# Detecting Consistency Violations in Distributed Storage Systems

by

Beom Heyn Kim

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2020 by Beom Heyn Kim

# Abstract

Detecting Consistency Violations in Distributed Storage Systems

Beom Heyn Kim
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2020

As cloud computing becomes increasingly popular, there is a growing need for replicated distributed storage systems. These distributed storage systems provide various consistency models for client applications. Nonetheless, potential software bugs and security attacks can lead to consistency violations, the contravention of the guarantees promised by consistency models. Consistency violations can cause client applications to make incorrect decisions leading to devastating results. Therefore, it is critically important to detect consistency violations.

In this dissertation, we show it is possible to significantly improve existing detection measures against consistency violations by making contributions in two areas: 1) detecting consistency violations caused by software bugs in distributed storage systems and 2) detecting consistency violations caused by security attacks against cloud services hosting distributed storage systems. Previous solutions related to finding software bugs in distributed systems use systematic state-space exploration, but the explored state-spaces are too generic to directly find specific types of bugs. Existing solutions relevant to detecting consistency violations caused by security attacks require either direct communication between clients or a highly available client device. However, emerging end-user devices often do not meet these requirements because these devices may be sitting behind a firewall that prevents direct communication or running on batteries.

We designed and implemented two novel systems to overcome the limitations of existing solutions: 1) Modulo to detect consistency violations caused by software bugs and 2) Caelus to detect consistency violations caused by security attacks. Modulo takes a targeted approach to state-space exploration. It focuses on exploring various scenarios of diverging replicas to see if replicas successfully converge again regardless of how they have evolved. We found eight bugs, leading to convergence failures, in three well-known open-source distributed storage systems. Caelus detects consistency violations using battery-powered end-user devices that do not have direct communication channels. The proposed solution employs a novel combination of cryptographic, network security and distributed systems techniques to enable near real-time detection on limited end-user devices. Our empirical study demonstrated that it can effectively detect simulated attacks on the Amazon S3 service and significantly reduce battery consumption for end-user devices.

# Acknowledgements

First of all, I would like to thank my supervisor, Professor David Lie. Throughout my graduate studies, his guidance has been a beacon unwaveringly showing me the way even in bad times. Without his dedicated and careful guidance, I could not have made it through. Everything thing I know about research is learned from him and I sincerely appreciate the opportunity to work with him.

Moreover, I would like to thank my supervisory committee members, Professor Ashvin Goel and Professor Eyal de Lara, and my external examiner, Professor Kevin R. B. Butler. They provided invaluable advice and feedback on my work, which significantly improved its clarity and thoroughness.

Furthermore, I want to express my gratitude to Professor Yashar Ganjali and Professor Michael Stumm for their comments very helpful to complete this thesis. Also, Professor Yashar Ganjali kindly helped me find the opportunity that initiated my graduate studies. I would like to additionally thank for that.

Although I cannot mention every name here, I thank colleagues who helped me in many ways. I especially thank everyone who helped me improve the presentation of my work more concisely and clearly.

Finally, I would like to dedicate this thesis to my family. My parents have been indescribably supportive, devoted and patient. At their sacrifice, they gave me an opportunity that they had never been given. Their support was the very reason why I could not give up. Also, my brother and his wife have been a great comfort to me during my graduate studies and fed me well so I could survive.

I would like to acknowledge the funding provided by University of Toronto and the Province of Ontario, Bell Canada, the department of Computer Science and School of Graduate Studies.

# Contents


**1 Introduction** **1**

**2 Background** **4**

2.1 Replicated Distributed Storage Systems . . . . . 4

2.2 Consistency Models, Guarantees and Violations . . . . . 6

**3 Related Work** **11**

3.1 Detecting Convergence Failure Bugs . . . . . 11

3.1.1 Distributed Systems Testing . . . . . 11

3.1.2 Distributed Systems Model-Checking . . . . . 12

3.2 Detecting Consistency Attacks . . . . . 13

3.2.1 Fork Detection . . . . . 13

3.2.2 Consistency Analysis . . . . . 13

3.2.3 Ensuring Data Possession . . . . . 14

**4 Detecting Convergence Failure Bugs** **15**

4.1 Introduction . . . . . 15

4.2 An Example CFB . . . . . 17

4.2.1 ZooKeeper's Background . . . . . 17

4.2.2 ZooKeeper Bug #2832 . . . . . 17

4.3 Divergence Resync Models . . . . . 19

4.3.1 DRM Overview . . . . . 19

4.3.2 DRM Examples . . . . . 22

4.3.3 Discussion . . . . . 22

4.4 Modulo . . . . . 23

4.4.1 Schedule Generator . . . . . 23

4.4.2 Concrete Executor . . . . . 25

4.4.3 DRM Interface . . . . . 26

4.5 Implementation . . . . . 29

4.5.1 ZooKeeper Integration . . . . . 30

4.5.2 MongoDB Integration . . . . . 31

4.5.3 Redis Integration . . . . . 32

4.6 Evaluation . . . . . 33

4.6.1 ZooKeeper Bugs . . . . . 33

4.6.2 MongoDB Bugs . . . 34
4.6.3 Redis Bugs . . . 35
4.6.4 Testing Performance . . . 37
4.7 Discussion . . . 38

**5 Detecting Consistency Attacks** **39**
5.1 Introduction . . . 39
5.2 Motivating Scenarios . . . 41
5.3 Security Model and Guarantees . . . 42
5.3.1 Security model . . . 42
5.3.2 Security guarantees . . . 44
5.4 Design . . . 45
5.4.1 System overview . . . 45
5.4.2 Basic system . . . 47
5.4.3 Battery-friendly system . . . 59
5.4.4 Handling failures . . . 60
5.4.5 Operating parameters . . . 61
5.5 Security Analysis . . . 62
5.5.1 Analysis of guarantees . . . 62
5.5.2 Colluding clients . . . 63
5.6 Implementation . . . 63
5.6.1 Cloud service . . . 63
5.6.2 Clients . . . 64
5.7 Evaluation . . . 64
5.7.1 Detecting consistency violations . . . 65
5.7.2 Client verification costs . . . 65
5.7.3 Phone battery consumption . . . 67
5.7.4 Network cost . . . 68
5.8 Discussion . . . 69

**6 Conclusion** **72**
6.1 Future Work . . . 73

**Bibliography** **73**

# List of Tables


2.1 Asynchronous Replication and WAL in Distributed Storage Systems . . . . . 5

4.1 DRM State Description . . . . . 20
4.2 DRM Transition Description . . . . . 21
4.3 Schedule Generation Interface to Implement . . . . . 27
4.4 Concrete Execution Interface to Implement . . . . . 28
4.5 CFB Root Cause Analysis . . . . . 33
4.6 Performance Measurements to Find Bugs . . . . . 37

5.1 Threats and Countermeasures Enabled by Caelus Operation Request and Response Formats . . . . 50
5.2 Consistency verification performance on a PC. . . . . 66
5.3 Consistency verification performance on a smartphone. . . . . 66
5.4 Battery savings and percentage time sleeping comparison between when the phone acts as an attestor and when attestor-partitioning is used. . . . . 67
5.5 Battery drain and average CPU frequency of an active phone with and without the attestor role. . . 68
5.6 Network bandwidth consumed by Caelus operations. . . . . 68

# List of Figures


2.1 Asynchronous Replication Illustration . . . 6
2.2 Synchronous Replication Illustration . . . 7
2.3 Divergence and Convergence . . . 8
2.4 Illustrative Consistency Violation Example 1 . . . 9
2.5 Illustrative Consistency Violation Example 2 . . . 10
2.6 Illustrative Consistency Violation Example 3 . . . 10

4.1 ZooKeeper Bug #2832 Illustration . . . 18
4.2 Modulo Architecture . . . 24
4.3 State Exploration during Schedule Generation . . . 25
4.4 DRM Interface Architecture . . . 26

5.1 Caelus Architecture. . . . 45
5.2 The Caelus Protocol . . . 48
5.3 Operation Request Message and Response Message Formats . . . 49
5.4 Operation Path from a Client to the History Server . . . 51
5.5 Attestation Procedure . . . 52
5.6 Networking and Processing Delay for Reading a New Attestation . . . 53
5.7 Caelus Consistency Verification Procedure Overview . . . 55
5.8 Verification of Eventual Consistency . . . 56
5.9 Verification of Causal Consistency . . . 57
5.10 Percentage of `Gets` with Consistency Violations on S3 as a Function of $T_S$. . . . 65
5.11 Percentage of `Gets` with Consistency Violations on S3 as a Function of $T_A$. . . . 66
5.12 Operation Reordering due to Benign Delay May Lead to False Positive . . . 70
5.13 Waiting for the Upper Bound for the Delay and Requiring the Distributed Key-Value Store to Honour Client Timestamps . . . 71

# Chapter 1

# Introduction

Distributed storage systems are becoming increasingly critical components in the age of information technology. Cloud services rely on distributed storage systems; for instance, Cassandra powers Netflix, Instagram, Facebook and Reddit, while MongoDB is running behind eBay, SurveyMonkey, Sony and Twitter [74]. According to Juniper Research, there were 2.4 billion cloud service users in 2013 and this number reached 3.6 billion in 2018 [86], demonstrating its rapid growth. This growth is ongoing, as a report by Gartner predicts that the public cloud computing market will reach 411 billion dollars by 2022 [37]. Therefore, the importance of distributed storage systems will proportionally increase. Also, many cloud providers are providing distributed storage systems as cloud storage services, such as Amazon S3, Microsoft Azure Storage, and Google Cloud Storage. According to Research and Markets, the growth of the cloud storage service market alone is expected to reach 92 billion dollars by 2022 [85]. Therefore, distributed storage systems are becoming irreplaceable and ubiquitous in modern-day computing services.

For reliability, durability, scalability, availability and short latency, modern distributed storage systems, such as ZooKeeper, Cassandra, MongoDB and Redis, replicate data across distributed servers or nodes. Naturally, those *replicated distributed storage systems* (hereinafter "distributed storage systems") must handle concurrent operations from geographically distributed clients. Each client usually makes requests to the closest replica in terms of the physical distance or the round-trip delay over the network. Updating one replica does not instantly replicate the same operation in other replicas because networking and processing delays are inevitably involved, and spontaneously occurring failures may block replications. Hence, any write operation to a replica will make it different from others; therefore, replicas become *diverged*. This divergence will last until a replication mechanism completes replicating operations to remote replicas. Once replication is complete, diverged replicas become equivalent or *converged*. Here, questions related to data consistency arise. If a client attempts to read before the replication of an operation is complete, should the system show the data before applying the operation or after? If the system lets the client read stale data, how stale can that data be? If clients make more than one write operation, which write should the system commit first, and which should come next? Answers to these questions require a *consistency model*, which is the contract between clients and systems that governs the timing and order of operations.

Currently, various consistency models exist and provide a spectrum of *consistency guarantees* [97]. For example, eventual consistency guarantees that every replica will eventually converge, and strong consistency, also known as linearizability, guarantees that every read operation sees the effect of all previous write operations in a single global order. However, implementation bugs and security attacks can lead to the breach of consistency

guarantees. For instance, if eventual consistency's guarantee is infringed upon by software bugs in a synchronization mechanism, the affected replicas will not see some operations, and convergence between them will fail. If strong consistency's guarantee is violated by an adversary that omits the latest write for some replicas, the affected replicas will send clients stale data. We define such an infraction of consistency guarantees as a *consistency violation*. Consistency violations can have devastating effects on client applications, leading them to make incorrect decisions. For example, an application may rely on a distributed storage system to store their access control policies and related data. A data owner may attempt to revoke a user's permission to access confidential data via the application. Either implementation bugs in the distributed storage system or security attacks mounted against the system may omit the operation revoking permission. Thus, some nodes of the distributed storage system may not apply the revocation operation. Then, the application may make an access control decision based on data from the node that did not apply the revocation operation. Consequently, the application may mistakenly allow the prohibited user to access confidential data. Therefore, detecting consistency violations is necessary and critical for applications relying on distributed storage systems.

In this dissertation, we show that it is possible to significantly improve existing detection measures against consistency violations in replicated distributed storage systems. We make two novel contributions: (1) a systematic testing tool for directly detecting the specific types of bugs leading to convergence failures and (2) a distributed system suitable for end-user devices to promptly detect consistency violations caused by security attacks. First, we develop Modulo, a systematic testing tool that can more directly find software bugs in distributed storage systems. Because modern distributed systems are extremely complex, it is almost inevitable that software bugs will be present in implementations. Software testing methodology can contribute to the verification of the implementation correctness by finding software bugs. However, manual or random testing methods are not systematic. Thus, state-space exploration is often used by model-checkers to systematically test distributed systems [105, 47, 44, 66, 51, 62]. Nevertheless, previous proposals did not use targeted state-space exploration to find a specific class of bugs, so the state-spaces they searched were generic and very large. Being generic allows various classes of software flaws to be more easily found, but it is inefficient for finding a specific class of bugs. Considering that asynchrony is allowed in distributed storage systems more often these days, there are more concurrent operations than ever. Also, as the scale of distributed storage systems increases, failures will be more frequent and prevalent than ever. These trends all increase the size of the state-space to search; therefore, generic approaches for finding specific types of bugs will become increasingly difficult.

Instead of being generic, we focus on finding *convergence failure bugs* (CFBs), which are software bugs that lead to the failure of convergence between replicas. More specifically, we focus on finding CFBs that can manifest only by injecting a sequence of external events. Thus, unlike previous proposals, we do not interleave low-level operations, such as acquiring or releasing the lock, I/O operations or sending or receiving network messages, during the state-space exploration. Our approach focuses on the divergence and convergence of distributed storage systems. We further focus on interleaving specific external events: write operations, crash or network failures, and restart or reconnection. As a result, our state-space exploration is more directly targeted at specific CFBs we want to find. We applied our targeted state-space exploration technique to test three well-known open-source distributed storage systems: ZooKeeper, MongoDB and Redis. Accordingly, we found CFBs in all of them.

Second, we develop Caelus, a distributed system that can promptly detect security attacks that cause consistency violations with end-user devices. Because cloud services offering distributed storage systems are difficult to make free of vulnerabilities, there is always a risk that adversaries can compromise a cloud service. Adversaries that successfully compromise a cloud service hosting a distributed storage system may omit, reorder, replay, delay, truncate, tamper or forge client operations. In doing so, they can intentionally cause consistency violations that

are very subtle and hard to detect. Detecting consistency violations requires trusted clients to share their *views*, which are the history of operations perceived by clients. For instance, one client makes a write operation, and another client does not see it because of omission attempts by the compromised cloud service. This attack can be detected only by having the writer and reader clients exchange what each of them did and saw. If the writer tells the reader it has written an operation, then the reader client will detect the omission attack. It is now more challenging to do this because end-user devices, such as laptops, tablets and smartphones, have two distinct limitations. First, end-user devices may be sitting behind a gateway without additional configurations for Firewall/NAT Traversal, so there may be no direct communication between them across the wide-area network. Second, most end-user devices today run on batteries, so power is often scarce. Previous proposals have detected consistency attacks by relying on direct communication between clients or a trusted device that is highly available for clients to exchange views and perform checking frequently. Thus, previous proposals are not suitable for working with end-user devices.

We enable a near real-time detection that can work with trusted end-user devices that have the limitations mentioned above. Our proposal has an untrusted cloud that is hosting a distributed storage system declare a view for any client. Also, an arbitrary client can use the cloud to distribute the declared view to every other client. If a distributed view from one client does not match views declared for other clients or does not conform to the consistency model promised by the cloud service, then the clients detect consistency violations. Because the cloud service is highly available and provides a view for any client, we do not need a separate highly available device getting and distributing its view—any client device can do it. By combining with cryptography and time-bound enforcement, we can create a secure communication channel over the untrusted cloud service that has bounded delay to distribute the coherent declared view to every client. Accordingly, direct communication between clients is not required to distribute a view to compare with securely. Since we do not require clients to be up and running solely for our mechanism, our proposal is also battery friendly. In addition, clients get the history of operations, and they can perform checks to detect miscellaneous attacks on their operations for various consistency models. We evaluated our proposal using Amazon S3 and an Android Nexus smartphone. As a result, we could detect every simulated consistency violation and dramatically reduce the battery-power consumption of client devices.

Both Modulo and Caelus target replicated distributed storage systems in adversarial environments, but their scopes are distinctively different. Modulo assumes that the system can operate in a situation where nodes can fail, but when failures occur, nodes do not behave arbitrarily. Thus, it does not take Byzantine failures into account. On the other hand, Caelus assumes the cloud service providing distributed storage systems as a service may get compromised by adversaries and show a Byzantine behavior in a way violating consistency violations.

We begin by presenting the background information about distributed storage systems and consistency models, guarantees and violations in Chapter 2. Then, we discuss related techniques and previous proposals in Chapter 3. We present our systematic testing tool for finding bugs leading to convergence failures in Chapter 4. Also, we describe our distributed system for end-user devices to detect consistency violations caused by security attacks in Chapter 5. Last, we conclude in Chapter 6.

# Chapter 2

# Background

## 2.1 Replicated Distributed Storage Systems

Many distributed storage systems keep datasets on a persistent storage device, such as a hard drive or a solid-state drive. Although systems usually retrieve data from a volatile type of memory like RAM for greater speed, such data can be lost when failures occur and machines crash. Hence, to prevent data loss, those systems employ a mechanism to restore data from their persistent storage device. *Write-ahead logging* (WAL) is a well-known technique that logs every operation in an *append-only log* (AOL) on persistent storage before applying the operation to data cached in memory [76]. All the distributed storage systems summarized in Table 2.1 employ WAL. Cassandra, HBase, MongoDB, Redis and ZooKeeper have a built-in WAL component, while Druid, MemcacheDB and Voldemort rely on the WAL of another system, such as BerkeleyDB or Kafka. Couchbase and Riak can be configured to log operations on their persistent storage device before applying them to data in the memory, though the default setting is the opposite.

Distributed storage systems usually replicate a dataset across multiple nodes. Many systems allow a node to operate as an independent storage server. However, the node storing the dataset may fail for various reasons, such as hardware failure, OS crash, fire, flood or network cable disconnection, to name a few. Once the node fails, clients can no longer access the dataset stored on it. By replicating the dataset on more than one node, clients can still access their data even if a node fails. Also, replication helps provide better scalability for client applications because it inherently distributes the load across replicas. Shorter latency is another benefit of replication, since a client may access the replica closest to it. Thus, to provide a reliable, highly available, scalable and quickly responding storage server, modern distributed storage systems operate using replication. There are two types of replication: *asynchronous replication* and *synchronous replication*. Figure 2.1 illustrates asynchronous replication. Asynchronous replication commits or applies operations on one node first and then asynchronously propagates operations to other nodes. The synchronous replication depicted in Figure 2.2 commits or applies operations only after sending operations and receiving consent from a *quorum* consisting of the majority of nodes. Compared to synchronous replication, asynchronous replication offers higher availability and better scalability because it can ingest operations without coordination and without a quorum. Today, the best-known distributed storage systems provide asynchronous replication, as is summarized in Table 2.1.

