (3) a test generator. VConMC modified SAMC in all three components to customize it for consistency verification. First, the interposition layer is where the system-under-test is instrumented with hooks, which interpose non-deterministic events and send to the model-checking server. The SAMC source code is implemented to verify ZooKeeper's leader election protocol. Also, SAMC's hooks only interpose the network communication operations like sending and receiving. VConMC focuses on exploring crash failure recovery mechanisms after leader election is finished. Thus, VConMC adds the hooks in the deeper part of the code to interpose on two additional operation types: (1) operations taking a snapshot (namely, takeSnapshot()); and (2) operations synchronizing with the leader (namely, syncWithLeader()). Second, for adding a couple of operation types, the model-checking server is modified to support those during its stateless search process. Third, the test generator of the original SAMC has been modified to record the trace of client API calls, which is used to perform consistency invariant checking against consistency oracles. SAMC does not have the component specific to check for consistency models, so the test generator is enhanced with the new verifier component for consistency invariant checking, and the new verifier component performs a series of reads after the workload generation has been finished and then runs consistency invariant checking.

4. Results

Using the aforementioned consistency invariant checking methodology along with our model checking tool, we found that our tool can detect violations of monotonic reads consistency guarantee, which are caused by several known bugs in the real-world distributed key-value store system, ZooKeeper. For all bugs, our consistency invariant checking does not require an exhaustive search for every possible interleaving of concurrent events.

We analyzed bugs which were previously reported in the ZooKeeper's bug database, which are ZooKeeper-1154, ZooKeeper-1319, and ZooKeeper-1549. The summary of the bugs we found is given in Table 1. As bugs are all labeled as either "Blocker" or "Major" in terms of the priority for fixing bugs, the bugs are all serious and critical ones. The duration taken to fix bugs varied from 3 days to unresolved for the last decade. For the latter, developers gave up fixing bugs and instead decided to move onto the newer version while deprecating the usage of the old versions. Symptoms were all related to data integrity. Data get lost or some corrupted data remain in the data store. Thus, it is expected that any applications operate based on these data may result in incorrect behavior, which may cause serious impacts. The root causes were mostly due to programmers' mistakes in implementing complex synchronization protocols. The synchronization protocol's bugs were usually manifested during the failure recovery. We believe such an error is common because it is very difficult to find out all corner cases in advance to testing. Consequently, we think the model checking approaches will be very effective tools for finding consistency-violating bugs.

Table 1. Summary of bugs that VConMC found.

Bug ID	Severity	Duration	Symptoms	Root Cause
1154	Blocker	21 days	Data Loss	Resync Errors
1319	Blocker	3 days	Data Loss	Replication Errors
1549	Major	Unresolved	Corrupted Data	Truncation Errors

These bugs are all labelled as serious bugs as they are regarded to have either "Blocker" or "Major" severity. All bugs that we found result in data integrity violations, either losing data or having corrupted data. Also, the root causes were in the synchronization mechanisms where resync, replication, or truncation should have correctly happened.

ZooKeeper-1154 is the bug that manifests when a node that has seen the highest "zxid", the transaction ID in ZooKeeper's parlance, is not present during the leader election during which a new leader gets elected. Then, the node that has seen the highest zxid that the new leader has not seen may join again later. The joining node does not receive any

truncation request for the zxid that the new leader has not seen. Thus, only the joining node will return the data applied with the zxid, which has not been seen by the new leader, while the new leader does not hold that data. The reproduction step involves a cluster of three server nodes. Subsequently, the following steps are needed: (1) create one entry, (2) async write to one node A but not to the other two nodes B and C, (3) stop A, B, and C (4) restart B and C, (5) B and C form the quorum and B becomes a new leader, and (6) restart A and let it apply the async write to the data tree and join the quorum. By issuing read operations for each key on every replica, we can detect the violation of consistency guarantees. We verified the bug resulting in the violation of not only monotonic reads but also eventual consistency guarantees. To detect the violation of monotonic reads, we ensure issuing read operations for each key on every replica and perform one more read operation checking if all clients read the version of values that are monotonically increasing. As a result, by reading the data that are more stale then the data returned by the previous read, it can detect the violation of monotonic read consistency guarantee.