*Divergence* occurs as operations are asynchronously propagated to distributed replicas. In distributed storage systems, replicas inevitably commit operations at different times due to various networking and processing delays involved in replication. Divergence can be defined as the degree of difference in the replicated dataset of nodes

| System Name | Async. Replication | WAL |
|---|---|---|
| Cassandra | ✓ | commit log |
| HBase | ✓ | WAL or<br>HLog (prior to 2.0) |
| MongoDB | ✓ | journal |
| Redis (Cluster) | ✓ | Append-Only File (AOF) |
| ZooKeeper | ✓ | transaction log |
| Druid | ✓ | Message Bus (Kafka) |
| MemcacheDB | ✓ | BerkeleyDB WAL |
| Voldemort | ✓ | BerkeleyDB WAL |
| Couchbase | ✓ | Couchstore ($persistTo = 1$) |
| Riak | ✓ | Bitcask ($o_sync$ enabled) |

Table 2.1: Asynchronous Replication and WAL in Distributed Storage Systems

due to the disparity in writes committed on nodes. The degree of divergence is usually limited, as systems aim to replicate and commit operations on every replica quickly. However, failures may block replication so that the degree of divergence may increase, and such divergence will last until failure recovery occurs. A simple example illustrating this concept is given in Figure 2.3 on the left-hand side. At time 1, a client sends a request to node A for an operation to write 0 to the data object X, $W(X, 0)$. At time 2, node A commits the operation, and at time 3, node A replicates the operation to nodes B and C. Thus, there is a small degree of divergence between time 2 and time 3. Between time 4 and time 7, node B fails, and a client sends another request for an operation to write 1 to the data object X, $W(X, 1)$. Node A can replicate the operation to C but cannot replicate it to node B due to the failure. Here, divergence will last until the failure of node B is recovered. The example shows the degree of divergence increase between time 8 and time 10, as node C also fails, and another operation, $W(X, 2)$, is committed on node A.

On the other hand, *convergence* usually occurs after replication is complete and every replica has committed replicated operations. Convergence can be defined as the inverse of divergence, the decrease of divergence in the system. However, if convergence did not occur due to replication being blocked by failures, the failures must be recovered first. To recover from failures, the recovery process of distributed storage systems must restore the consistency of a node's local dataset (i.e., *internal consistency*) and must eliminate any differences among replicas to make them converge again (i.e., *mutual consistency*) [93]. Generally, distributed storage systems use WAL to restore the internal consistency of each replica. Then, the resync procedure should take place to regain mutual consistency. During the resync procedure, nodes should discard invalid conflicting operations via *truncation* and replicate missing operations. To do so, each resync participant exchanges the *timestamp* of the last operation it committed to determine what operations to truncate or replicate. Figure 2.3 gives an example of convergence on the right-hand side. Node B initially failed at time 1 and rejoins at time 2. At this point, the *resync* procedure attempts to get node B in sync with other nodes. In the example, node A replicates missing operations to node B. At time 3, replicas become converged, as every replica commits the same operations and, therefore, there is no divergence remaining in the system.

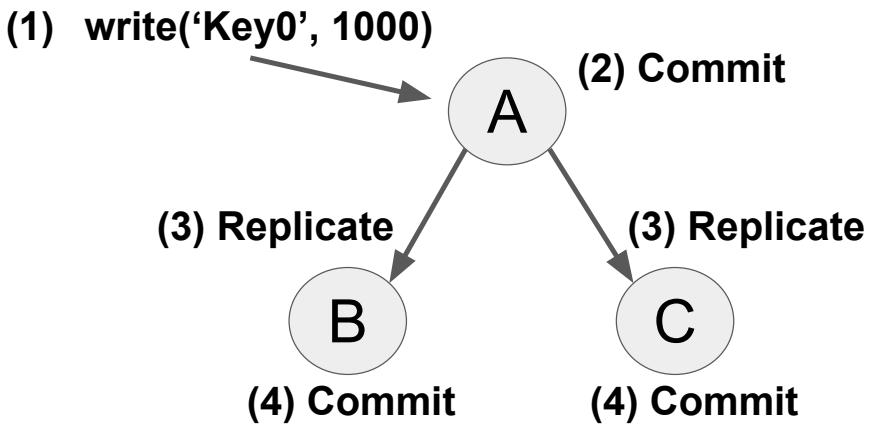


Figure 2.1: Asynchronous Replication Illustration. (1) a client makes a request for a write operation and node A receives the request, (2) node A commits the operation right away, (3) node A replicates the operation to other nodes and (4) nodes B and C eventually commit the operation.

## 2.2 Consistency Models, Guarantees and Violations

Modern distributed storage systems offer various consistency models. Traditionally, strong consistency is the only model that is widely accepted. [1] It guarantees that each read operation sees the effect of all previous write operations in a single global order. Hence, it is the easiest one to reason about and program with. Nevertheless, because most implementations require the strict ordering of operations through coordination among replicas, it is not only the least scalable but also the least tolerant to failures. Whenever a failure occurs, there is a good chance that clients will not be able to use the systems. For instance, if a network partition occurs, clients connected with a replica isolated from other replicas will not be able to access their data.

As scalability and high availability have become more favourable in large-scale applications that can work without strong consistency, distributed storage systems have started to provide relaxed consistency models [35, 8]. For instance, eventual consistency became a popular consistency model for applications, such as Amazon's shopping cart [100] and Netflix's data warehouse [58]. Eventual consistency only guarantees that each read sees any subset of previous writes, and it does not require any coordination up front. Thus, systems providing eventual consistency can ingest client operations even when the clients access the replica without support from a quorum. Although this is the most scalable and highly available model, it provides no real safety property—every read operation will be valid as long as the resulting data are not corrupted. So, the application domains that can use eventual consistency are restricted—applications that use eventually consistency must be specially designed to handle the uncertainty about the freshness of the data.

Many large-scale applications can give up strong consistency but require more guarantees than eventual consistency. For example, social networking applications need messages posted by clients to appear in an order that maintains causality [9]. If a client sees one message and replies, the reply should not come before the message. Causal consistency is considered the best consistency model for such applications [67, 68]. Because causal consistency ensures that every read operation sees write operations in a causal order, clients never see the reply

[1] Strong consistency is also known as one-copy serializability or linearizability.

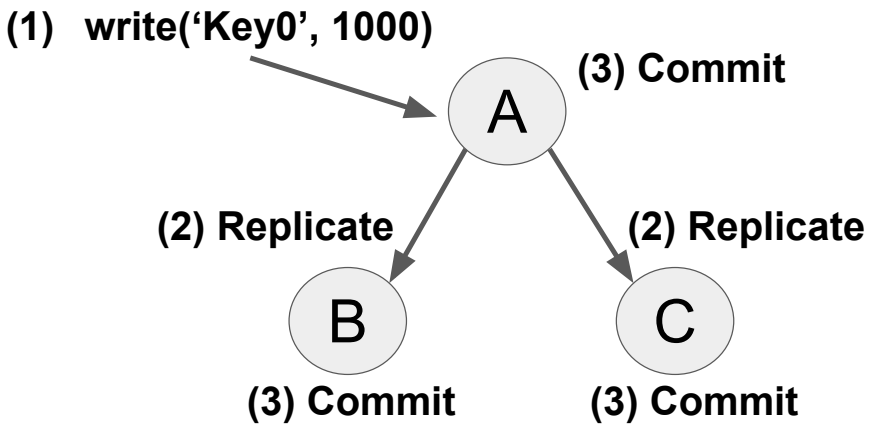


Figure 2.2: Synchronous Replication Illustration. (1) a client makes a request to distributed storage systems composed of three nodes: A, B and C by sending a write operation, *write('Key0', 1000)*, to node A, (2) node A does not commit the operation right away and instead replicates to other nodes B and C and (3) finally, nodes A, B and C commit the operation.

before they see the message. Similar to the social networking application example, different application domains may require different intermediate-level consistency guarantees that optimally meet their goals. Thus, various intermediate consistency models have been developed, and Terry compiled a set of practically useful and cleanly describable intermediate-level models in his article [97].

Although various consistency models provide a broad spectrum of guarantees, software bugs and security attacks can break these guarantees. As discussed earlier, distributed storage systems are complex, thus software bugs are almost always present in their implementations. Specifically, software bugs residing in resync mechanisms can cause violations of the convergence guarantee, which is the most fundamental consistency guarantee provided by virtually every existing consistency model in practice. Under normal circumstances, resync mechanisms should eliminate divergence by removing invalid conflicting operations and replicating missing operations. However, after going through a specific divergence scenario, a convergence failure bug (CFB) may manifest, and replicas cannot converge, even if resync mechanisms are complete, leaving the system in a state where replicas remain diverged permanently. Moreover, even if distributed storage systems do not contain such bugs, distributed storage systems and their host cloud services may be vulnerable to security attacks. Adversaries who successfully compromise those systems and services can mount consistency attacks to cause consistency violations [2].

Here, we explain how consistency violations can occur and affect client applications through a few illustrative examples. Suppose we have three friends who want to determine when to hold a meeting at a prearranged location. They are communicating through a chatting application, and clients of the application use a distributed storage system hosted by a cloud service to exchange messages. Clients make read operations to retrieve previous messages and write operations to append new messages. The underlying distributed storage system promises strong consistency, so read operations should see the effect of all previous write operations in a single global order. Despite this promise, the distributed storage system may have a bug that can manifest in a particular failure

[2]Consistency attacks include omitting, reordering, replaying, delaying and truncating operations; the forging and tampering of operations and data; and forking attacks [63].

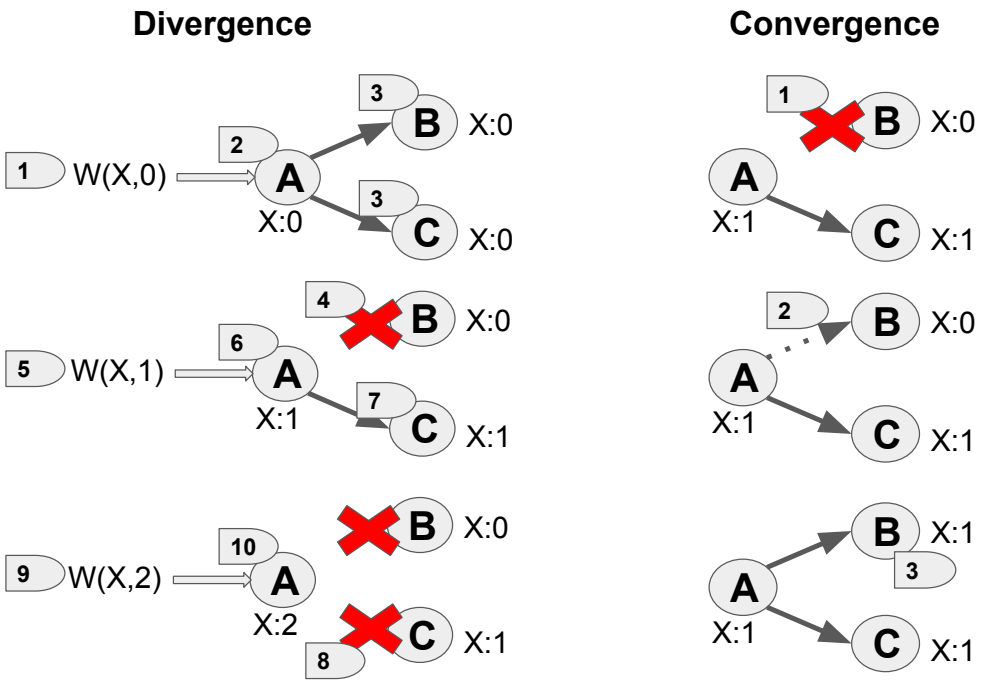


Figure 2.3: Divergence and Convergence.

recovery scenario, or it may be under threat from malicious attackers. Once the distributed storage system falls into a situation where the system cannot continuously provide strong consistency, clients may differently perceive the history of operations—we define a history of operations perceived by a client as a *view*.

Figure 2.4 shows an example in which clients perceive different views due to the omission or delay of a client's message. Note that the underlying distributed storage system is supposed to provide strong consistency and that is what the clients assume. Under strong consistency, every client must perceive an identical view. Yet, a problem arises if a bug or an attack causes the distributed storage system to provide a weaker level of consistency by omitting or delaying a client's message. Initially, client 1 makes a write operation to send a message proposing to hold a meeting at 5 PM, but they figure out that there is a conflict in the schedule. Client 1 tries to reschedule the meeting from 5 PM to 7 PM by making the subsequent write operation. Client 2 reads what other clients have proposed so far and sees two messages from client 1 in the correct order. Then, client 2 agrees to meet at 7 PM and replies with a message saying "OK." When client 3 tries to read other client messages, the latest message from client 1 is omitted or delayed. Because client 3 does not see the effect of the latest write operation of client 1, it is a violation of strong consistency; therefore, a consistency violation occurs due to the omission or delay, and this affects the view of client 3. From the perspective of client 3, it seems that client 1 is proposing to meet at 5 PM, and client 2 is agreeing with client 1's proposal. Without any additional precaution such as asking for confirmation from other clients, client 3 will show up at the meeting at 5 PM and wait for two hours unexpectedly. The client operations also can be reordered and replayed. Figure 2.5 shows that a consistency violation can occur when client 1's messages are reordered. Although client 3's read operation sees all previous write operations, client 1's messages are reordered only for client 3, which contravenes the rule of a single global order that strong consistency must guarantee. Figure 2.6 shows that a consistency violation can occur when client 1's message is replayed. Because client 1's message was repeated only for client 3, this example also infringes on the rule of a single global order. All three examples here make client 3 think the meeting is at 5 PM.

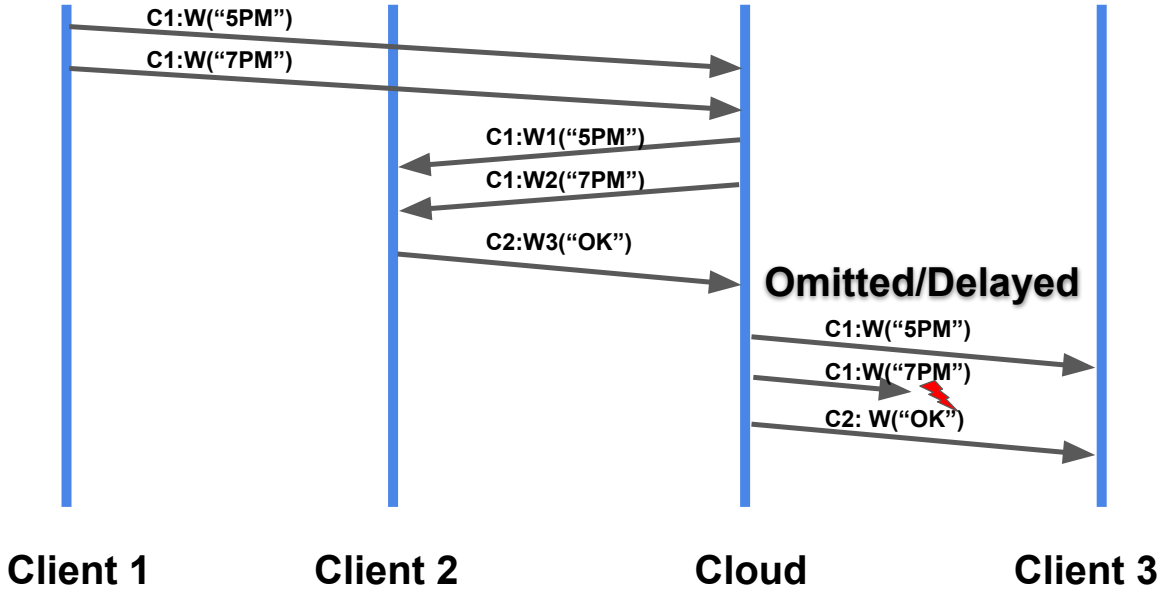


Figure 2.4: Illustrative Consistency Violation Example 1. Delayed Messages or Omitted Messages

As these examples illustrate, consistency violations directly affect how clients perceive the history of operations. Because the perceived views are different for clients in a way that violates the promised guarantees, the data they make decisions based on are not precise. Accordingly, consistency violations can cause applications of distributed storage systems to make incorrect decisions due to imprecise data, which can have devastating results. For example, doctors who are exchanging diagnostic data and opinions may misjudge the treatments they should implement to save their patients' lives. Also, financial software that is exchanging information can mistakenly sell or buy large amounts of stock. Self-driving cars, while sharing important traffic information with other cars or traffic control systems, may go in a misconceived direction and collide.

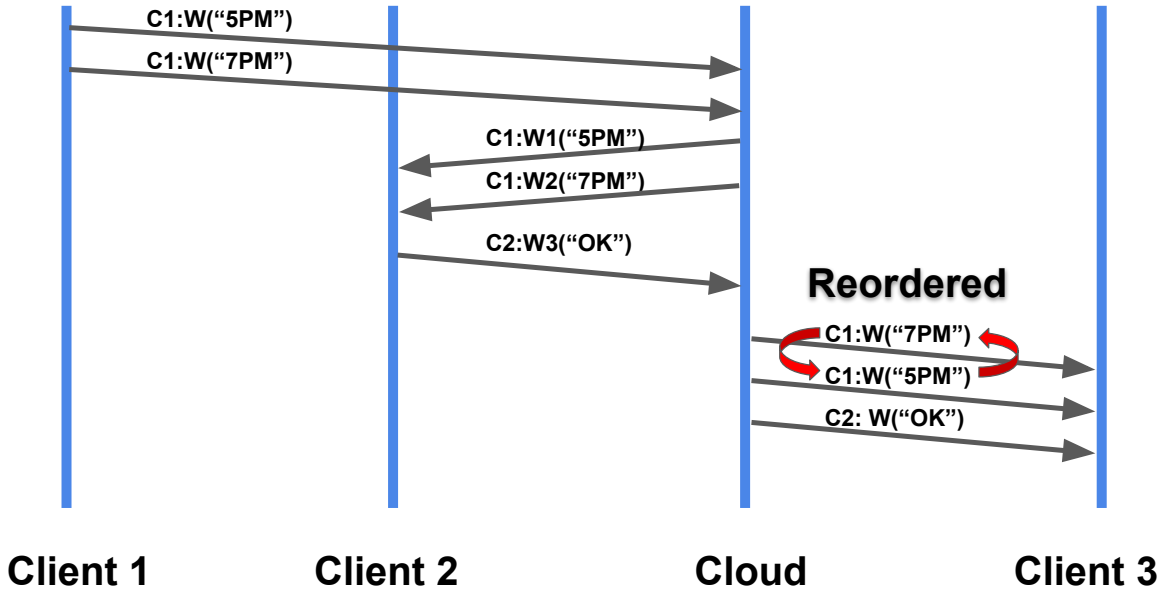


Figure 2.5: Illustrative Consistency Violation Example 2. Reordered Messages

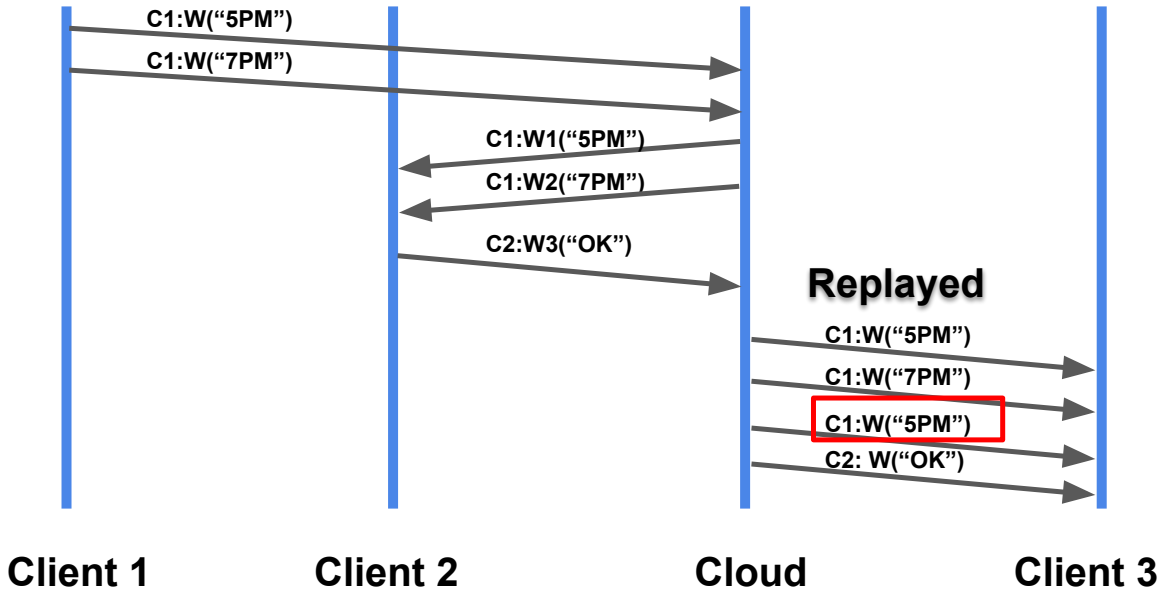


Figure 2.6: Illustrative Consistency Violation Example 3. Replayed Messages

# Chapter 3

# Related Work

In this chapter, we discuss related works to which Modulo and Caelus are relevant. First, we start with presenting works related to Modulo, which is a tool to help developers find convergence failure bugs, which are a specific type of consistency bug [46]. Second, we discuss works related to Caelus, which is a distributed system running on a small set of end-user devices for monitoring untrusted services hosting distributed storage systems.

## 3.1 Detecting Convergence Failure Bugs

Because Modulo's goal is to find bugs in distributed storage systems, we present various advanced techniques that help developers find flaws in complex distributed systems. First, we discuss related works for automated software testing. Modulo is different from previous proposals because it systematically tests the divergence and convergence of replicated distributed storage systems only by externally injecting test inputs, such as API calls, failures and environmental events. Second, we present existing techniques for model-checking. Although model-checking is a formal verification method, it is a widely used tool for finding bugs in complex systems [61]. Modulo employs state-space exploration on which model-checking relies, thus we compare previously proposed model-checkers with Modulo.

### 3.1.1 Distributed Systems Testing

Some previous proposals for testing distributed systems employ state-space exploration. However, their state-spaces are more focused on interleaving concurrent internal events, such as thread scheduling or network message delivery [45, 29]. Also, a previous work presents a tool for injecting network-partitioning failures for cloud systems [11]. Yet, it does not inject crash failures, and it requires an OpenFlow-capable hardware component to simulate network-partitioning. Jepsen is an open-source testing tool that injects various types of failures into distributed database systems [57]. However, Jepsen randomly generates inputs, which implies that the state-space for input sequences cannot be efficiently reduced without missing corner cases and reproducing bugs may not be possible due to the non-deterministic ordering of events that Jepsen does not record and cannot reproduce. One work focuses on interleaving low-level file system operations across distributed nodes along with crash failures [10], but it is limited to crash injection only and does not test the divergence and convergence behaviors of distributed systems.

Model-based testing systematically derives test cases from an abstract model of the system-under-test [80, 56, 95, 87, 21, 78]. Dalal et al. devised a method that can generate various input parameter values for tests from an

abstract model called the Test Data Model [28]. A technique that can derive test cases for system testing from an UML statechart was developed by Offutt et al. [81]. Also, Gargantini et al. propose to use model-checking to derive test cases from an abstract model [36]. In addition, Andrews et al. came up with a technique that models a web application as a finite state machine to generate tests [13]. Also, Yang et al. applied model-based testing to find security flaws in about 500 implementations of OAuth 2.0 [107]. However, previous proposals for model-based testing do not look for convergence failure bugs. Also, none of the existing model-based testing works test the divergence and convergence of replicated distributed storage systems as the system-under-test.

### 3.1.2 Distributed Systems Model-Checking

Model-checking has been extensively studied and used to prove the correctness of complex systems and to find bugs in them. Clarke and Emerson were the first to propose model-checking, which exhaustively explores the state-space of abstract models specified in temporal logic [24]. Dill developed a model-checker called Murphi, which is used to prove the correctness of various systems, including distributed shared memory systems [30]. SPIN is another popular abstract model-checking tool [48]. More recently, Lamport developed TLA+, a specification language, and TLC, a model-checker for TLA+, which has been used by Amazon [61], showing the practicality of model-checking in the industry. Nevertheless, abstract model-checking cannot find bugs in implementations directly.

Concrete model-checking is used to employ an implementation as a model to explore directly. Musuvathi et al. proposed a concrete model-checker using implementations as the model to verify [77]. Godefroid also proposed the same idea around the same time [41]. Killian et al. devised a technique that enables checking for not only safety properties but also liveness properties [51]. Lin et al. developed a black-box concrete model-checker that does not need to know about the source code by interposing events at the interface layer between the target system and the underlying operating system [66]. Model-checking not only proves the correctness of the system but also predicts if the implementation execution is driving the system to faulty states and steers away the system execution to prevent that [105]. Simsa et al. explored the generalization of concrete model-checking to provide flexibility for determining the level of non-determinism controls [91]. Recently, concrete-model checkers have been improved to detect deep bugs by exploring scenarios involving multiple failures [62, 70]. However, concrete model-checking is much slower than abstract model-checking because it involves execution delays. Moreover, a state-space to explore is usually not targeted to find a specific type of bug but is instead general to find various concurrency bugs because previous proposals explore the interleaving of both internal and external events while focusing more on internal ones.

Researchers have explored various techniques for reducing the state-space to explore. Symmetry and partial-order reduction techniques were proposed and studied [23, 99, 34]. Symmetry reduction allows model-checkers to not have to repeatedly revisit symmetric states that are defined as states that can be considered equivalent for verification purposes. For instance, suppose the state of a program consists of $X = 1$ and $Y = 2$. Also, assume that we want to compute $Z = X + Y$ to check whether the sum of two variables does not exceed a threshold value of 5. Then, the state consisting of $X = 2$ and $Y = 1$ is symmetric to the aforementioned state, and computing $Z = X + Y$ will lead to the same result. Hence, we can skip checking the second state and reduce the state-space to explore. In addition, dependency between operations may be utilized via partial-order reduction because transitions that are independent from each other may lead to the same state regardless the interleaving of those independent operations. Also, there have been proposals to reduce the state-space by decomposing local state transitions from the global state transition [47, 44]. Additionally, some recent proposals explored semantic knowledge of implementations to exploit symmetry and dependency between events further to reduce the state-

space [62, 70]. Though various reduction techniques can help model-checking to mitigate the state-explosion problem, these techniques do not help model-checkers find specific types of bugs more effectively. Modulo is specialized for exploring the targeted state-space with a focus on looking for CFBs.

## 3.2 Detecting Consistency Attacks

In this section, we discuss Caelus-related works that can detect attacks that cause consistency violations. Because Caelus' foremost goal is to detect violations caused by malicious services hosting replicated distributed storage systems, we start with discussing fork detection works that allow clients to monitor untrusted storage servers. However, previous proposals on fork detection are not general enough, so these proposed approaches cannot support multiple consistency models on a single framework. Therefore, we present related works that can identify violations of various consistency models. Also, there have been several works on ensuring possession and retrievability of files stored on untrusted cloud storage services. We will compare these works with Caelus.

### 3.2.1 Fork Detection

A fork occurs when there is more than one partition of clients that have different views of the history of operations than others. Fork detection requires the exchange of each client's view and comparison to see if there are any discrepancies among the different views. Li et al. studied fork consistency for the first time and proposed a highly available "timestamp box" to exchange views with every client as the fork detection method. Alternatively, Popa et al. proposed a centralized auditing service that collects and analyzes all views instead of the highly available runtime monitor [83]. As the third option, several other proposals explored the approach using distributed clients exchanging views directly [72, 33, 90]. Nevertheless, previous proposals do not consider end-user devices' restrictions, such as limited battery power and unreliable network connectivity. Also, they do not provide verification mechanisms for detecting violations of various consistency models on a single framework, as we mentioned earlier.

### 3.2.2 Consistency Analysis

By inspecting the history of client operations, consistency analysis can tell if distributed storage systems have been providing promised consistency guarantees. Several previous works developed a graph algorithm approach to analyze the history of client operations [12, 43]. For example, a precedence graph, a directed graph, can be built from the information contained in the history, and a depth-first search strategy can be applied to traverse the graph to look for any cycle in it. If there is a cycle, then it means there is a violation of atomicity. Other approaches use timestamps to compute the time difference between each operation to tell how stale each read is [104, 18]. Determining if the given history conforms to strong consistency is NP-complete in the general case where writes are not distinguishable, as they are writing the same values [40]. Yet, researchers found that encoding unique values (e.g., timestamps or monotonically increasing versions) into the data value will help distinguish each write and analyze the given history in the polynomial time [43]. Although previous proposals provide various algorithms for consistency analysis to detect if a violation of some arbitrary consistency model occurs, these previous solutions do not target adversarial environments. Additionally, previous proposals do not consider specific constraints of the end-user devices aforementioned. Caelus can detect violations of various consistency models in adversarial environments with end-user devices.

### 3.2.3 Ensuring Data Possession

Several previous proposals present various techniques to ensure the intactness of files stored on the untrusted cloud storage services. Some related works propose to distribute data across multiple cloud services to protect the integrity and recoverability of data [103, 19, 20]. Some other related works present novel cryptographic techniques to probabilistically prove retrievability [50, 22] or data possession [14]. However, these previous proposals do not address the consistency of data and mostly deal with static data. On the other hand, Caelus does not directly address recoverability or retrievability. Thus, earlier proposals for ensuring file intactness are orthogonal but complementary to Caelus.

# Chapter 4

# Detecting Convergence Failure Bugs

## 4.1 Introduction

Various distributed storage systems [60, 1, 6, 88, 49, 106, 5, 96, 2, 7] support asynchronous replication [101] for high availability and scalability, even in the presence of failures. With asynchronous replication, ingesting client requests does not require any coordination up front. Instead, systems replicate the operations requested by clients asynchronously, so clients and server nodes do not have to be blocked for an extended time, even if some of the nodes have failed and become unreachable. Therefore, for large-scale applications, asynchronous replication is often more suitable than synchronous replication.

However, asynchronous replication poses a potential reliability issue regarding the data consistency of distributed storage systems. As discussed in Vogels's paper [101], asynchronous replication can make replicas *diverge* during an *inconsistency window* because replicas cannot commit writes at the same time. It will take some time for replicas to *converge* due to processing and networking delays. Hence, asynchronous replication naturally leads to inconsistent replicas. Moreover, failures can leave divergence unresolved for an extended time, because replication completion will be interrupted and delayed until the recovery process occurs successfully. That is, divergence may be left unresolved permanently if the successful recovery never happens.

Software bugs can be introduced to distributed storage systems by developers' mistakes, so the failure recovery may fail to resolve divergence even after it is complete. In particular, failure recovery may not work correctly due to software bugs in resync implementations. For example, it is possible that the resync procedure may not successfully discard invalid conflicting operations and those operations may remain on only the subset of nodes even after the resync is complete. Also, some stale nodes may fail to replicate missing operations from other nodes. Therefore, divergence among replicas may still be unresolved even after the failure recovery is complete—the duration of the inconsistency window may be unbound until an administrator manually discovers unresolved divergence. Because convergence may never occur without a manual intervention, we call these failures *convergence failures*. Also, we call bugs leading to convergence failures *convergence failure bugs* (CFBs).

We found that we can manifest CFBs by injecting external events, such as write operations, failures and recoveries, into a system-under-test. These bugs are critical to find since the convergence property is the most fundamental consistency guarantee provided by the majority of existing consistency models, if not all. That is, the violation of the convergence guarantee is a sufficient condition to violate virtually every existing consistency model in practice—even including eventual consistency. Also, we found that injecting a sequence of external events relative to externally observable internal events allows us to reliably reproduce these bugs without con-

trolling thread-level or network events to precisely interleave concurrent internal events. Consequently, we can abstract out many unnecessary test scenarios that are different only by the interleaving of internal events and are irrelevant to reproducing CFBs.

There have been various approaches used to find bugs in distributed systems, but none of them have intensively looked for the aforementioned CFBs. Software testing is the most basic approach used to find flaws in distributed systems. Conventional manual testing, where developers manually write test cases, is not practical, if not infeasible, to cover all corner cases. Generative testing tools, such as Jepsen [57], have been developed, and their effectiveness in finding bugs in distributed systems has been demonstrated. Yet, these techniques cannot reliably reproduce bugs and still may miss corner cases. Model-based testing has been explored by researchers [28, 81, 36, 13, 80, 56, 95, 87, 21, 78], and some of them employ the state-space exploration technique to test software implementations systematically. Nevertheless, none of these techniques specifically looked for CFBs in distributed systems. Recently, model-checking, a state-of-the-art formal verification technique that relies on state-space exploration, has been used as a bug-finding tool for distributed systems [77, 51, 66, 105, 91, 47, 44, 62]. However, previous proposals explored various interleavings of concurrent internal events as their focuses are not on specific CFBs but rather on concurrency bugs in general. Accordingly, existing model-checkers explored more generic state-spaces. Thus, their state-space exploration is inefficient and less targeted on finding specific bugs. Unlike previous model-checkers, we develop a more targeted state-space exploration approach in this work.

We present a novel tool that is specialized to find CFBs and allows us to systematically explore various divergence and convergence scenarios in the presence of multiple failures. To abstractly capture divergence and convergence, we created a concept of *Divergence Resync Models* (DRMs). DRMs are incorporated into our prototype system, Modulo, to generate sequences of external events, which are externally injectable test inputs, encoded in *schedules*. Modulo explores the state-space of a DRM that is essentially a finite state machine encapsulating the abstraction of the various divergence and convergence scenarios a distributed storage system (hereinafter "system-under-test", or "SUT") can go through. Then, Modulo drives the SUT by injecting actual test inputs guided by schedules. After finishing with the input injections for each schedule, Modulo waits for every resync to complete. Finally, Modulo performs a verification procedure to check if every replica gets converged successfully. Modulo detects a convergence failure if resync never completes within some pre-configured period or reading data across replicas shows that some of them are still out-of-sync even after resync finishes. Because schedules, results and logs are recorded in files permanently, one can reproduce bugs and perform post-mortem analysis to confirm bug manifestations.

When we ran Modulo on ZooKeeper, MongoDB, and Redis, we found eight total CFBs, including three new CFBs that had not been found before—two new ones from ZooKeeper 3.4.11 and one new one from MongoDB 3.0.0. To find these CFBs, we implemented Modulo prototype written in about 6.4K LOC. In doing so, we constructed four DRM models. Some of those models were reusable across different applications and configurations.

In short, this work makes the following novel contributions:

- Observations on CFBs that can be manifested by interleaving external events only
- Creation of DRMs specifically designed to model divergence and convergence behaviors of SUTs
- Design and implementation of a prototype specialized for finding specific CFGs
- Evaluation through empirical study—testing three well-known open-source distributed storage systems with the proposed method and finding CFBs in all of them

Section 4.2 discusses an example of the CFBs we are trying to find. Section 4.3 presents DRMs that capture the divergence and convergence behavior of distributed storage systems. Then, we describe the design of Modulo in Section 4.4. After that, implementation details are presented in Section 4.5. Then, we share our experience of applying Modulo to test distributed data stores and results in Section 4.6.

## 4.2 An Example CFB

To give an example of the types of bugs Modulo can find, we describe a new bug that Modulo discovered. This example bug was found while we were using Modulo to find bugs in ZooKeeper. Thus, we begin by first providing a quick overview of ZooKeeper. Then, we present the detailed description of the example bug.

### 4.2.1 ZooKeeper's Background

ZooKeeper maintains a leader for each epoch. The quorum is the group of nodes consisting of more than one-half of the nodes in the system and must exist to elect a new leader. ZooKeeper needs a leader to ingest client operations, and every write operation should go through the leader, which serializes the operations. ZooKeeper uses a unique transaction ID for each write operation (called zxid in ZooKeeper) to serialize operations. The lower bits of the transaction ID signify a transaction number, while the upper bits signify the epoch in which the transaction took place. ZooKeeper has two mechanisms that it uses to save data so that it can recover after a crash. First, ZooKeeper has a write-ahead log that it can use to replay operations after recovering from a failure to restore data in memory. Second, ZooKeeper periodically takes a snapshot of the in-memory data and persists them to the disk. This snapshot can then be reloaded into memory after a failure. Taking a snapshot of memory is considerably slower than writing transactions into the write-ahead log as they occur, so ZooKeeper only takes snapshots after the transaction log has grown to a certain point.