ZooKeeper-1319 is a regression bug introduced by one patch, which results into a leader sending empty diff to followers during the addition of new members into the cluster. Consequently, one can read the data on the leader but not on the followers as the view becomes inconsistent between different replicas. Regression was introduced into one of the methods that is supposed to correctly set the variable "lastProposed", which is the transaction number for the last operation proposed to be applied. The proposed operation must be replicated to new members, but due to the incorrectly set value for the variable, those operations that are only applied on the leader are replicated to new members. The bug could be reproduced by the following steps: (1) start up one server cluster, (2) create some records, (3) shut it down, and (4) start up two server clusters including the first server with its existing data. The verification found the violation even for the eventual consistency guarantee as the read attempt from the second server will return no valid value at all.

ZooKeeper-1549 is the bug where several crash-reboots causes the older primary to contain data that are supposed to be discarded in its dirty snapshot, leading to an inconsistent in-memory data tree. The source of the bug was in the incorrect implementation of recovery logic, resulting in the production of the dirty snapshot containing the data that are supposed to be discarded during the crash failure recovery. The summarized bug-reproducing step for ZooKeeper-1549 is as follows: (1) write to a key on the leader only, (2) crash all, (3) reboot the old leader and one of the followers, (4) crash all again before sync, at which point the dirty snapshot is produced by the old leader, (5) start two followers and have them write to another key, and (6) start the old leader as a follower this time. Now, even after syncing with the new leader, the old leader still loads the in-memory data tree from the dirty snapshot. Therefore, it leads to inconsistent values, which can be detected via consistency invariant checking for monotonic reads.

As a running example, the following gives more detailed bug reproduction steps for ZooKeeper-1549:

1. Initially, three nodes start and one becomes leader while the other two become followers. Also, all three nodes have the same two key-value pairs after executing create("k1", "k1v0") and create("k2", "k2v0").
2. Crash Node B
3. Crash Node C
4. Client invokes an API call setData("k1", "k1v1")
5. Node A executes the setData("k1", "k1v1")
6. Crash Node A
7. Start Node A
8. Start Node B
9. Start Node C
10. Node A executes takeSnapshot()
11. Crash Node B
12. Crash Node C

13. Crash Node A
14. Start Node B
15. Start Node C
16. Node B executes takeSnapshot()
17. Node C executes syncWithLeader()
18. Client invokes an API call setData("k2", "k2v1")
19. Node B executes commit()
20. Node C executes commit()
21. Start Node A
22. Node A syncWithLeader()

At Step 1, the values of keys "k1" and "k2" are set to "k1v0" and "k2v0", respectively. Steps 2 to 5 crash follower nodes and have only the leader A update the key "k1" to have a value, say "k1v1". Then, crash the leader as well at Step 6. Steps 7 to 9 restart all nodes, including the old leader and two old followers. Steps 10 to 13 have the old leader A take a snapshot, which includes the key "k1" with the value "k1v1" but crashes all nodes before nodes become synchronized. Now, only the old leader A has the key "k1" of value "k1v1", while the other two old followers' value of the key "k1" are "k1v0". Steps 14 to 17 start the old followers, and one of them becomes a new leader; suppose Node B becomes the new leader, and have Node B and C synchronize. Steps 18 to 20 write the value "k2v1" to the key "k2". Finally, Steps 21 and 22 start A as a follower and synchronize with the new leader Node B. The bug manifests because the synchronization procedure only replicates the update to the key "k2" but does not rollback the update performed to the key "k1" at Step 5. As a result, Node A has the value "k1v1" for the key "k1" and the value "k2v1" for the key "k2". However, Node B and C have the value "k1v0" for the key "k1" and the value "k2v1" for the key "k2". The verifier reads "k1v1" as the value of the key "k1" on Node A; later, it then reads "k1v0" as the value of the key "k1" on Node B. Note that the value "k1v1" is more recent than the value "k1v0" for the key "k1". Thus, it causes the violation of monotonic reads consistency guarantees, which the verifier can detect after conducting the consistency invariant checking procedure in the same way as ZooKeeper-1154's scenario described above.

Discussion

ZooKeeper offers logical clock timestamps called zxid, and this work demonstrates that leveraging it is possible. All bugs found and summarized previously are detected by using zxid to determine the order of concurrent `setData`, and the consistency invariant checking used does not involve an exhaustive search through all possible interleaving. Consequently, it shows that if the target system exposes the internal ordering-related information, we can avoid the expensive exhaustive search for consistency invariant checking. However, we have not checked for two other consistency guarantees of ZooKeeper. The consistent prefix for ZooKeeper is not suitable to check separately as ZooKeeper's write entirely replaces each data item. Timing-based approaches may have been checked if ZooKeeper had provided some metadata revealing the timing of handling each request.

Furthermore, this work also shows that there are some distributed key-value stores providing useful partial-order information to clients by replying the result along with the metadata that is actually used to internally order concurrent operations. It is well known that many systems already employ means to coordinate concurrent updates such as Lamport clock, vector clock, etc. [41–44]. We believe the API of various distributed key-value store systems can be extended to reveal some of the internal metadata related to the order of operations not only for consistency invariant checking but also for real-time auditing purposes.

Failure injection does not increase the severity of the state explosion issue, because our consistency invariant checking only consider requests that complete with a response from the system. More specifically, without receiving the response for some request, we do not have zxid for the corresponding request and do not add that request to the history of

operation. This is a beneficial side effect of utilizing the partial-order information revealed via API—it indirectly filters out failed operations from consistency invariant checking.

Consistency oracles should be used with the several assumptions. First, every write operation should use non-repeated values. If write operations are allowed to repeatedly write the same value, finding the write operation from which the given read operation obtains the value will become computationally very inefficient due to too many possible matching pairs of read and write operations. Our approach is to encode data along with a unique ID for each operation. For example, the combination of a client ID and a monotonically increasing sequence number of each client can form a unique ID for each write operation. Second, consistency oracles also require the exact order of each operation. For the systems that do not reveal the internal ordering of operations, they cannot be directly integrated with consistency oracles. Third, consistency oracles currently support only those systems that entirely overwrite a data item for each write. Consistency prefix in this setting is equivalent to eventual consistency. Supporting the consistency invariant checking of the consistent prefix for systems that may partially update data is left as future work.

Note that there is still more work to perform to fully understand the potential of using partial-order information obtained via API. We have not yet comprehensively inspected other systems and consistency guarantees, except for monotonic reads in ZooKeeper. In addition, it will be useful to find an automated way to extend the API to expose useful partial-order information used internally.

Also, one limitation of this work is that it focuses on a system that does not support transactions containing multiple operations. Thus, it is left as future work to extend the proposed methodology to transactional systems where multiple reads or writes can be contained and executed. Another limitation to point out is that this work has put emphasis on integrating consistency invariant checking based on consistency oracles with implementation-level model checking. Therefore, it has not explored how to manifest previously unknown consistency bugs more effectively by improving the underlying implementation-level model checker. Exploring the utilization of various coverage-based testing approaches or advanced program analysis techniques as the heuristics for more effective state space exploration strategies remains as future work.

Note that non-deterministic behaviors are relatively controlled in this work. So, the non-deterministic behaviors of the system-under-test may be observed sometimes, even if careful instrumentation is performed to tightly control them. One notable way in which such non-deterministic behaviors occur is that sometimes some concurrent event that was seen in the previous visit of a certain state may not happen again in the next visit to the same state for another path exploration. This is highly likely because some of the causal events did not happen at the correct timing due to the incomplete control. Also, it is possible that some causal event may not occur within the timeout period set by our synchronous event hooking method. In such a case, our path exploration will check if the current state has been visited last time. If so, it also checks if the list of concurrent events collected for the previous visit of the state is same as the list of concurrent events collected for the current visit. Otherwise, VConMC stops path exploration and retries. If the retry still fails, then VConMC gives up after recording the trace and the result of the failed path exploration so that the developer can inspect the source code more closely and add more controls or adjust the timeout period.