These two node-level failure recovery mechanisms also influence the two ways that ZooKeeper resynchronizes a node with its peers after it has restored its local key-value store. Non-leader ZooKeeper nodes try to synchronize data from a leader. This can be done in two ways. One way is a DIFF resync, where the leader transfers all missing operations to the recovering node. The other way is a SNAP resync, where the leader sends its entire key-value store to the recovering node. Also, during resync, it may be necessary for a recovering node to *truncate* its write-ahead log to remove operations that conflict with other operations on the other nodes. To do so, the leader sends a TRUNC message to the recovering node. Truncation is necessary if the recovering node was the only node that committed some operations that are now stale to replicate. For example, suppose the recovering node was initially a leader and committed some transactions locally but could not replicate them to any other nodes before failing. Then, another node might become a new leader and accept subsequent operations, causing a conflict.

### 4.2.2 ZooKeeper Bug #2832

We discovered a previously unknown bug in ZooKeeper and reported it to developers. The identifier for the reported bug is #2832. The steps required to trigger bug #2832 are illustrated in Figure 4.1. The state of the transaction log is given in the grey boxes:▭ and snapshots that are transferred by the SNAP resync are illustrated with the purple ovals:⬭. Initially, we start three nodes A, B and C and wait for a leader to be elected. In this case, node A is initially the leader. Then, $Key0 = X$ and $Key1 = Y$ are written, bringing the current transaction ID (zxid) to #12 in the initial state. Recall that the upper bits in the ZooKeeper zxid encode an epoch, while the lower

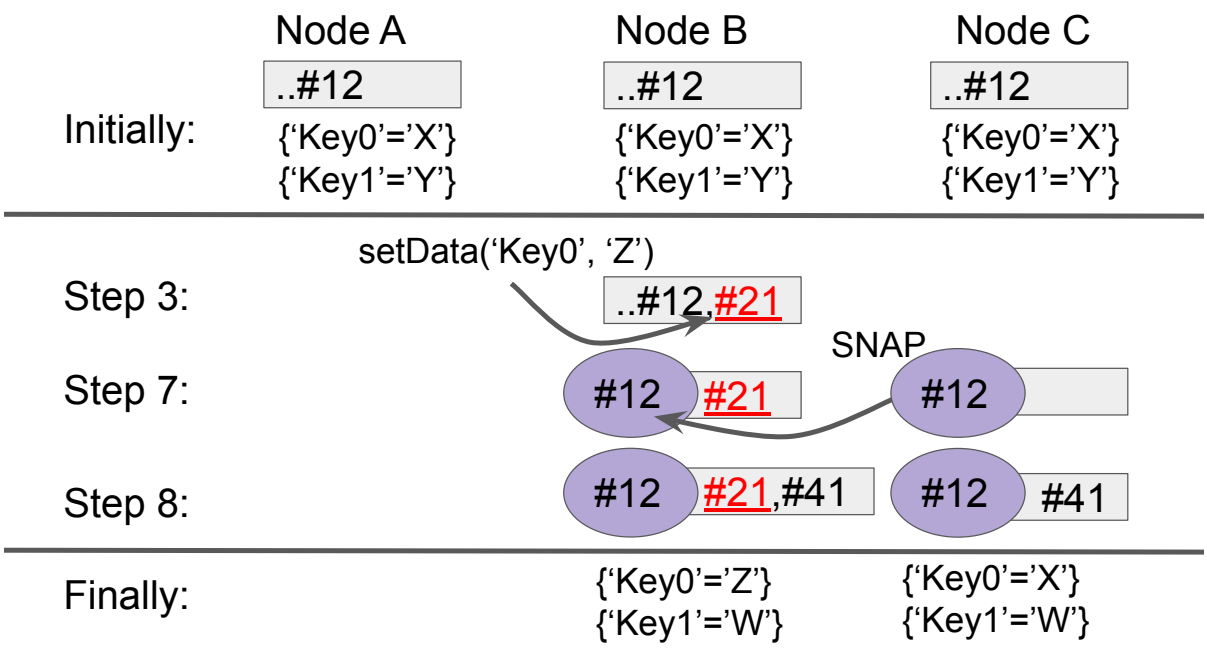


Figure 4.1: ZooKeeper Bug #2832 Illustration.

bits encode the number of transactions that have taken place in the epoch. For instance, the transaction ID #12 means the current epoch is 1 and 2 transactions have taken place in the epoch 1.

We then take the following steps to trigger the bug. At step (1), we crash all nodes and then restart nodes A and B. In this case, node B is elected as the leader, and the epoch increases from #1X to #2X. We then crash node A at step (2), and then, at step (3), we perform a write $Key0 = Z$ which node B accepts and commits as transaction #21. At step (4), node B crashes and at step (5), nodes A and C restart, and node A is elected as the leader, increasing the epoch from #2X to #3X. Node A resyncs with node C by sending a DIFF of transactions up to #12. Note that neither node A nor node C has seen transaction #21 yet at this point. At step (6), nodes A and C crash, and then at step (7) nodes B and C are restarted, and C is elected as the new leader. The epoch again increases from #3X to #4X. Because node C is the leader, it must resync with node B and use the internal ZooKeeper logic which selects the SNAP resync. Node B accepts and commits the SNAP but fails to truncate transaction #21 from its log, which is the root cause of the bug. However, at this point, the bug is not yet apparent. At step (8), we perform another write $Key1 = W$ which is committed as transaction #41. Node C commits the transaction and replicates it to node B, at which point node B believes it has converged with node C. Finally, nodes B and C crash at step (9) and restart at step (10). Since nodes B and C believe they are in sync, an empty DIFF is sent during the recovery. This triggers node B to replay outstanding transactions in its log and now persists the write $Key0 = Z$ into node B's key-value store, while nodes A and C still have $Key0 = X$. Because of the failure to truncate, the nodes believe they have converged, when in fact they have not, and the system permanently fails to reach eventual consistency.

As we can see from the example bug above, a series of failures and recovery procedures, along with asynchronous replication of write transactions, can lead to consistency bugs resulting in the violation of eventual consistency. One peculiarity about the bug described above is that it could be triggered merely by injecting external events, without exploring the various interleaving of concurrent internal events caused by multi-threading within

each node or network communication among nodes. Unlike existing model checkers, we focused on finding the specific class of consistency bugs leading to convergence failures by exploring the interleaving of externally injectable events. Therefore, our method does not require the interleaving of internal events, which makes our state-space exploration more targeted to find the specific types of CFBs. Moreover, the failure of convergence can be observed by reading data from distributed storage systems using their read API calls without inspecting the systems' log or snapshot files directly. It makes the integration with different implementations easier because we do not need to examine those files, which are formatted in a system-specific way.

## 4.3 Divergence Resync Models

DRMs are specific to each SUT and SUT-configuration. We first discuss the overview of DRMs. Then, we discuss DRM examples used to find CFBs in ZooKeeper, MongoDB and Redis. We name the DRMs using two criteria. The first is whether the SUT requires a quorum of nodes (more than one-half of the nodes) must be online to elect a leader or not. Our DRMs support both systems that require a quorum (Q) and those that are stand-alone (S). The second is how node failures are injected into the model. For failure methods, they can forcibly fail nodes by simulating a crash (C), by suspending their process (S), or by preventing nodes from communicating by simulating a link failure using an API (L). Models are named using the scheme $\langle quorum_requirement \rangle / \langle failure_modes \rangle$. We describe our models in more detail below.

### 4.3.1 DRM Overview

The DRMs we used in this work capture the behaviours of systems that use primary-backup replication schemes and maintain a complete key-value store replica on each node. Such replication schemes require a leader to ingest client operations, and the leader serializes all write operations and attaches timestamps. For example, ZooKeeper, MongoDB and Redis take client operations only when there exists a leader; that is, interchangeably, a primary or a master. All non-leader nodes, so-called followers (sometimes, secondaries or slaves), forward write requests they received to the leader. Then, the leader will apply the write operations locally first. Subsequently, the leader asynchronously replicates the write operations to followers. Although a leader must exist to ingest write operations, electing a leader may or may not require the quorum of nodes. Some systems need a quorum to elect a leader among them, while some systems allow to set a leader between a pair of nodes by invoking an API call explicitly. Also, some systems need a quorum to trigger resync, but others allow resync to occur between any arbitrary pair of nodes. Some systems automatically trigger resync, while others require an explicit request. Moreover, some systems enable a replication chain between followers, while others do not—with the replication chain, a follower can replicate not only from the leader but also from another follower. DRMs must reflect these differences.

DRMs model a specific behavior of distributed storage systems consisting of multiple cycles of the following events: (1) failures, such as crash or network failures, (2) divergence occurrences and (3) resync attempts. Accordingly, some subsets of nodes will fail (e.g., crash) first. Then, write requests will be injected. Online nodes will apply the client operation, while offline nodes fail to do so. Being *online* means the node is free of failure, converged and actively replicating, while being *offline* means the node is either experiencing a failure, not converged, or not actively replicating. After applying operations on online nodes only, there will be divergence created between online nodes and offline nodes. To examine if the divergence introduced can be successfully resolved by recovery mechanisms, we recover failed nodes (e.g., restart some crashed nodes) and discern if the

| Component | Description |
|---|---|
| numNode | Constant. Non-negative integer value. The number of replica nodes |
| nodeState | Variable. The list of non-negative integer values. The state of all replicas that defines the latest write operation applied |
| onlineStatus | Variable. The list of boolean values. The status of the node indicating if the node is online or offline |
| numAsyncOp | Variable. Non-negative integer value. The number of client operations that can be injected |

Table 4.1: DRM State Description

resync is triggered and correctly converges replicas. After executing multiple cycles of failures, divergence and resync attempts, we finally read values to check if replicas are successfully converged.

A DRM is essentially a finite state machine. Table 4.1 describes the state of the DRM. Each state is composed of one constant, *numNode*, and three variables: *nodeState*, *onlineStatus*, *numAsyncOp*. First, *nodeState* represents each node's current replica state. Each replica state is represented by a non-negative integer value, and nodeState is a list containing replica states of each node where the index of the list is mapped to each node. Second, *onlineStatus* represents which nodes are currently online or offline. Third, *numAsyncOp* represents the number of remaining injectable client operations. It must be configured by a user to determine the degree of divergence allowed and how many iterations of failure, divergence and convergence to execute. Table 4.2 shows the description of the DRM's transitions that will be applied to the current state and will produce the next state by making changes to the state variables. The DRM has two types of transitions: divergence and convergence. A transition is enabled only if associated *guard conditions* for the transition are satisfied. Guard conditions are a boolean expression resulting in true or false based on the variable values in the current state. The DRM also includes read and write operations to interact with the real system to execute test scenarios encoded by sequences of divergence and convergence transitions, which we will explain in Section 4.4.2.

Modulo explores the state space of a DRM to generate the sequence of test inputs, as described in Section 4.4.1. The DRM's state space is bounded by the number of nodes, number of write operations to inject and the fact that nodes that are already resynced cannot resync again as they are already converged. Similar to explicit model checkers, Modulo starts with the initial state and applies transitions to traverse a path to the next state. The path exploration eventually terminates, as the number of injectable write operations will become zero, and every node will be resynced. At the end of the path exploration, the explored path, which is the sequence of transitions, will be recorded as the *schedule*. Then, Modulo repeats exploring unexplored paths until its state exploration is complete. Modulo, therefore, employs state enumeration, which is widely used by explicit software model checking, to generate actual externally injectable test inputs.

Enabling transitions require careful consideration to cover every interesting corner case. To consider all possible ways to diverge, we can think about the given number of client operations to inject and how we will let them commit differently across nodes. For example, suppose we have one client operation to inject and three nodes are to be diverged. Then, we compute all possible divergence scenarios by considering how many client operations we inject will be committed by each node. For instance, with one client operation, we can have each node commit zero or one client operation. Assuming all nodes are symmetrical, we want to get the different combinations of the number of client operations committed on each node. Thus, all possible ways for each node to commit the given client operation will be $[0, 0, 0]$, $[0, 0, 1]$, $[0, 1, 1]$ and $[1, 1, 1]$, where $[0, 0, 1]$ means we will have only one out of three nodes commit the client operation, and $[0, 1, 1]$ means we will have two arbitrary nodes out of three commit

| Transition | Description |
|---|---|
| divergence | **numNode:** remains same<br>**nodeState:** increase integer value at the index corresponding to the node where client operations are applied<br>**onlineStatus:** set online status of nodes according to the divergence scenarios modelled<br>**numAsyncOp:** deduct the number of client operations to be applied |
| convergence | **numNode:** remains same<br>**nodeState:** synchronize the integer values at the index corresponding to the nodes that are supposed to resync<br>**onlineStatus:** set true for nodes previously offline but will be come online after applying this transition<br>**numAsyncOp:** no change will be made |

Table 4.2: DRM Transition Description

the client operation. Finding all such sets is known as computing multisets. Subsequently, we should filter out multisets that will not lead to divergence—e.g., $[0, 0, 0]$ and $[1, 1, 1]$. Therefore, each node's state change caused by a divergence transition can be modelled as a multiset we have named *target state change*. Hence, a target state change is the array of non-negative integer values. It is similar to nodeState, but it represents the number of client operations to be applied to each replica after applying the divergence transition. We compute all possible such target state changes in the given state of the DRM via multiset calculation. We only consider state changes of the online nodes because the invoked client operations can be replicated only by the online nodes. Finally, we can enable divergence transitions for all target state changes, i.e., multisets we calculated and filtered for the given state. Each target state change essentially corresponds to the distinct divergence transition that can be enabled in the state.

When enabling a convergence transition, we decide on a combination of offline nodes to be resynced. For example, if we crashed all three nodes after applying a divergence transition, we may only resync two out of three nodes or all three nodes. Yet, for systems requiring a quorum to elect a leader, we should ensure that we will restart the sufficient number of offline nodes to form the quorum. We do not enable convergence transitions that will have an insufficient number of nodes participating in resync. For instance, suppose we have three nodes in total and there is no online node at the given state. Then, we will enable convergence transitions that will have at least two nodes to participate—we will never enable convergence transitions involving just one node. To systematically explore resync scenarios, we need to consider all combinations of at least two offline nodes in this example.

The current state can be updated by applying enabled divergence or convergence transitions, which are described in Table 4.2. Applying a divergence transition will decrease as many available client operations as needed by the target state change. We also increase nodeState accordingly, referring to the target state change associated with the transition. For integer values at each index of the nodeState, we add the integer values at the corresponding index in the target state change. In addition, since we crash some nodes during divergence, we should adjust the online status accordingly. A divergence transition always occurs among online nodes. Thus, we can assume node states are symmetrical for those nodes that will be diverged by applying the transition. Even if we will have any deviation from this assumption, it is not a false positive because it merely means resync has failed in the previous convergence transition due to a CFB. Thus, we will detect such a resync failure during convergence verification procedure later on or in one of the shorter schedules.

Applying convergence transitions changes the onlineStatus and nodeState. The onlineStatus of the offline nodes that each convergence transition will restart should be set to $true$. Once nodes are recovered and the resync has occurred, the nodeState should be updated to reflect that the nodes that participated in the resync have converged. To do so, we can set all integer values in nodeState that correspond to all resync participants to the same value. Yet, instead of replicating one of the existing integer values in nodeState, we may also compute the new arbitrary integer value that is higher than any of integer values in nodeState. Subsequently, we set every corresponding index in the nodeState to the computed value to model the epoch increase in systems like ZooKeeper. As explained earlier, ZooKeeper uses the higher bits of timestamp for the epoch and the lower bits as the counter for operations committed within the specific epoch. Lately, MongoDB became another example of this—except that MongoDB uses the terminology 'term' instead of 'epoch.'

### 4.3.2 DRM Examples

**Q/C-DRM.** This model is used in systems such as ZooKeeper and MongoDB that require a quorum of nodes to be online to elect a leader and start servicing requests. In addition, we can simulate node failures by crashing the nodes in these systems. Since a leader is required to ingest write operations, the model is restricted in that it must ensure a leader is available before it can start injecting write operations. Similarly, these systems will not perform resync unless a quorum of nodes is available, so the model must ensure that enough nodes are recovered from failures to form a quorum.

**S/S-DRM.** This model is used in systems that do not require a quorum, such as Redis. In Redis, there is still a leader (called a Master), but the leader can be explicitly designated between any arbitrary pair of nodes. By suspending nodes and then resuming them, this DRM allows the graceful shutdown of the Redis nodes, which allows them to use a "partial" resync mechanism that attempts to retrieve missing operations only. Without using node suspension to simulate failure, Modulo would not be able to test Redis' partial resync mechanism.

**S/L-DRM.** Redis also supports "slave chains [1]", where a set of slaves will replicate from another slave instead of directly from the master. In this case, another failure mode is link failure, through which a node in the chain is not able to replicate data from its master. To implement link failure, this DRM uses the Redis API command "slaveof no one", which tells a node that it has no master, causing it to stop replicating data from its master. Convergence in this model involves having a node that stopped replicating from a master re-establish the link with a new master. Also, unlike ZooKeeper and MongoDB, Redis does not automatically resync after a node restart but instead requires an API call to invoke resync explicitly.

**S/CL-DRM.** This model includes both crash and slave chain failures and thus generates a larger number of schedules since Modulo has more possibilities to choose from for each divergence transition. Convergence transitions must recover nodes using a mechanism that corresponds to the failure mode—crashes must be recovered using snapshot resync, and link failures must be recovered using partial resync. This DRM can test interactions between the two resync mechanisms.

### 4.3.3 Discussion

We vary the number of writes in the write sequence between 1 and 5 in the above models. By alternating between divergence and convergence transitions along with the write, this typically results in models around 10 transitions

[1] See: https://redislabs.com/ebook/part-2-core-concepts/chapter-4-keeping-data-safe-and-ensuring-performance/4-2-replication/4-2-3-masterslave-chains/

long. Given the number of choices Modulo has at each divergence and convergence transition, the models produce anywhere from a few schedules to 100,000's. In comparison, models checked by modern explicit state model checkers contain millions to billions of states [65]. Our relatively small DRMs mean that Modulo can fully explore all scenarios that are possible under the given model, increasing the likelihood of finding bugs. Our main limitation is that the schedules generated by the abstract model must be run on the real SUT, which executes much slower than an abstract model would. Also, many operations require pauses and timeouts before they can complete. For example, a system may not consider a node failed until a particular time has passed, and leader election must end after nodes recover before the system can ingest writes. These pauses and timeouts are the ultimate limit to the number of and rate at which schedules can be tested and explored in Modulo.

## 4.4 Modulo

As shown in Figure 4.2, Modulo consists of two core components: a schedule generator and a concrete executor. To use Modulo, the user specifies a DRM and configuration parameters indicating the number of nodes (numNode) and the number of write operations to put in the write schedule (numAsyncOp). These are then used by the components to test the SUT. The schedule generator generates schedules by systematically and exhaustively exploring the state space of the DRM. The generated schedules are stored in schedule files and passed to a concrete executor who will execute schedules on the SUT. Since each schedule is independent, the concrete executor can parallelize the execution of the schedules on several instances of the SUT. After executing each schedule, the concrete executor checks for eventual consistency and flags schedules where violations of eventual consistency occur. The result of the eventual consistency check is stored in a result file alongside the file storing the executed schedule, so schedules causing convergence failures can be identified for further inspection. The schedule generator and the concrete executor both interact with the DRM over a DRM interface, which consists of functions and symbols that the DRM defines for Modulo components to call or access.

### 4.4.1 Schedule Generator

The schedule generator performs an exhaustive depth-first search traversal of the state space of a DRM. To begin, users should provide two configuration parameters, such as the number of nodes (numNode) and the number of client operations to inject (numAsyncOp). These parameters bound the state space of the DRM to explore. At each state, the schedule generator selects one of the enabled transitions at the given state and then applies it to traverse down to the next state. As the state space is bounded, traversing transitions will be terminated at a *leaf state* where no additional transition can be enabled. The sequence of transitions traversed through DRM states is called *path*, and traversing the path is called *path exploration*. When finishing path exploration, the schedule generator will pause the state exploration and record the explored path in a *schedule file*. The path starting from the initial state and ending at a leaf state is referred to as a *schedule*, which consists of the sequence of divergence and convergence transitions. The threshold can be set by users to limit how many schedules a schedule file can store. After reaching the threshold, the schedule generator will create another schedule file and start saving newly generated schedules into the new schedule file.

Figure 4.3 illustrates state exploration during schedule generation. From the initial state $S_0$ to $S_1$, path exploration applies a divergence transition that is crashing node B and performing a write operation. Then, it chooses to apply another divergence transition to get to $S_2$, which will eventually lead to the leaf state $S_n$. The explored path from $S_0$ to $S_n$ will be saved as a schedule. Then, for each intermediate state, prefixes leading to unexplored paths

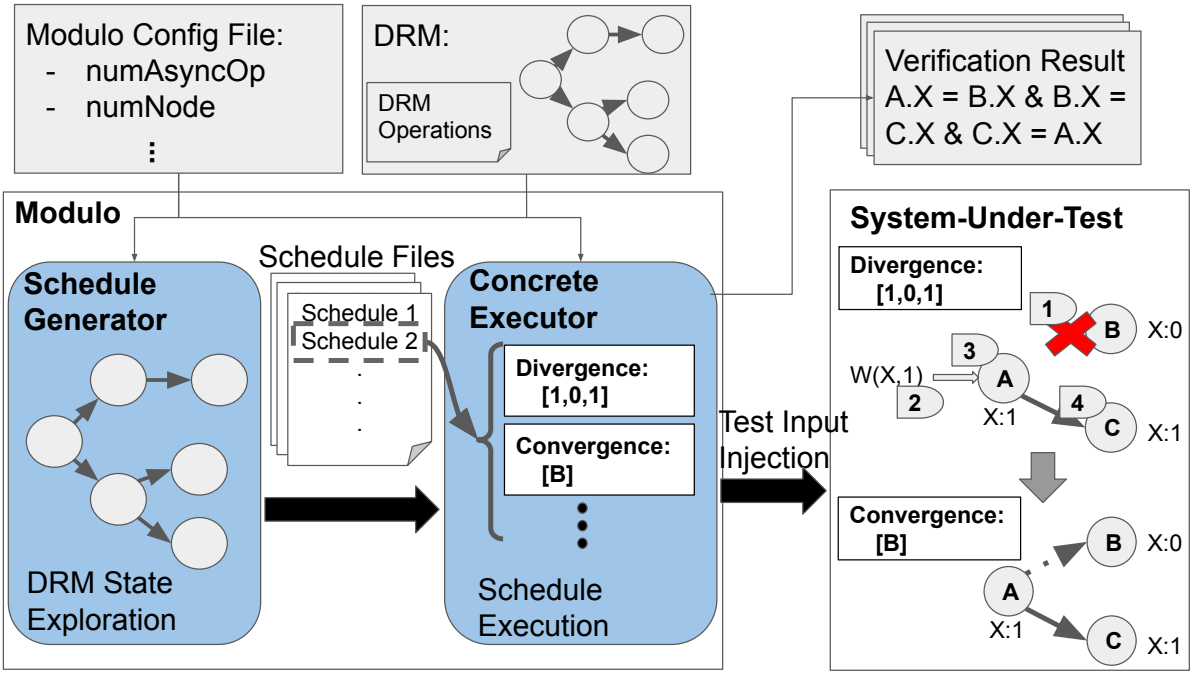


Figure 4.2: Modulo Architecture.

are found. For instance, at $S_1$, the convergence transition restarting node B was enabled and could be explored. The prefix from $S_0$ to $S_i$ will be found as it leads to an unexplored path. By replaying the prefix from $S_0$ to $S_i$, it fast-forwards to $S_i$, and path exploration continues up to an arbitrary leaf state $S_m$.

Algorithm 1 shows the state exploration procedure to generate schedules. At line 3, the initial state $S_0$ is instantiated with the given numAsyncOp and numNode, which are the number of write operations to inject and the number of nodes to involve, respectively. Then, path exploration starts from the initial state $S_0$ at line 4. At line 5, the explored path is saved as a schedule. Subsequently, for each intermediate state in the explored path, findPathPrefixes() finds all unexplored alternative paths that could lead to new states, and the resulting prefixes of paths will be stored in *pathPrefixes* at line 6. Then, at line 7, it goes into the while loop that keeps iterating until we cannot find a prefix leading to an unexplored path. At line 8, it backtracks to the initial state $S_0$. Then, a prefix leading to an unexplored path is popped from the *pathPrefixes* at line 9. Line 10 will fast-forward the

**Algorithm 1** Modulo Schedule Generation Algorithm

```
1:  procedure SCHEDULEGENERATION
2:      pathPrefixes ← φ
3:      S_0 ← init(numAsyncOp, numNode)
4:      exploredPath ← stateExploration(S_0)
5:      saveSchedule(exploredPath)
6:      pathPrefixes ← pathPrefixes ∪ findPathPrefixes()
7:      while pathPrefixes ≠ φ do
8:          reset(S_0)
9:          pathToReplay ← pop(pathPrefixes)
10:         S_i ← replayPath(pathToReplay)
11:         exploredPath ← stateExploration(S_i)
12:         saveSchedule(exploredPath)
13:         pathPrefixes ← pathPrefixes ∪ findPathPrefixes()
```

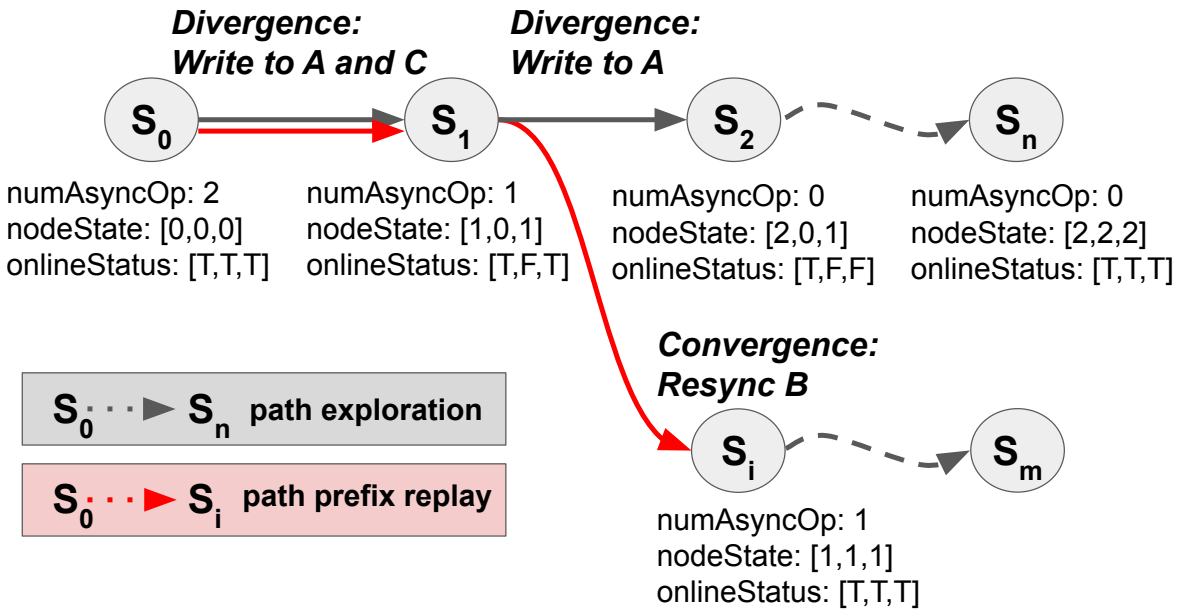


Figure 4.3: State Exploration during Schedule Generation.

path exploration to the intermediate state $S_i$ by traversing the prefix. From $S_i$, path exploration can continue at line 11, and a newly explored path can be saved as a new schedule at line 12. Afterwards, new prefixes leading to unexplored paths will be found for the last explored path and added to *pathPrefixes* at line 13. Finally, the steps between line 8 and line 13 are repeated by the while loop at line 7.

Modulo's schedule generator is implemented in Java. For some of the DRMs, the trees can be quite deep, which caused our initial recursive algorithm to exhaust the memory in the Java heap. We thus converted the implementation to an implementation that used a worklist and dynamic programming.

### 4.4.2 Concrete Executor

The concrete executor takes schedules from schedule files and executes them one at a time. Schedules consist of divergence and convergence transitions. Transitions are interpreted accordingly into a short sequence of actual test inputs to inject. Transitions are specified in the schedule file as the type of transition (divergence or convergence). Divergence transitions specify a target state change, which encodes which nodes to crash and how many write operations to inject. The concrete executor determines how many client operations need to be injected for a divergence by finding the highest integer value in its target state change. The concrete executor will then interleave writes and failures to change nodes' states according to the target state change. For example, in Figure 4.2, suppose we have three nodes: A, B and C, and the initial nodes' states set to $[0, 0, 0]$. The schedule 2 in the example specifies a divergence transition with the target state change $[1, 0, 1]$. This target state change means that node B will diverge from nodes A and C by one write operation. The executor fails node B first. Then, it injects a write into the SUT. Subsequently, it waits until nodes A and C commit the write. If the target state change had been $[2, 0, 1]$, the concrete executor would perform additional divergence steps. It fails node C after the first write and performs the second write, which only node A can commit. Then, the executor waits until node A commits.

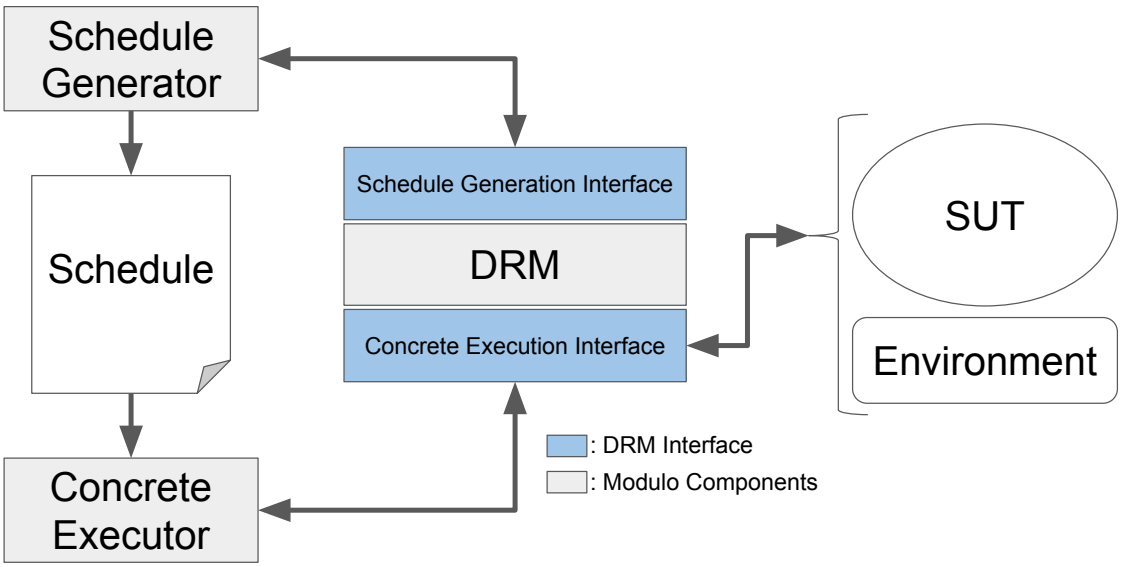


Figure 4.4: DRM Interface Architecture.

Later, as shown in Figure 4.2, a convergence transition on nodes $[B]$ is specified by the schedule. To execute the convergence transition, the concrete executor restarts node B and then has node B resync with online nodes. The concrete executor needs to understand how to interact with the specific SUT and its configuration as well as the environment on which SUT is running. Users should implement the DRM interface for the concrete executor to properly inject test inputs and control the environment. The DRM interface to implement is further described in Section 4.4.3.

When the concrete executor finishes processing each schedule, it performs a *consistency verification* procedure. It first ensures that all nodes are recovered and online. Second, the concrete executor performs a DRM-specific procedure to ensure that the system is quiescent—that is, there is no pending write operation and all nodes are synchronized. Finally, the concrete executor verifies consistency by performing a read API on each node and comparing the results for equivalence across the nodes. Any divergence between nodes' states is recorded in a file alongside the executed schedule, and usually indicates a CFB. The schedule that is associated with such a failure is equivalent to a counter-examples that would be provided by an explicit-state model-checker [77].

The concrete executor currently parallelizes execution by splitting the schedule files into equal sized batches and scheduling them to be run by many concrete executor instances running on different physical machines. In the future, we may achieve better load balance by having a work-queue of schedules and concrete executor instances drawing work from the central work-queue. Since each schedule takes some time to execute, we do not foresee a centralized work-queue becoming a point of contention unless Modulo runs with a vast number of parallel instances.

### 4.4.3 DRM Interface

Modulo generates schedules and executes each of them to inject test inputs to SUTs by interacting with DRMs via DRM interface, which users should implement and provide. How the schedule generator and concrete executor work via the DRM interface is illustrated in Figure 4.4. Here, we discuss more detailed specifications about the DRM interface. The basic DRM interface will be presented, but it can be easily extended for more complex

| Name | Description |
|---|---|
| enum TransitionType | Types of transitions |
| Class AbstractState | The class contain various state variables and related functions |
| void initAbstractState(AbstractState curState) | Initialize the abstract state for the schedule generator |
| void getEnabledTransitions(AbstractState curState) | Find newly enabled transitions in the current state |
| boolean isGuardCondSatisfied(TransitionType tranType, AbstractState curState) | Check if a given tranType transition can be enabled in the given curState |
| boolean diverge(AbstractState curState) | Update state variables accordingly |
| boolean converge(AbstractState curState) | Update state variables accordingly |

Table 4.3: Schedule Generation Interface to Implement

DRMs.

#### Schedule Generation Interface

Table 4.3 specifies the symbol, classes and functions the DRM must implement for a particular SUT. We discuss each in more detail below.

**`TransitionType`.** This enumerates the different transition types, such as divergence and convergence. It can be extended with more specific types. For example, we may extend it to additionally express a convergence using offline resync in addition to online resync. Offline resync is manually copying a snapshot of a node to restart with, while online resync is restarting nodes to trigger built-in resync mechanisms.

**`AbstractState`.** This class contains various state variables such as numNode, nodeState, onlineStatus and numAsyncOp. It can be extended with more variables. For instance, a variable to capture how nodes are failed currently—crashed or disconnected—can be added. Also, it may contain a customized function to compute the hash of the current state so as not to explore the same states repeatedly.

**`initAbstractState`.** This function initializes the state with which the schedule generator's state exploration begins. For instance, initially, nodes may be all online and several writes may be committed. All variables in AbstractState should be appropriately set.

**`getEnabledTransition`.** This function will explore all possible cases of divergence for the given number of nodes and write operations to use. It will perform multiset computation to enable divergence transitions and will find a set of nodes to recover to enable convergence transitions.

**`isGuardCondSatisfied`.** This function checks if a transition of the given type can be enabled in the current state. For example, if an offline node exists, this function will return true for the convergence transition to be enabled. Another example is that if several nodes are online and the number of injectable write operations is greater than zero, this function will return true for the divergence transition to be enabled. Based on the return value of this function, getEnabledTransition accordingly enables transitions.

**`diverge`.** This function updates the variables in the current abstract state according to the given divergence transition. As an example, if we diverge several online nodes in a way so that every online node crashes after divergence, this function will set onlineStatus to be false for failed nodes. Also, it should update nodeState

| Name | Description |
|---|---|
| enum FailType | The types of failures used in the DRM |
| void initConcreteState() | Initialize the concrete SUT state for concrete executor |
| void recoverNode(int nodeID, FailType failType) | Recover the specified node after a failure |
| void failNode(int nodeID, FailType failType) | Inject a failure for the specified node |
| byte[] readData(int nodeID, String key) | Read a key value from the specified node |
| void createData(int nodeID, String key, int value) | Given nodeID, key and value, sending the node with nodeID the request to create data record for the first time with key as the identifier and value as the content |
| void writeData(int nodeID, String key, int value) | Write a key on the specified node with the specified value |
| boolean waitForCommit(int[] commitNodeIDs, String key, int value) | Wait until a specified value, has been committed to a specified key on the nodes in commitNodeIDs |
| boolean waitForResync(int[] resyncNodeIDs) | Wait until for all nodes in the resyncNodeIDs have completed resync. |
| int findLeader() | Find the ID of the node whose role is a leader |

Table 4.4: Concrete Execution Interface to Implement

to appropriately reflect the number of write operations committed on each node. Moreover, the numAsyncOp should be decreased as much as the given divergence scenario needs.

**`converge`.** This function updates the variables in the current abstract state according to the given convergence transition. For instance, similar to the divergence transition case, it should be set to true in onlineStatus for offline nodes that participated in convergence. Also, nodeState should be updated to reflect that converged nodes are appropriately synchronized.

### Concrete Execution Interface

Table 4.4 specifies the symbol and functions the DRM must implement for a particular SUT. We discuss each in more detail below.