5. Related Work

5.1. Distributed Systems Model Checking

Implementation-level model checking has been applied to find bugs in distributed systems. Previous proposals for implementation-level distributed systems model checking mostly focus on seeking out the violation of protocol-specific safety properties [18–22]. Those previous proposals can detect deadlock and protocol-specific, implementation-specific, or language-specific errors. However, the limitation of those proposals was the

lack of smart state space reduction or abstraction techniques. Researchers quickly noticed this limitation and explored various smart approaches to significantly reduce the size of the state space to search through [23–25]. However, those proposals do not consider how to detect consistency violations of the distributed systems. CrystalBall checks for inconsistency in distributed systems [45]. However, its definition of being consistent and inconsistent is different from data consistency, as it says the system is inconsistent if there is no global tree topology that satisfies every node's local view. Due to such a restrictive definition about consistency, it is not generic and cannot be applied to find data consistency violations. More recently, researchers have developed a formal verification methodology and tool that can generate an implementation along with an abstract model for model checking [16]. The technique allows developers to specify their distributed systems implementation as an abstract model using formal language. Then, the PGo compiler toolchain will generate both the executable and the specification to model check. Nevertheless, PGo cannot be applied to verify the existing implementation of consistency models. Also, there is another line of verification effort for distributed systems by using symbolic execution supported by the language runtime and underlying framework, as performed in Serval [46]. Nevertheless, none of previously proposed implementation-level model checkers can detect the violation of a relaxed consistency model.

5.2. Distributed Systems Testing

There have been many research ideas proposing novel testing techniques to effectively find concurrency bugs in distributed systems. Fault-injection techniques such as crash injections or network-partition injection have been explored extensively [47–49]. However, those proposals focus on finding out fault injection timing by analyzing the source code and do not support invariant checking for various data consistency models. Thread interleaving is usually the root cause of the source and therefore deterministically exploring the order of thread execution allows developers to reliably reproduce concurrency bugs. FlakeRepro proposes the technique combining static and dynamic analysis to automatically add such a capability to the existing implementation of distributed systems [50]. Another work attempts to explore the sequence of non-deterministic events systematically and extensively, which has been applied for modern cluster managers [51]. However, those proposals do not look for data consistency violations caused by concurrency bugs. Instead of using systematic testing, GFuzz proposes employing a fuzzing technique that has been developed instead of using the deterministic ordering of concurrent events [9]. Also, researchers explored a random testing technique by randomly sampling partial order to more accurately pinpoint the sequence of event execution that is more likely to cause errors [52]. Nevertheless, none of the previous proposals on distributed systems testing offer a deterministic means to verify consistency model implementation.

5.3. Consistency Checking

Benchmarking the consistency model implementation has been explored previously by researchers [53–56]. Also, the consistency model's boundness has been probablistically analyzed via the modeling and analytical measurement technique [57]. Various consistency model verification techniques are developed based on the dependency-graph analysis looking for cycles in the directed graph of read and write operations [58–61]. Techniques to monitor the consistency behavior of an untrusted cloud are developed where an untrusted storage provider may be compromised so that it violates consistency guarantees [62–64]. FoundationDB developers have implemented a deterministic simulation mode to find bugs that are caused by non-deterministic events by using actor models and a simulated source of a non-deterministic event [65]. Because consistency-checking algorithms additionally require the overhead for analysis, researchers explored a technique to apply verifiable computation to check the correctness of serializability [66]. Instead of developing a consistency-checking algorithm as an arbitrary invariant checker, differential testing has been employed [67]. Viper has been developed to check for snapshot isolation by checking

for a polygraph-based transaction dependency [68]. Nonetheless, those previous proposals only include the consistency-checking mechanism but do not have ability to drive the system to the corner case. Moreover, consistency models that those existing techniques can check for are limited to a few models statically embedded in the checking algorithm.