**`FailType`.** This symbol enumerates the different failure modes, such as crash, suspend and link failure, that a node may experience.

**`initConcreteState`.** This function initializes the state of the SUT that the concrete executor begins with. Depending on how a DRM models the initial state, nodes may need to be brought up, and links between them must be set up correspondingly regarding the modelled abstract initial state. Also, key-value stores may need to be pre-populated properly.

**`recoverNode`.** This function will recover the node specified by `nodeID` from a failure. The schedule will also specify the `failType`, which must correspond to the way the recovered node failed. This way, the DRM can select the appropriate recovery type. For example, in the S/CL-DRM described in Section 4.3.2, a node that failed due to link failure must be recovered with a corresponding link restore operation.

**`failNode`.** This function generates a node failure on the SUT. For example, to simulate a crash in the Q/C-DRM, the `failNode` function for that DRM calls 'kill -9' on the target node process.

**`readData` & `createData` & `writeData`.** These functions are wrappers around the read and write APIs for

the SUT. `readData` is used to read each key-value from each node during convergence verification procedure. `createData` is usually used to pre-populate the key-value store if the SUT differentiates an API call to create a new key-value from an API call to update an existing key-value. Typically, the `writeData` function calls an asynchronous write API so that the SUT will asynchronously replicate the data.

**`waitForCommit`.** This function waits until the specified node has replicated and committed the specified write. As we can see, the key-value tuple for each write must be unique so that this function can easily tell if the desired write has been replicated. There are two general methods for implementing `waitForCommit`. The alternative way is just to sleep for a specified amount of time after which all online nodes have expectedly replicated the write. However, this method may not be reliable and may also slow down testing throughput with unnecessary waiting time. A more sophisticated approach would be scanning the log files of each of the nodes to see if they have replicated the value. We currently use a timeout of 60 seconds in our DRMs.

Interestingly, it might appear that using `readData` to check if a node returns the latest result might also be the third way of checking to see if data have been committed. However, this turns out not to be reliable for some systems, as they may commit some writes locally, but not return them to a readData API call until some number of nodes has committed them. For example, ZooKeeper exhibits this behavior and will not return committed writes until the quorum has committed them.

**`waitForResync`.** This function waits until the recovered nodes have completed resync. Like `waitForCommit`, a generic way to implement `waitForResync` is to have a timeout bound within which all nodes have expectedly completed their resync procedures. However, it is also possible to implement SUT-specific methods. For example, ZooKeeper will only complete leader election after all nodes have resynced, so a simple way is to test whether a leader has been elected or not, which can be ascertained by scanning the log files on each node. MongoDB does not require resync to be completed for a node to become a Master, though. As a result, we use administrative API calls such as *replSetGetStatus*, which return the internal status of each node and allows us to retrieve the list of committed operations. We can infer that resync is complete when the last committed operation matches that of the other nodes.

`waitForResync` may never terminate because some nodes never commit the latest operation according to the schedule. In this case, a CFB has likely been found. Thus, `waitForResync` must implement an internal timeout, which, if triggered, will cause the executor to halt and report a convergence failure.

**`findLeader`.** This function returns the ID of the leader node. If there is currently no online leader, it returns -1 instead. Because only the leader will ingest writes in the primary-backup architecture, we need to determine the leader before starting divergence. Once we have determined the leader, we can make the write request to the leader. For systems requiring a quorum to elect a leader, the return value of this function is enough to tell if the quorum is formed successfully and the SUT is ready to serve client operations.

## 4.5 Implementation

Modulo is implemented in roughly 6.4K lines of Java code. The schedule generator is implemented in 1.6K LOC, and the concrete executor takes about 4.2K lines of code. Other miscellaneous utility functions or common data structures are written in 0.6K lines of code. The concrete executor can be further divided into two parts: the core and the system controller. The core is implemented in 0.7K lines of code, while the system controller takes roughly 3.5K lines of code.

Q/C-DRM requires about 0.9K LOC, S/S-DRM requires about 1.3K LOC, S/L-DRM requires about 1.3K

LOC, and S/CL-DRM requires about 1.6K LOC. A significant part of the DRM implementation consists of SUT log parsing, API interaction and analyzing to infer the state of SUT nodes. To give a concrete example, ZooKeeper records whether a node is a leader or not in its log file as follows:

```
LEADING - LEADER ELECTION TOOK - 230
Follower sid: 0 : info : ...
Synchronizing with Follower sid: 0 ...
```

Similarly, a follower node will log the following messages:

```
FOLLOWING - LEADER ELECTION TOOK - 217
Resolved hostname: 127.0.0.1 to address: ...
Getting a diff from the leader 0x100000009
```

Thus, the DRM can scan log files of ZooKeeper to look for messages containing either `LEADING` or `FOLLOWING`, which indicate which role they have switched to, which we will discuss more in Section 4.5.1. The DRM needs to keep track of which node is the leader to successfully inject write operations to cause divergence because only the leader node can ingest write operations.

Before delving into the details of integrating Modulo with target replicated distributed storage systems, here we describe each system briefly to give some background. First, ZooKeeper is a distributed key-value store for providing a coordination service for large distributed systems. It must elect a leader and keep followers synchronized with it. It implements its leader election protocol by having the majority of nodes agree on a new leader for each epoch before starting to service. Second, MongoDB is a document-oriented NoSQL database system, using JSON documents as its data items. It also uses primary-backup replication scheme. Unlike ZooKeeper, it allows chain replication where a secondary can be a primary for other secondaries. Third, Redis is a distributed in-memory key-value store with optional durability. Redis also uses a primary-backup replication scheme, but it requires for users to explicitly specify which node between two becomes the source of synchronization. It also allows chain replication.

### 4.5.1 ZooKeeper Integration

ZooKeeper requires a quorum to elect a leader and needs a leader to be available to ingest a write operation. Thus, we need to wait until ZooKeeper nodes have started and the quorum is formed with an elected leader after we have restarted a sufficient number of nodes. When the quorum is formed, restarted nodes will switch their roles after the resync completes. Accordingly, switching the role to either a leader or a follower is the necessary condition to identify the completion of the resync in ZooKeeper. Therefore, learning what the node's roles currently are is an important feature to integrate Modulo with ZooKeeper. We implemented the *findLeader* API function for ZooKeeper using a log scanning approach. ZooKeeper leaves log messages indicating the result of the leader election. Thus, the log files of ZooKeeper nodes contain the string, such as "LEADING," "FOLLOWING" and "LOOKING." Depending on which role each node has at the moment, one of "LEADING" or "FOLLOWING" will appear. "LOOKING" means the node has not switched its role to either a leader or a follower yet. The fact that such messages are logged implies that ZooKeeper has completed the leader election. Each ZooKeeper's log file is scanned, and the leader node can be identified if the log file contains the message "LEADING." Other online nodes will have "FOLLOWING" in their log files. The findLeader function will then simply return the node ID of the leader node.

*readData*, *createData* and *writeData* are implemented using the ZooKeeper API for clients. For readData and createData, we simply used the synchronous version of *getData* and *create*, respectively. However, writeData is implemented using asynchronous *setData*, because we want to cause divergence between online nodes and offline nodes during the divergence path execution. Write operations such as createData and writeData will always be sent to the leader, because only the leader can ingest the write operations.

The *waitForCommit* function is implemented by using the sleep method, although ZooKeeper log messages can be scanned and searched for a specific string, indicating the commit is completed. The idea is that we can wait long enough, so the server node successfully commits the write operation with a very high probability. We used 60 seconds for the duration; we wait after injecting a write operation to a leader. We generated a small-scale load for write operations, as we are not doing stress testing for the SUT. Accordingly, we believe 60 seconds is long enough to be sure that the injected write operations are committed.

As ZooKeeper only switches the node's role to either a leader or a follower after the resync completes, *waitForResync* simply keeps waiting until the leader is elected. After this function returns, every divergence between nodes that participated in the resync is expected to be resolved. Also, every online replica is expected to be converged.

### 4.5.2 MongoDB Integration

Similar to ZooKeeper, MongoDB also leaves log messages in log files indicating the role switch of each server node. We implement *findLeader* by scanning the log files of each online node. We look for string "primary" or "Secondary" and the node whose log file contains the string "primary" is determined to be the leader.

MongoDB allows clients to read from a specific server node with certain configuration options for a read operation. When we implement *readData*, we use the *find* MongoDB API call with the option *secondaryPrefered* along with the tag feature. By using the *secondaryPreferred* read option, we can set up the MongoDB cluster to allow clients to read from a secondary as well. Then, with the *tag* feature, we can associate a distinctive tag for each node. We used the node ID we assigned to each node as the tag for each node. When clients send a read request, clients can specify the tag of the specific node from which they want to read. As we give a distinctive tag to each node, by setting the tag for each read request, we can read data from the specific server node, no matter which role the node has. For *createData*, we use the *insertOne* MongoDB API call and, for *writeData*, we use the *updateOne* MongoDB API call for implementation. Both API calls are to be used to either insert or update one entry instead of a batch of multiple entries. In MongoDB, replication between the primary and a secondary node is asynchronous for both types of write operations.

Unlike ZooKeeper, we use a log scanning technique to implement *waitForCommit*. MongoDB leaves a log message containing strings such as "_id" and "value," followed by the key and value for the retrieved document (i.e., the data item of MongoDB).

To implement *waitForResync*, we take a different approach from ZooKeeper's. Because MongoDB does not guarantee that resync is complete when the node's role is switched to either primary or secondary, we need to perform further investigation about the internal status regarding the resync's progress. We utilize administrative API calls such as *replSetGetStatus*. The command produces an output document containing various status information related to the *replica set*, which is the MongoDB terminology for the ensemble of replicas kept synchronized. We can figure out each node's most up-to-date *optime* (i.e., the internal timestamp associated with each event). To do so, we parse the document returned by the command *replSetGetStatus* to retrieve the embedded document stored in the "members" field. From that document, we can retrieve the value of the field "optime." Among the optime values of each node, we figure out the latest value. Then, we wait until every node's optime value becomes

converged to the latest one. If so, nodes are converged, and the resync is considered complete. If this condition is not met for a long time, such as 10 minutes, we conclude that convergence will never be achieved, probably due to bugs.

Furthermore, *beforeDivergencePath* is implemented as an extension to the DRM interface for MongoDB to prevent the system from going into a state where no more write operation can be ingested during the divergence path's execution. MongoDB will automatically step down the primary if there are not enough nodes available to sustain the quorum (i.e., the majority of replica set members should be online and connected). MongoDB uses a heartbeat mechanism to keep track of the status of each node. Heartbeat messages are exchanged at every heartbeat period, and the status of each node is checked. If the primary node figures out that the quorum cannot be sustained due to the lack of available nodes, it steps down. For instance, this may happen when we are about to execute the divergence path. While executing the divergence path, we may crash secondary nodes, and we can end up having only the primary available to ingest the last write operation to maximize the extent of the divergence between primary and secondary nodes. If the heartbeat period of MongoDB is too short relative to the speed of injecting write operations [2], MongoDB may mistakenly step down the primary node before successfully committing all injected write operations. Then, there will be no quorum and no primary to ingest the last write operation injected for the divergence path. To prevent MongoDB from mistakenly making the primary step down due to the overhead imposed by our tool, we set the heartbeat period to an arbitrary but much longer value just before executing the divergence path. Yet, after the divergence path, we often need to execute a subsequent resync path. During the resync path, we restart nodes to form the quorum and elect a new primary. If the heartbeat period is left as the long value that we set before executing the divergence path, the primary cannot detect other nodes because heartbeat messages will not be exchanged anytime soon. That is, nodes will have no idea if others are available until the heartbeat occurs. Thus, we restore the default heartbeat period at the beginning of the resync path. Adjusting the heartbeat period was done using administrative API calls provided by MongoDB.

### 4.5.3 Redis Integration

Redis is different from ZooKeeper and MongoDB because it does not require a quorum to start having a *slave* replicate from its *master*—master and slave are Redis' terminology for leader and follower, respectively. Unlike ZooKeeper and MongoDB, clients can flexibly specify a *sync source* for each slave from which the slave replicates. Even a slave can become a sync source of another slave so nodes can form a replication chain. Therefore, to find a master, we do not need to inspect the internal state, but simply remember which node we made the sync source of each node as an array of integers; that is, the array of the sync source's node IDs. Thus, for Redis, the *findLeader* function is simply iterating the array of the sync source's node IDs to look for the online node without any sync source. In addition, to trigger resync after nodes recover from failures, we need to explicitly invoke a command for each node to have it replicate from a specific sync source. Accordingly, we implemented *triggerResync* as an extension to the DRM interface. The function is implemented using the "slaveof [IP] [PORT]" API call, and it gets invoked after recovering nodes during a convergence transition execution.

With Redis, we experimented with several failure modes, including the crash failure, suspending nodes and bringing down network links. To implement *recoverNode* to resume nodes and *failNode* to suspend nodes, we used 'kill -CONT' to resume the process and used 'kill -STOP' to suspend the process. Moreover, to bring up and down links, we used "slaveof [IP] [PORT]" and "slaveof no one."

For *readData*, we used Redis's *get* API call. For both *createData* and *writeData*, we used the *set* API call.

[2] It becomes more likely to happen due to artificially injected sleep statements or multiple iterations of log scanning done by DRM interface functions, such as *waitForCommit*.

| Bug ID | Root Cause |
|---|---|
| ZooKeeper Bug #2832 (**New!**) | Fail to truncate operations due to missing invocation |
| ZooKeeper Bug #2946 (**New!**) | Fail to truncate operations due to file handling mistake |
| First MongoDB Bug (**New!**) | Fail to truncate operations due to incomplete timestamp information |
| Second MongoDB Bug | Fail to replicate operations due to incomplete protocol design |
| Redis Bug #2694 | Fail to invoke snapshot sync due to incomplete protocol design |
| Redis Bug #4316-1,2 | Fail to replicate operations due to lacking resync related information |
| Redis Bug #4407 | Fail to truncate operations due to incomplete protocol design |

Table 4.5: CFB Root Cause Analysis

*waitForCommit* uses a log scanning method as the default. More specifically, we directly scan the commit log because Redis's commit log contains data as a plaintext string without any encoding.

*waitForResync* is implemented by relying on both internal status inspection via an *info* API call and sleep method. The info API call will output detailed information about various internal statuses of Redis nodes. As the indication of resync completion, we make sure *master_link_status* is "up," *master_sync_in_progress* is 0, *aof_rewrite_in_progress* is 0 and *rgb_bgsave_in_progress* is 0. Until then, we repeatedly wait and time out after the period that is long enough to conclude the failure of converging. However, this method occasionally fails to identify that the resync completes correctly. Hence, on top of monitoring each status variable explained previously, we wait for a short period additionally, a few seconds, before returning.

## 4.6 Evaluation

We present our results from running Modulo on three well-known, open-source, distributed database systems: ZooKeeper, MongoDB and Redis. Q/C-DRM was used for both ZooKeeper and MongoDB experiments, while the other DRMs are used on Redis. We summarize the CFBs we found in Table 4.5 and discuss them in more detail below. Finally, we also tabulate the performance of Modulo on our three SUTs.

### 4.6.1 ZooKeeper Bugs

We discovered two new bugs in ZooKeeper version 3.4.11. We reported them to the ZooKeeper developers, and they were designated bugs #2832 and #2946 [52, 54]. ZooKeeper bug #2832 is described in Section 4.2. All our experiments used three nodes, and the same initial condition illustrated in Figure 4.1 is kept in all runs. Also, the Q/C-DRM implementation for ZooKeeper crashes all nodes after each divergence transition and begins convergence by restarting nodes enough to form a quorum, which automatically triggers resync between each node and the elected leader. In the following paragraphs, we will skip these details in the bug reproduction steps presented for each bug. Also, we simplify the value of zxid's for ease of presentation—e.g., 30000001 to 31.

**ZooKeeper Bug #2832.** This bug is previously described in Section 4.2 and occurs because of a missing truncation operation. As a result, the untruncated operation remains in the transaction log and gets replayed, resulting in a convergence failure.

**ZooKeeper Bug #2946.** This bug occurs because of an error in the code that implements the transaction log file, again causing transactions that should have been truncated to remain in the transaction log incorrectly. Modulo

finds the bug after executing the following schedule on nodes A, B and C: (1) invoke a write operation on C with zxid #11; (2) resync A and B; (3) invoke a write operation on B with zxid #21; (4) resync A, B and C; (5) invoke a write operation on C with zxid #31; (6) resync A and C; (7) invoke a write operation on C with zxid #41; (8) resync A and B; (9) invoke a write operation on B with zxid #51; and (10) resync A and C. At this point, C is restarted and resynced with A but fails to truncate zxid #41 from step 7. C actually attempts to truncate all transactions after zxid #21, but an error in the file management code causes the failure to truncate zxid #41. Finally, as step (11), B is restarted and resynced with A. Similar to bug #2832, the untruncated transaction at zxid #41 is replayed only on C, and the system fails to guarantee eventual consistency.

**Discussions.** ZooKeeper implements two different resync mechanisms: DIFF resync and SNAP resync, which have complex and non-intuitive interactions. ZooKeeper dynamically selects which mechanism to use depending on how far behind the recovering node is from the leader. If the difference is not large, it uses the DIFF resync, and only uses the SNAP resync if the differences are large. For the first bug, Modulo generates some failures that initially use the DIFF resync, and then generates enough divergence so that the SNAP resync is invoked. However, the logic fails to correctly truncate the transaction logs with the SNAP resync to resolve conflicts consistently across all replicas. Accordingly, a CFB manifests and leads to persistent inconsistency. This problem is likely further exacerbated by an apparent reluctance to truncate logs aggressively, perhaps out of a conservative preference to not lose data unless necessary.

In addition, ZooKeeper maintains the transaction log as a set of files, each of which is for each epoch, which can create complex formations of log files across nodes. For the second bug, Modulo generates enough divergence for ZooKeeper to keep and manage several files for the transaction log. While the given zxid for the truncation logic is supposed to be used as the starting point to truncate any newer transactions, the error in the logic prevents ZooKeeper from correctly truncating transactions contained in files, later than the zxid specified for the truncation. That is, the root cause behind bug #2946 appears to be an error where developers did not consider that the number of log files could grow to a certain amount and neglect to check all log files for operations that should be truncated. Thus, this problem is perhaps caused because developers could not expect all possible formations of the transaction log files across multiple nodes after having divergence and going through several failure recoveries.

While both bugs are found in the same version of ZooKeeper, we found we could apply Modulo to several versions of ZooKeeper without any trouble. As a result, Modulo can also serve as a regression testing tool to ensure consistent behavior across versions.