5.4. Blockchain Databases

Recently, blockchains have gained a lot of attention to be used as a trusted distributed key-value store on top of peer-to-peer networks, even in the presence of a malicious participant [69–72]. Blockchain databases are designed to have the majority of participants agree on a single global history, that is, the chain of blocks containing transactions, via consensus. Therefore, those systems only offering serializability and relaxed consistency are not an issue. Nonetheless, it may be worthwhile to explore if there are any bugs in blockchain implementation that cause the violation of serializability by using the proposed method in the future.

6. Conclusions

Consistency verification has been actively and extensively explored recently. Consistency bugs are caused by concurrency and non-deterministic events occurring in a distributed system. Model checking is the useful verification and bug-finding tool that can systematically and exhaustively explore various corner cases. Nevertheless, there has been no previous proposal that employs model checking specifically to support the capability of finding software and design flaws causing consistency violations. As a result, offering consistency invariant checking for the model checking of distributed systems has not been explored previously. In this work, consistency oracles have been integrated with concrete model checking, and we verified several consistency bugs in a widely deployed distributed system such as ZooKeeper. The key enabler is leveraging partial-order exposing API for consistency invariant checking. Based on the current trend of adapting the distributed key-value store for various cloud services, the usage of model checking to find bugs violating consistency guarantees will be increasingly demanded by cloud service providers. Therefore, we believe consistency invariant checking using consistency oracles is going to be an important key technique for implementing reliable cloud services.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (No. RS-2023-00244368), by the research fund of Hanyang University (HY-2022-2917) and by the 2022 Research Institute of Engineering and Technology Research Fund at Hanyang University ERICA.

Data Availability Statement: The data can be shared up on request.

Conflicts of Interest: The author is affiliated with Hanyang University ERICA. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; nor in the decision to publish the results.

References

1. Hunt, P.; Konar, M.; Junqueira, F.P.; Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 23–25 June 2010; p. 11.
2. MongoDB. Available online: https://www.mongodb.com/ (accessed on 13 October 2023).
3. Lakshman, A.; Malik, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40. [CrossRef]
4. Apache HBase. Available online: https://hbase.apache.org (accessed on 10 October 2023).
5. Gunawi, H.S.; Hao, M.; Leesatapornwongsa, T.; Patana-anake, T.; Do, T.; Adityatama, J.; Eliazar, K.J.; Laksono, A.; Lukman, J.F.; Martin, V.; et al. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC), Seattle, WA, USA, 3–5 November 2014.
6. Kim, B.H.; Kim, T.; Lie, D. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models. In Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, 11–13 July 2022; Schindler, J., Zilberman, N., Eds.; USENIX Association: Berkeley, CA, USA, 2022; pp. 383–398.