### 4.6.2 MongoDB Bugs

We ran Modulo on MongoDB version 3.0.0 and discovered two bugs. One is a unknown bug which we reported on the MongoDB Bug Database [53], while the other one is the one already reported and fixed in a newer version of MongoDB. Similar to the ZooKeeper experiment, we use Q/C-DRM and three nodes in the DRM. The initialization procedure used for the MongoDB experiment is same as the ZooKeeper experiment. For presentation purposes, we also simplify the value of optime, which is the MongoDB transaction ID equivalent to the ZooKeeper zxid.

**MongoDB Bug #1.** This CFB occurs because MongoDB does not keep track of enough information to tell which operation is newer when comparing two operations during crash recovery. The steps Modulo applied from the schedule that trigger the CFB are: (1) invoke a write operation to A with optime #43, at the same time, B and C have #38 as their most recent operation, which A has not replicated yet; (2) resync B and C—after which they still have #38 as their most recent operation; and (3) resync A. Even though A has a newer operation, it is the

one trying to join the quorum consisting of B and C. Since the MongoDB node is designed to refuse to resync with more "stale" nodes, A fails to resync with B and C, and the write with optime #43 is neither truncated nor replicated to others. Consequently, nodes A, B and C remain in a conflicted state, and the system fails to achieve eventual consistency.

**MongoDB Bug #2.** This bug occurs because not enough old operations are in the log. Modulo executes the following steps to reproduce the bug: (1) invoke multiple (e.g., 10) write operations to A; (2) resync A and B—B does a snapshot sync with A, which is full synchronization that replicates the snapshot directly and adds only the last operation to the MongoDB oplog (MongoDB's transaction log), which bumps up the optime of B to #377; (3) invoke multiple write operations on A; (4) resync A and C—similar to step 2, C does a snapshot sync with A, which adds only the last operation to its oplog and bumps the optime of C to #681; (5) invoke multiple write operations on A; and (6) resync B and C—B contains only the write operations at optime #377 in its oplog and C contains only the write operations at optime #681 in its oplog. Note that C does not have operations between #377 and #681. Because B's oplog is not empty, B tries to do partial sync, which is a resync operation that attempts to only copy missing operations. However, since C does not have operations between #377 and #681, B fails to replicate those operations it needs. Consequently, the system fails to converge.

**Discussion.** The first MongoDB bug we discussed was introduced because developers failed to consider a situation in which an old primary comes back with data not yet replicated to other nodes, while a new primary node and secondary nodes have not yet committed any new updates. Although this is a simple bug to reproduce, developers were not aware of this problem in advance because they did not have a systematic testing tool to find CFBs. While this bug had been previously found and fixed, it was difficult for developers to pinpoint the root cause of the bug and fix the issue. In the end, the bug was fixed when MongoDB developers introduced the concept of epochs to MongoDB, which fixed this and several other issues by fundamentally changing the way MongoDB synchronizes nodes. We believe that if the MongoDB developers had access to a simple schedule that triggered the original bug, such as the one that was found by Modulo, the problem could have been fixed in a much more direct and expedient way.

The second MongoDB bug was caused because the MongoDB developers did not consider a scenario in which a primary node has a non-empty oplog but does not contain all old operations, which are required by a secondary node for resync. As reproduction steps are complex, it is likely that the MongoDB developers simply could not foresee the sequence of events that would have been required to trigger this CFB.

### 4.6.3 Redis Bugs

Redis is significantly different from ZooKeeper and MongoDB in that it does not require a quorum to elect a leader. Also, Redis does not automatically resync recovered nodes but expects the administrator to tell the new node which node to resync from manually. As a result, the DRM needs to explicitly tell recovered Redis nodes to resync and specify which nodes to sync from. Similar to MongoDB, Redis supports slave chain replication. Thus, the master is also called a *sync source*, while slaves are called *sync targets*. Moreover, because Redis allows slaves to replicate from another slave node, a slave node can be a master of its slaves, allowing daisy-chaining of nodes for replication.

Modulo uses S/S-DRM, S/L-DRM and S/CL-DRM to test Redis. Modulo found four CFBs in Redis versions 2.8.0 and 4.0.0, all of which we confirmed had been previously found. We discuss these CFBs in more detail below.

**Redis Bug #2694.** Redis bug #2694 [38] occurs in Redis 2.8.0 when a node is suspended while a new primary

joins and forms links with other nodes. The new primary node may have a completely different dataset than other nodes. Suppose the suspended node resumes later and attempts to replicate from another secondary node after all other secondary nodes already resynced with the new primary. Then, the resumed node will fail to resync properly because the secondary node fails to determine what to replicate to the resumed node accurately. This CFB was found with the S/S-DRM.

To reproduce the bug, we need to initialize four nodes: A, B, C and D. The CFB-triggering schedule specifies the following steps: (1) suspend A, B and C, write to D and suspend D; (2) resume A and B and let B resync with A; (3) resume C and let C resync with B; (4) suspend D, write to A and let B and C replicate the write and then suspend A, B and C; (5) resume node A and B; (6) resume D and let A replicate from D—A and B will have completely different datasets replicated from D and (7) resume C—C does not do snapshot sync (called init sync in Redis documentation) but do partial sync from B while B now has a completely different dataset than from what C has. As a result, C fails to truncate operations for the old dataset and fails to replicate operations for the new dataset.

**Redis Bug #4316-1,2.** These two bugs are reported in Redis bug #4316 [92], which occurs in Redis 4.0.0 when offline resync is used while nodes are experiencing crash failures or replication chain failures. Offline resync is a procedure where a snapshot is explicitly copied over from one node to another node replacing a failed node and the new replacement node restarts using the snapshot (the Redis RDB file). However, the RDB file does not contain enough information to synchronize diverged nodes properly, and, therefore, resync fails to resolve divergence. S/CL-DRM was used to find this CFB.

The bug report contains two bugs. The first we will call Redis Bug #4316-1, and the second we will call Redis Bug #4316-2. These bugs require only two nodes: A and B, where B is initially set to replicate from A. To reproduce Redis Bug #4316-1, we execute the following steps: (1) crash B and write to A; (2) offline resync B from A; (3) write to B—because there is no replication chain between A and B, only B commits the write; and (4) have B replicate from A—B fails to truncate the write operation made at step 3.

To reproduce Redis Bug #4316-2, slightly different steps should be executed: (1) crash B and write to A after switching to dataset 1 while the default dataset is in DB 0 (where DB is analogous to a table in RDBMS); (2) offline resync for B from A and (3) have B replicate from A—this does not replicate switching to DB 1 on B. The consistency verification procedure for this DRM was modified to perform a final write operation to the primary A. The last write will be written to DB 1 on A while replicated to DB 0, the default one, on B. Consequently, B fails to replicate the write operation to the correct DB, so convergence fails.

**Redis Bug #4407.** Redis bug #4407 [32] occurs in Redis 4.0.0 when the replication chain failure exceeds a time limit set by the Redis parameter `repl_backlog_time_limit`. Once the threshold is exceeded, the backlog, which is the temporary log storing operations that may be missed by unreachable nodes, is discarded. If a new operation is made, the operation will not be stored on the backlog after it is discarded. When the node tries to re-establish the replication chain, the resync mechanism does not rollback the operations not stored in the backlog. S/L-DRM found this CFB.

The CFB can be reproduced by initializing Redis with three nodes: A, B and C replicating in a daisy-chain topology. Thus, B replicates from A, and C replicates from B while A is the primary. The schedule specifies the following steps: (1) break the replication chain between A and B—the chain between B and C is still maintained, wait for the delay exceeding `repl_backlog_time_limit` and then write to A—the write committed on A will not be stored in backlog as A has no slave long enough to exceed `repl_backlog_time_limit`; (2) have C replicate from B; and (3) have A replicate from B—A fails to truncate the operation made at step 1.

| Bug ID | Cumulative Time | Average Time |
|---|---|---|
| ZooKeeper Bug #2832 | 40760571 ms | 33192 ms |
| ZooKeeper Bug #2946 | 61922754 ms | 38920 ms |
| MongoDB Bug #1 | 1093842 ms | 364614 ms |
| MongoDB Bug #2 | 13836574 ms | 276731 ms |
| Redis Bug #2694 | 20883584 ms | 37425 ms |
| Redis Bug #4407 | 99338 ms | 33112 ms |
| Redis Bug #4316-1 | 640137 ms | 13619 ms |
| Redis Bug #4316-2 | 11073 ms | 5536 ms |

Table 4.6: Performance Measurements to Find Bugs

**Discussion.** Modulo for Redis has a significantly different design from ZooKeeper and MongoDB. In addition, it has a wide array of configuration options and needs to be manually configured to resynchronize after a failure. Because of the different behaviors under different configurations, Redis required the development of three different DRMs to find the CFBs. However, each DRM still found at least one CFB, illustrating the power of the DRM abstraction and the Modulo approach to finding CFBs.

In terms of time and effort, it took a novice user of Redis a couple of weeks to initially write the DRM interface functions. Moreover, the DRMs for Redis were more complex than those for ZooKeeper and MongoDB, so this represents at least an average—perhaps a slightly worse than average case for Modulo. Qualitatively, we feel the effort to construct DRMs is similar to that of writing a specification in a formal specification language such as TLA+, which has been cited as an acceptable cost by developers at Amazon [79]. The user needs to understand the important properties of the system they want to test and be able to abstract them away from implementation details. Also, the user must be able to reduce a system to a smaller number of nodes and a smaller number of keys to reduce the state space. However, unlike model checkers such as TLA+, which run on an abstract state machine representation of the SUT, Modulo combines the advantages of a reduced state space gained by the manual abstraction with the ability to reproduce the bugs found using a counter example of real inputs that can be run on the concrete system.

### 4.6.4 Testing Performance

Table 4.6 shows how much time our prototype took to find each bug mentioned above. The second column shows how long it took until finding bugs. The third column shows the average time taken to execute each schedule. The average time for MongoDB is significantly longer than the others because its DRM implementation of `waitForResync` confirms resync by comparing the timestamp of the latest write on the node with that of the master. However, MongoDB sometimes takes a long time to achieve convergence, so this timeout for

`waitForResync` must be set very high (10 minutes in our experiments) to avoid false positives. This long timeout contributes to a higher average transition time for MongoDB. The implementation of `waitForResync` for ZooKeeper and Redis can infer that resync has completed in other ways and does not depend on the timestamp returned by the database to be the same, allowing these DRMs to infer resync completion much more quickly. While the state space of our DRMs is small, in practice, Modulo is limited in how much of the state space it can explore by the rate at which it can explore schedules, which ultimately limits the number of CFBs that can be found. For future work, we plan to further optimize our DRM implementations and increase the parallelism in our concrete executor to increase the testing throughput of Modulo.

Modulo found most of the bugs within one or two days. If it does not find a bug even after a couple of days, we found that it is unlikely to find target bugs—we tried running it for up to a week. We think these numbers are the upper bound, because we found these bugs without any parallelization. We plan to build a parallelization facility for Modulo using well-known cluster-computing framework, such as Spark [108]. With this, Modulo will be able to find bugs much more quickly. Thus, Modulo is an effective systematic testing tool that is more targeted to find specific types of bugs. We think it would be a potential usage case that many DRMs can be written as test suites, which is more systematic than conventional unit testing or random testing.

## 4.7 Discussion

Here, we assume our target replicated distributed storage systems can run in a hostile environment where frequent failures can occur either due to benign reasons, such as node crashes, or malicious reasons, such as denial-of-service attacks. However, our failure model for this work does not consider Byzantine behaviors of the faulty node and instead only considers fail-stop failures.

# Chapter 5

# Detecting Consistency Attacks

## 5.1 Introduction

A commonly available type of cloud computing service is cloud storage services, which offer distributed storage systems over the Internet. Such services include basic object storage services such as Microsoft Azure Storage or Amazon S3, personal storage services such as Dropbox, Google Drive or Microsoft OneDrive, and database services such as Amazon RDS, DynamoDB and Microsoft Azure SQL Database. These services are popular because they offer users many useful features, such as backup and versioning for data, automatic scaling and failure recovery, replication and access to data across devices and the ability to collaboratively share data with other users. Industry figures indicate that there has been a rapid increase in the use of such cloud services. For instance, Google Drive currently has over 10 million users [25].

However, cloud services could potentially be a threat to the security of their users. While we believe it is unlikely that a cloud service provider would deliberately attack its customers, using a cloud service still exposes users to new threats. Cloud services provide services to multiple, often mutually distrustful, users on a shared infrastructure. Vulnerabilities in the infrastructure may allow a malicious user to compromise parts of the cloud service and attack another user. In addition, cloud services' employees, as part of their duties, may have privileged access to parts of the infrastructure or the software that implements the infrastructure. While good industry practice often means that no particular employee will have access to the entire infrastructure, there has been evidence that strategically placed insiders have been used by certain organizations to attack cloud users [71]. Thus, while a cloud service as a whole might not be malicious, the component that stores and serves user data can be compromised by both external and internal attackers, which can threaten the confidentiality and integrity of the data stored on the cloud service by the user. For brevity, we will henceforth refer to a cloud service whose data storage component has been compromised as simply a "malicious cloud service," even though not all components of the service may be compromised.

Since cloud storage services are implemented as globally replicated distributed storage systems designed to provide different devices or users access (even concurrent access) to the same data, there is an implicit or explicit assurance of consistent access according to some *consistency model*. A consistency model defines acceptable delays between when the results of an operation by one device become visible to other devices as well as the order in which those operations should become visible. Applications using the cloud storage service are implemented based on the consistency model. Therefore, applications will misbehave when the assumption on the consistency model is violated. By omitting, reordering, replaying and truncating operations, the malicious attacker can mount

subtle *consistency attacks* to cause violations of delay and ordering constraints of the consistency model. Such attacks can seriously damage applications composed of distributed processes collaboratively interacting with each other across devices via cloud storage services because the applications' behavior depends on the ordering and timing of the previous operations. For example, a source code repository such as a Git repository may suffer from truncated operations (*fork* attack [63, 72, 33]), which cannot be resolved by the built-in hash chain mechanism. In addition, an authorization service may inadvertently reveal sensitive information to unauthorized users when a revocation request is maliciously delayed or dropped after the read and update operations. Such an attack would allow revoked users to access data they should not have permission to access.

While there have been several recent proposals put forth to protect users from such attacks, they suffer from deficiencies in terms of security, battery-friendliness or timely detection. For example, a number of approaches use an external service such as email or instant messaging to enable devices to exchange messages about data they have stored on the cloud service [90, 33]. Unfortunately, while this defends against a malicious cloud service, it relies on the external service being trusted to deliver messages reliably. Other approaches eschew an external service in favor of having clients communicate directly with each other in a peer-to-peer fashion [55, 72], or they rely on a highly-available device to broadcast information to all devices [63, 64]. However, such an approach is not battery-friendly, as such devices must always be awake to communicate, and a peer-to-peer approach causes devices to consume more energy due to increased network traffic. An increasingly large percentage of user devices are battery-powered, so a negative impact on battery life is a serious drawback. Finally, some approaches, such as CloudProof [83], advocate infrequent intervals where logs of operations will be collected from all devices at an auditing service. Naturally, such infrequent intervals preclude timely detection.

Furthermore, all existing proposals can each only check a single consistency model, but cloud services offer a variety of consistency models. For example, some proposals only check strong consistency [83, 63, 64, 55], while others only check causal consistency [90, 72, 33]. None have been demonstrated to be general enough to be used to check several consistency models.

In this paper, we present Caelus, which overcomes these shortcomings. The key insight that enables Caelus to do this is having the cloud service declare the timing and order of operations on the cloud service. This relieves Caelus devices from having to record and send the timing and order of operations to each other. Instead, they just need to ensure that the timing and order of operations both conform to the cloud's promised consistency model and all devices perceive the timing and order of operations identically. To do this in a secure, battery-friendly and timely manner, Caelus employs several novel mechanisms.

First, Caelus detects inconsistencies in near real-time by having an *attestor* sign (i.e., attest) the order and timing of operations declared by the cloud service. These *scheduled attestations* are written back to the cloud storage service every few seconds according to a pre-determined schedule that is known to all devices. Other devices can then read these attestations from the cloud service and use them to verify the consistency of the operations they have performed without having to directly communicate with each other or the attestor, thus reducing the network usage and battery drain. In addition, because an attestation for an operation is available within seconds, devices can perform timely detection of consistency violations. Moreover, Caelus's protocol guarantees that a malicious cloud service cannot subvert Caelus by dropping or delaying attestations. Accordingly, the storage and distribution of attestations do not need to be performed by a trusted service.

Second, in its most basic instantiation, the attestor is a single device that is actively signing attestations every few seconds, but this would not be battery-friendly for the attestor. To reduce the impact on battery life, Caelus introduces *attestor-partitioning*, which partitions the attestor into a single *root attestor* device that must be periodically accessible but can otherwise sleep to conserve battery and an *active attestor* that must be active but whose

role should be assigned to a device already active for other reasons (i.e., it is writing data, or the user is already using it). Caelus ensures that, even though the attestor's role may be distributed across several devices that only communicate via the cloud service, a malicious cloud service cannot partition the devices into groups that are out-of-sync.

Finally, to enable the detection of violations for different consistency models, Caelus modularizes the task of verifying responses from the cloud service into a *consistency verification procedure* that can be performed independently by each device. This allows the same Caelus system to be used to check different consistency models. Therefore, Caelus can verify cloud storage services that provide strong, causal and eventual consistency models. In addition, the distribution of these checks across devices means that no single device will become a bottleneck as the number of devices increases.
We make the following contributions in this paper:

- We present the design of Caelus, which uses *scheduled attestations* for verifying the consistency of a cloud storage service. We describe our Caelus prototype that runs on Amazon's S3 storage service and demonstrate that we can detect consistency violations when we ask Caelus to bound inconsistency to a period shorter than what S3 can provide.

- We show that *attestor-partitioning* can reduce the battery drain of Caelus on the attestor by about 40× without reducing the security of the system. Our measurements show that Caelus increases CPU utilization on clients by about 12.6% and imposes a network bandwidth overhead of about 1.3%. Under normal, failure-free scenarios with an honest server, the user should experience no perceptible overhead or loss of availability as a result of Caelus.

- We provide three *consistency verification procedures* that enable individual devices to use Caelus to verify strong, eventual and causal consistency using a series of logical checks over a signed log of operations.

We start with a couple of potential but realistic consistency violation problems in Section 5.2. Then, we discuss Caelus's security model and assumptions as well as the guarantees it provides in Section 5.3. The various design aspects of Caelus are discussed in Section 5.4. Then, we present an analysis of Caelus's security properties in Section 5.5. Section 5.6 describes the implementation details of our prototype, and Section 5.7 provides evaluation results of our Caelus prototype. Additionally, the measurement study on the connectivity of mobile phones and additional discussion about Caelus design are presented in Section 5.8.