7. Lloyd, W.; Freedman, M.J.; Kaminsky, M.; Andersen, D.G. Stronger semantics for low-latency geo-replicated storage. In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, Berkeley, CA, USA, 2–5 April 2013; pp. 313–328.
8. Kingsbury, K. Distributed Systems Safety Research. Available online: https://jepsen.io/ (accessed on 13 October 2023).
9. Liu, Z.; Xia, S.; Liang, Y.; Song, L.; Hu, H. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 28 February–4 March 2022; pp. 888–902. [CrossRef]
10. Ba, J.; Böhme, M.; Mirzamomen, Z.; Roychoudhury, A. Stateful Greybox Fuzzing. In Proceedings of the 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, 10–12 August 2022; pp. 3255–3272.
11. Jeong, D.R.; Lee, B.; Shin, I.; Kwon, Y. SEGFUZZ: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, 22–24 May 2023; pp. 2104–2121. [CrossRef]
12. Andronidis, A.; Cadar, C. SnapFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, 18–22 July 2022; pp. 340–351. [CrossRef]
13. Jiang, Z.; Bai, J.; Lu, K.; Hu, S. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In Proceedings of the 29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, CA, USA, 24–28 April 2022.
14. Wilcox, J.R.; Woos, D.; Panchekha, P.; Tatlock, Z.; Wang, X.; Ernst, M.D.; Anderson, T. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not.* **2015**, *50*, 357–368. [CrossRef]
15. Hawblitzel, C.; Howell, J.; Kapritsos, M.; Lorch, J.; Parno, B.; Roberts, M.L.; Setty, S.; Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP). ACM—Association for Computing Machinery, Monterey, CA, USA, 4–7 October 2015.
16. Hackett, F.; Hosseini, S.; Costa, R.; Do, M.; Beschastnikh, I. Compiling Distributed System Models with PGo. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023), Vancouver, BC, USA, 25–29 March 2023; Volume 2, pp. 159–175. [CrossRef]
17. Godefroid, P. Model Checking for Programming Languages Using VeriSoft. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, 15–17 January 1997; pp. 174–186. [CrossRef]
18. Guo, H.; Wu, M.; Zhou, L.; Hu, G.; Yang, J.; Zhang, L. Practical software model checking via dynamic interface reduction. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, 23–26 October 2011; pp. 265–278.
19. Guerraoui, R.; Yabandeh, M. Model checking a networked system without the network. In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, 30 March–1 April 2011; p. 225.
20. Lin, H.; Yang, M.; Long, F.; Zhang, L.; Zhou, L. MODIST: Transparent model checking of unmodified distributed systems. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, 22–24 April 2009.
21. Killian, C.; Anderson, J.W.; Jhala, R.; Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation, Cambridge, MA, USA, 11–13 April 2007.
22. Musuvathi, M.; Park, D.Y.W.; Chou, A.; Engler, D.R.; Dill, D.L. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Oper. Syst. Rev.* **2002**, *36*, 75–88. [CrossRef]
23. Simsa, J.; Bryant, R.; Gibson, G. dBug: Systematic Evaluation of Distributed Systems. In Proceedings of the 5th International Conference on Systems Software Verification, Vancouver, BC, Canada, 6–7 October 2010; p. 3.
24. Leesatapornwongsa, T.; Hao, M.; Joshi, P.; Lukman, J.F.; Gunawi, H.S. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, USA, 6–8 October 2014; pp. 399–414.
25. Lukman, J.F.; Ke, H.; Stuardo, C.A.; Suminto, R.O.; Kurniawan, D.H.; Simon, D.; Priambada, S.; Tian, C.; Ye, F.; Leesatapornwongsa, T.; et al. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, 25–28 March 2019; pp. 20:1–20:16. [CrossRef]
26. Gorjiara, H.; Xu, G.H.; Demsky, B. Jaaru: Efficiently Model Checking Persistent Memory Programs. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Online, 19–23 April 2021; pp. 415–428. [CrossRef]
27. Kim, B.H.; Oh, S.; Lie, D. Consistency Oracles: Towards an Interactive and Flexible Consistency Model Specification. In Proceedings of the 16th Workshop on Hot Topics in Operating Systems, Whistler, BC, Canada, 7–10 May 2017; pp. 82–87. [CrossRef]
28. Brewer, E. CAP twelve years later: How the "rules" have changed. *Computer* **2012**, *45*, 23–29. [CrossRef]
29. Abadi, D. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer* **2012**, *45*, 37–42. [CrossRef]
30. Terry, D. Replicated Data Consistency Explained Through Baseball. *Commun. ACM* **2013**, *56*, 82–89. [CrossRef]
31. Higham, L.; Kawash, J.; Verwaal, N. Defining and comparing memory consistency models. In Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems, New Orleans, LO, USA, 1–3 October 1997.

32. Herlihy, M.P.; Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **1990**, *12*, 463–492. [CrossRef]
33. Robinson, H. Consensus Protocols: Two-Phase Commit. 2011. Available online: https://www.the-paper-trail.org/post/2008-11-27-consensus-protocols-two-phase-commit/ (accessed on 3 March 2024).
34. Vogels, W. Eventually Consistent. *Commun. ACM* **2008**, *52*, 40–44. [CrossRef]
35. Lamport, L. The Part-Time Parliament. *ACM Trans. Comput. Syst.* **1998**, *16*, 133–169. [CrossRef]
36. Burrows, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle, WA, USA, 6–8 November 2006; Bershad, B.N., Mogul, J.C., Eds.; USENIX Association: Berkeley, CA, USA, 2006; pp. 335–350.
37. Corbett, J.C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; et al. Spanner: Google's Globally-Distributed Database. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, 8–10 October 2012; Thekkath, C., Vahdat, A., Eds.; USENIX Association: Berkeley, CA, USA, 2012; pp. 251–264.
38. Reed, B.C.; Junqueira, F.P. A simple totally ordered broadcast protocol. In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS'08, Yorktown Heights, NY, USA, 15–17 September 2008; Dekel, E., Chockler, G.V., Eds.; ACM: New York, NY, USA, 2008; pp. 2:1–2:6. [CrossRef]
39. Ongaro, D.; Ousterhout, J.K. In Search of an Understandable Consensus Algorithm. In Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC'14, Philadelphia, PA, USA, 19–20 June 2014; Gibson, G., Zeldovich, N., Eds.; USENIX Association: Berkeley, CA, USA, 2014; pp. 305–319.
40. Zhou, S.; Mu, S. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, Boston, MA, USA, 2–4 April 2021; Mickens, J., Teixeira, R., Eds.; USENIX Association: Berkeley, CA, USA, 2021; pp. 687–703.
41. Lamport, L. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: The Works of Leslie Lamport*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 179–196.
42. Liskov, B.; Ladin, R. Highly available distributed services and fault-tolerant distributed garbage collection. In Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, Calgary, AB, Canada, 11–13 August 1986; pp. 29–39.
43. Fidge, C.J. A limitation of vector timestamps for reconstructing distributed computations. *Inf. Process. Lett.* **1998**, *68*, 87–91. [CrossRef]
44. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.* **1993**, *18*, 423–434. [CrossRef]
45. Yabandeh, M.; Knezevic, N.; Kostic, D.; Kuncak, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In Proceedings of the NSDI 2009, Boston, MA, USA, 22–24 April 2009; Volume 9, pp. 229–244.
46. Nelson, L.; Bornholt, J.; Gu, R.; Baumann, A.; Torlak, E.; Wang, X. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, Huntsville, ON, Canada, 27–30 October 2019; pp. 225–242. [CrossRef]
47. Gao, Y.; Wang, D.; Dai, Q.; Dou, W.; Wei, J. Common Data Guided Crash Injection for Cloud Systems. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, Pittsburgh, PA, USA, 25–27 May 2022; pp. 36–40. [CrossRef]
48. Chen, H.; Dou, W.; Wang, D.; Qin, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, Australia, 21–25 September 2021; pp. 536–547. [CrossRef]
49. Lu, J.; Liu, C.; Li, L.; Feng, X.; Tan, F.; Yang, J.; You, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19), Huntsville, ON, Canada, 27–30 October 2019; pp. 114–130. [CrossRef]
50. Leesatapornwongsa, T.; Ren, X.; Nath, S. FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–16 November 2022; pp. 1509–1520. [CrossRef]
51. Sun, X.; Luo, W.; Gu, J.T.; Ganesan, A.; Alagappan, R.; Gasch, M.; Suresh, L.; Xu, T. Automatic Reliability Testing For Cluster Management Controllers. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), Carlsbad, CA, USA, 11–13 July 2022; pp. 143–159.
52. Yuan, X.; Yang, J. Effective Concurrency Testing for Distributed Systems. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 16–20 March 2020; pp. 1141–1156. [CrossRef]
53. Golab, W.; Rahman, M.; Auyoung, A.; Keeton, K.; Gupta, I. Client-Centric Benchmarking of Eventual Consistency for Cloud Storage Systems. In Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS), Madrid, Spain, 30 June–3 July 2014.
54. Golab, W.; Li, X.S.; López-Ortiz, A.; Nishimura, N. Computing Weak Consistency in Polynomial Time: [Extended Abstract]. In Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, San Sebastian, Spain, 21–23 July 2015; pp. 395–404. [CrossRef]

55. Golab, W.; Li, X.; Shah, M.A. Analyzing Consistency Properties for Fun and Profit. In Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, San Jose, CA, USA, 6–8 June 2011; pp. 197–206. [CrossRef]
56. Rahman, M.R.; Golab, W.; AuYoung, A.; Keeton, K.; Wylie, J.J. Toward a Principled Framework for Benchmarking Consistency. In Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep), Hollywood, CA, USA, 7 October 2012.
57. Bailis, P.; Venkataraman, S.; Franklin, M.J.; Hellerstein, J.M.; Stoica, I. Probabilistically Bounded Staleness for Practical Partial Quorums. *VLDB Endow.* **2012**, *5*, 776–787. [CrossRef]
58. Bermbach, D.; Tai, S. 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), Lisbon, Portugal, 12 December 2011.
59. Anderson, E.; Li, X.; Shah, M.A.; Tucek, J.; Wylie, J.J. What Consistency Does Your Key-Value Store Actually Provide? In Proceedings of the 6th International Conference on Hot Topics in System Dependability (HotDep), Vancouver, BC, Canada, 3 October 2010.
60. Wada, H.; Fekete, A.; Zhao, L.; Lee, K.; Liu, A. Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: The Consumers' Perspective. In Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, 9–12 January 2011.
61. Bermbach, D.; Sakr, S.; Zhao, L. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In Proceedings of the 5th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2013), Trento, Italy, 26 August 2013.
62. Li, J.; Krohn, M.; Mazières, D.; Shasha, D. Secure Untrusted Data Repository (SUNDR). In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, CA, USA, 6–8 December 2004.
63. Feldman, A.J.; Zeller, W.P.; Freedman, M.J.; Felten, E.W. SPORC: Group Collaboration using Untrusted Cloud Resources. In Proceedings of the The 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, BC, Canada, 4–6 October 2010.
64. Kim, B.H.; Lie, D. Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, Washington, DC, USA, 21 May 2015; pp. 880–896. [CrossRef]
65. Zhou, J.; Xu, M.; Shraer, A.; Namasivayam, B.; Miller, A.; Tschannen, E.; Atherton, S.; Beamon, A.J.; Sears, R.; Leach, J.; et al. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In Proceedings of the 2021 International Conference on Management of Data, Online, 20–25 June 2021; pp. 2653–2666. [CrossRef]
66. Xia, Y.; Yu, X.; Butrovich, M.; Pavlo, A.; Devadas, S. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022; pp. 1478–1492. [CrossRef]
67. Cui, Z.; Dou, W.; Dai, Q.; Song, J.; Wang, W.; Wei, J.; Ye, D. Differentially Testing Database Transactions for Fun and Profit. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022; Association for Computing Machinery: New York, NY, USA, 2023.
68. Tan, C.; Zhao, C.; Mu, S.; Walfish, M. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), Online, 4–6 November 2020; USENIX Association: Berkley, CA, USA, 2020; pp. 63–80.
69. Gürsoy, G.; Brannon, C.; Gerstein, M. Using Ethereum blockchain to store and query pharmacogenomics data via smart contracts. *BMC Med. Genom.* **2020**, *13*, 1–11. [CrossRef] [PubMed]
70. Fekete, D.L.; Kiss, A. A Survey of Ledger Technology-Based Databases. *Future Internet* **2021**, *13*, 197. [CrossRef]
71. Loghin, D. The Anatomy of Blockchain Database Systems. *IEEE Data Eng. Bull.* **2022**, *45*, 48–58.
72. Sharma, A.; Schuhknecht, F.M.; Agrawal, D.; Dittrich, J. Blurring the Lines between Blockchains and Database Systems: The Case of Hyperledger Fabric. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 105–122. [CrossRef]