## 5.2 Motivating Scenarios

To demonstrate the seriousness of consistency attacks, we describe two common scenarios in which a malicious cloud service can subvert victim software systems.

**Git Repository.** In the first example, consider a user or group of users that use an online Git repository hosted in the cloud, such as Github[1]. Git repositories should be strongly consistent. Individual commits do not need to be totally ordered, but a set of commits pushed to the Git service should be visible to every client as soon as the push is completed, thus forming a total order of pushes and enabling consistent conflict resolution. Git uses hashes to verify the integrity of the commit history. The hash of each commit depends on the parent commit, thus forming a hash chain. Moreover, each Git client keeps a complete history of all commits by all others. Thus, it might

[1] https://github.com/

initially appear that the Git's hash chain generated by the Git service and local copies of the commit history stored by each client prevent the misbehavior by the Git service.

However, by subverting the timing requirement that all operations should be made visible to all other requirements as soon as the operation completes so some commits are *never* made visible, a malicious Git server can corrupt the source code repository in a way that is undetectable by individual clients. Consequently, a malicious service can hide a particular push and all future pushes by one or more clients from a different set of one or more clients. This effectively partitions or forks the group of clients into two or more sets that are unaware of each other's pushes, but they also cannot tell whether the other group has not committed anything or whether the server is maliciously dropping commits without directly communicating with each other or using a trusted service to communicate.

**Authentication Service.** A security-sensitive operation in federated identity and authorization services such as OAuth is credential revocation. OAuth is used in a variety of online services such as Google and Facebook to authorize untrusted parties ("relying parties" in OAuth parlance) to access information belonging to the user. In OAuth 2.0, revocation requests should be processed and propagated to all servers immediately so that the revocation takes effect as soon as possible [69]. If an OAuth implementation uses a cloud storage service, it will depend on the cloud service providing consistent updates of revocation requests to all application servers. For example, consider a user that revokes access to a document to an untrusted party before adding sensitive information to it. Since the revocation happens before the addition of sensitive information, a cloud service that promises strong or causal consistency should ensure that all servers see the revocation request before the addition of the sensitive information; that is, in the same order. However, a malicious cloud service may break this promise and replicate the operations in an inconsistent order. Consequently, some nodes may receive the revocation request after the addition of the sensitive information and reveal the sensitive information to the revoked parties, even though this contradicts the user's expectations.

## 5.3 Security Model and Guarantees

Caelus is designed for any user who owns multiple Internet devices, some of which are battery-powered devices, such as tablets, smartphones or laptops, that may have wireless network connections that can fail. While Caelus can support non-battery-powered devices that have reliable network connections, Caelus uses special mechanisms to mitigate power consumption and network failures. We envision that many devices of the future, such as smart-home devices, smart-cars and Internet-of-things devices, will also have wireless network connections and be battery-powered. We begin by explaining our security model, which describes our assumptions regarding the behavior and capabilities of clients, the network and the cloud service. We then state the security guarantees that Caelus provides.

### 5.3.1 Security model

**Clients.** Clients are devices that are under the user's control and are used to access data stored on the cloud service. For example, a client can be the individual user's laptop, tablet, or other battery-powered device. A client can also be a device owned by other users with whom the primary user has shared access to the cloud service. We assume clients can become malicious or unavailable for a variety of reasons. They can become malicious due to infection by malware, compromise or theft. Likewise, they can become unavailable due to software failure, loss of network connectivity, loss of battery power or system sleep to conserve power. In cases where impending

unavailability is known beforehand, such as a system sleep or battery depletion, the client can warn the other devices, allowing them to take actions to mitigate the effect of the unavailability.

We assume that each client has a public-private key pair that can be used for digital signatures and that the public keys of clients are known to other clients and the cloud provider. To protect against man-in-the-middle attacks, we assume either public keys are distributed using a protected channel or a PKI exists to certify their authenticity. Each user also has an encryption key used to encrypt the user's data to protect it from disclosure to the cloud service. We also assume a secure key distribution mechanism for the shared encryption key, so it is only shared with the user's clients and the clients with whom the user is sharing data.

We assume that clients have reasonably synchronized clocks. The degree of clock synchronization required depends on the accuracy at which the user wants to detect a malicious cloud server. While previous work has shown that very highly synchronized clocks are possible [27], for storage with personal data, we believe that limiting clock skews to several milliseconds, which can be achieved by using NTP, should be sufficient.

**Network.** We assume a network model that provides connectivity between each client and the cloud service but does not provide direct connectivity between clients. In addition, the network may fail to transmit messages between a client and the cloud service, and clients cannot distinguish between a failure of the network and a failure of the cloud service. Assuming that all communication between clients must traverse the cloud service enables clients to communicate, even if both are not online simultaneously [31].

Furthermore, we assume that an upper bound exists for the end-to-end delay on the network path. Specifically, we assume an upper bound on the end-to-end delay between when a client issues its request and when a service actually processes the request. Because requests will go through a network medium and intermediate nodes before arriving at the destination on the cloud service, the network path will involve variable delays, which makes it extremely difficult to enforce an upper bound on the end-to-end delay. Implementing a fully deterministic network is therefore currently infeasible in practice. Nevertheless, related technologies and standards, such as Time-Sensitive Networking [3] and Deterministic Networking [4], have recently emerged. In the near future, we envision that these new technologies and standards will become increasingly prevalent for implementing a fully deterministic network. More importantly, even if this assumption does not hold, Caelus still provides guarantees described below with the increased chance of having false-positives; that is, incorrectly flagging benign cases as malicious.

**Cloud Service.** The cloud service promises a certain consistency model for data stored on the cloud service. An honest cloud service will respond to requests for data from various clients according to the promised model. Caelus further assumes that cloud services offer a time bound on consistency models, which means that operations are guaranteed to become visible to all clients within some *visibility time bound*, which is specified by the cloud service provider in SLA. In practice, consistency models that are not bounded are less useful because it is very hard to reason about the data when developing client software. Furthermore, unbounded consistency models can result in unresolvable conflicts. Accordingly, recent work has shown that, in practice, most systems that claim to be weakly consistent are still bounded [15]. In fact, there are a number of proposals in the literature that enable users to measure the time bound a cloud service offers [42, 18, 12, 84, 104, 82, 17]. Thus, bounded consistency models are realistic, and we believe cloud service providers may even be motivated to claim a shorter time bound than their competitors.

In our security model, a malicious cloud service's goal is to violate the promised consistency model and trick the user into unknowingly using inconsistent data or, alternatively, to claim a consistency model stronger than what they offer and hope that the user does not notice the discrepancy. A malicious cloud service can selectively omit, replay, reorder or delay the results of clients' operations. In addition, since all client communication goes

through the cloud service, it can also selectively fake client failures by preventing operations made by a client from becoming visible to one or more other clients. However, we assume that standard cryptographic assumptions hold—a malicious cloud provider cannot decrypt data for which it does not know the key. Nor can it forge cryptographic signatures. In other words, we use the Dolev-Yao attack model for a malicious cloud service.

Like clients, we assume that the cloud service has a public-private key pair and that the public key is well-known to all clients. Thus, a cloud service's response signed with the cloud's private key is non-repudiable.

**Collusion.** As we will discuss in Section 5.3.2, Caelus makes security guarantees against both a malicious cloud service and malicious clients. Caelus assumes that malicious clients can collude and defends against them. However, if the clients and the cloud provider are both malicious and collude, it would be difficult to make any guarantees since there are no non-malicious components left in the system. Consequently, we weaken the security model slightly by assuming that clients are *cloud-secure*, meaning that they can be compromised and act maliciously, but are always secure against compromise by the cloud service. For example, the cloud-secure assumption holds if clients are infected with malware or have been stolen, so long as that malware or the thief is not under the control of the cloud service.

We believe this assumption is realistic for several reasons. First, many cloud services provide APIs for developers to develop their own client software [89]. For example, a plethora of third party Dropbox clients exist that enable users to automatically backup their files, synchronize data or use multiple backup services [102]. Accordingly, the provenance of the client software is largely independent of the cloud service provider.

Second, in cases where the user is using a client provided by the cloud provider, there can still be independence if the client software and cloud storage service are hosted on separate systems. Thus, an attacker who compromises the cloud storage service does not automatically get the ability to corrupt or control the client software.

### 5.3.2 Security guarantees

We now state the security guarantees that Caelus provides. Because our security model allows for both a malicious cloud service and malicious clients, we separately describe the guarantees that hold against each of these.

Caelus provides the following security guarantees against a malicious cloud service:

**SRV1:** A malicious cloud provider cannot read user data.

**SRV2:** A malicious cloud provider cannot tamper with user data without being detected.

**SRV3:** A malicious cloud provider that responds with inconsistent data will be detected within a finite time bound defined by $T_{Caelus}$.

Against malicious clients, Caelus provides a different set of guarantees. Since devices have the ability to read and modify data, Caelus cannot protect the confidentiality or integrity of data on the cloud against a malicious device. This could be somewhat mitigated by access control, but the amount of protection would still be dependent on the access control policy, so it cannot provide complete protection for data confidentiality and integrity. We thus leave the integration of access controls into Caelus for future work.

However, since all operations must be signed, Caelus does guarantee that operations by clients are non-repudiable. In addition, a malicious client may attempt to falsely accuse the cloud provider of violating consistency guarantees. Caelus guarantees that such false accusations can be invalidated using an audit procedure. In summary, Caelus provides the following guarantees against malicious clients (including multiple colluding clients):

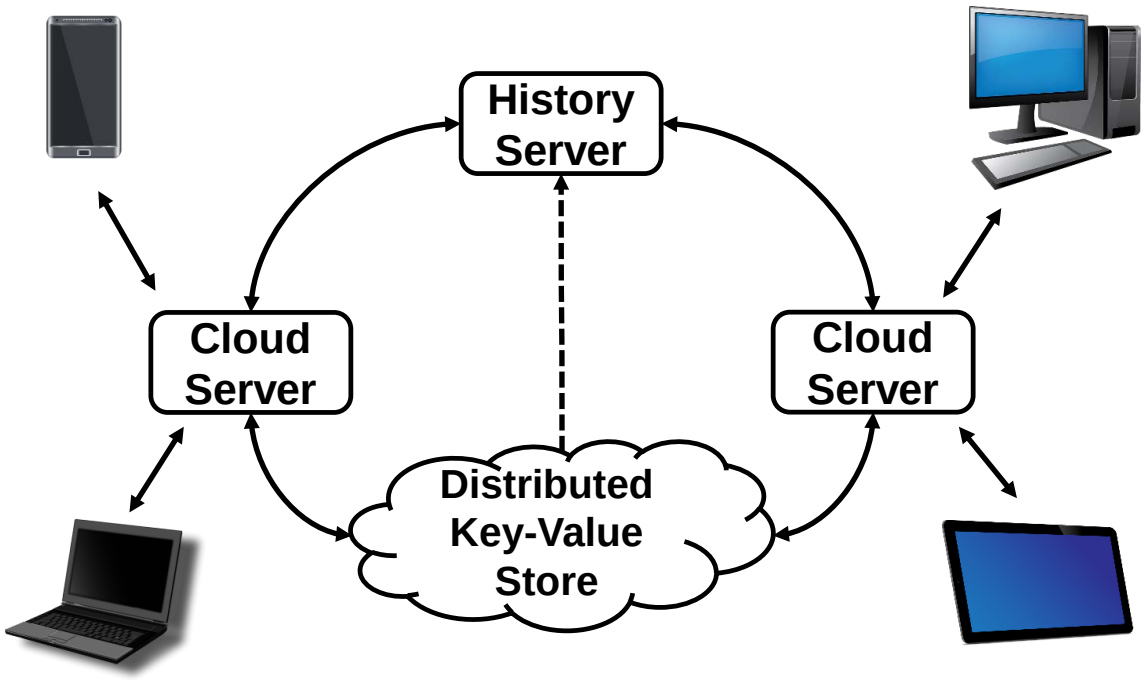


Figure 5.1: Caelus Architecture.

**CLT1:** Malicious clients cannot repudiate modifications they have made to data on the cloud.

**CLT2:** Malicious clients cannot falsely accuse the cloud service of violating the promised consistency model.

Caelus does not protect against the loss of data due to a malicious cloud provider. A malicious cloud provider can always drop a user's request or destroy data after receiving it. Similarly, a malicious client can overwrite data or refuse to perform its duties, thus affecting the durability and availability of user data. However, in the absence of benign failures or malicious activity, Caelus provides the following guarantee:

**AV1:** Under normal operations where clients and the cloud service are free of failures and malicious activity, Caelus will not cause delays or unavailability of the cloud service.

Since Guarantee AV1 does not hold if there are malicious parties, it should be clear that it is more of a performance guarantee than a security guarantee. However, we believe this guarantee is still important since to be practical, Caelus's security guarantees should impose little or no cost under normal (non-malicious) circumstances.

## 5.4 Design

### 5.4.1 System overview

Caelus consists of four components: *clients*, *cloud servers*, a *history server* and a *distributed key-value store (DKVS)*, as illustrated in Figure 5.1. Clients are deployed and run on customer devices. Cloud servers, the history server and the DKVS are deployed and run on cloud infrastructure to provide a cloud service serving a distributed storage system. Clients connect with cloud servers that are the closest ones in terms of latency. Clients perform operations on the DKVS by sending corresponding requests to connected cloud servers (e.g., clients can perform `Get` or `Put` to read or write the value of a specific key). In addition, Caelus clients are responsible for verifying

their own operations. Cloud servers are geographically distributed to provide shorter latency for clients. Also, cloud servers appropriately forward client requests to the DKVS and the history server. In addition, cloud servers send back the results of operations to clients. The DKVS manages data as collections of key-value pairs where a `Put` entirely updates the value for the specific key rather than partially updates the portion of the value. The DKVS replicates data across multiple nodes that are geographically distributed. Also, the DKVS consistently maintains data replicas according to the consistency models the DKVS provides. In Caelus, *strong consistency*, *bounded eventual consistency* and *bounded causal consistency* are provided—they will be defined and discussed in more detail later. The history server maintains the *global history log*, which is the log containing the *global history* of client operations. The global history is the history of all operations previously performed by every client. Also, the history server is responsible for storing control messages that are sent between clients, such as attestations and selection, which will also be discussed in more detail later.

Clients read and write data from the cloud service with `Get` and `Put` operations. Caelus relies on the Network Time Protocol (NTP) for clock synchronization among client devices. In order to perform `Get` and `Put` operations, clients should establish a connection with the closest cloud servers and send requests to connected cloud servers. Before clients send requests, they obtain timestamps from their own clock and embed those in their requests. More specifically, timestamps are stored in the field of the request header. The format of the request header will be discussed in more detail in the next section. Timestamps are also used by the DKVS to determine the ordering of client operations.

Each client must store and manage the *local history* in its own storage device. A local history must contain information such as values a client's `Get`s read, operations that are not verified yet, the last log segment the client received from the history server and the time when operations initially occurred.

Among clients, we have a special client called the *attestor* that periodically generates *attestations*, which are signed hashes of the log segment. The attestor periodically performs the following procedure: (1) obtaining a log segment from the history server via a cloud server; (2) generating an attestation by signing the hash of the log segment; and (3) uploading the attestation to the history server via a cloud server. More details on the format of attestations and the aforementioned procedure will be discussed in more detail in the following section.

Deploying Caelus clients to customer devices can be accomplished by having users install client software equipped with the Caelus verification scheme. The security of their data is one of the main concerns that cloud users have. As a result, cloud service providers may be motivated to deploy Caelus to convince users that their data are safe and remove legal liability from themselves as Caelus guarantees hold even if the cloud infrastructure is compromised.

Cloud servers forward `Get` or `Put` requests from clients to a DKVS. When the DKVS returns, cloud servers check the result of those requests. If the request was successfully handled by the DKVS without any error, cloud servers send those requests to the history server so that requested client operations can be logged in the global history log. If the request was failed, cloud servers should relay back the error message to the client without having the history server log the failed operation. For attestation-related or control messages, cloud servers directly forward requests to the history server.

A distributed key-value store must honour embedded timestamps in client requests by committing operations in the order dictated by those timestamps—we will discuss the reason for doing this in more detail in Section 5.8. There can be several ways to achieve this. For example, the DKVS may wait up to the upper bound of an end-to-end delay before committing requests. This gives the DKVS the ability to serialize client operations as strictly as it needs with respect to the real-time order in which operations are issued.

The history server logs all operations that have occurred on the DKVS. `Get` requests may be logged as soon

as they arrive on the history server. However, the key-value store is globally distributed. Therefore, there is a delay between the time when the distributed key-value store starts handling the request and the time when the distributed key-value store fully replicates the `Put` to every replica so that the `Put` becomes visible to all clients. Thus, `Puts` are only logged by the history server when it has been notified by the key-value store that they have been made globally visible. If the key-value store is not capable of such notifications, the history server logs the `Puts` after the visibility time bound has passed since requests are sent to the DKVS—this is an approximation and the networking and processing delay involved should be considered. The order that operations are logged on the history server is unimportant, as clients rely on embedded timestamps in requests to reconstruct the order of operations.

Although the history server is under the control of the cloud service and not trusted by the user, it plays a crucial role in Caelus. Instead of having client devices assemble a view of all operations that have taken place, the history server on the cloud assembles all operations as the history of all operations, enables the cloud service to declare the history of all operations that has been faithfully performed on the DKVS and has the declared history highly available for clients. Then, all the clients should do is to verify that 1) the declared history conforms to the promised consistency model and 2) all the operations they have performed are reflected in the log. As the history of operations are assembled and declared by the cloud service, clients' consistency verification procedure can be considerably simplified. Also, the verification procedure can be distributed across clients. Furthermore, clients do not have to be awake to communicate and exchange the history of operations. The main guarantee that Caelus must provide is then that all clients perceive the same declared history from the cloud service that can be achieved by having an attestor discussed in the following section.

We begin by describing a *basic system* that uses a single monolithic attestor and provides all the security guarantees described in Section 5.3.2. However, the basic protocol is not battery-friendly, so we then describe an enhanced *battery-friendly system* that uses attestor-partitioning to enable devices that are not being actively used to sleep and conserve energy. Finally, we will discuss some operating parameters of Caelus.

### 5.4.2 Basic system

We describe our basic system in six steps. First, we describe how `Put` and `Get` operations are implemented by the cloud service. Second, we describe how the logging of operations is implemented. Third, we describe the attestation procedure Caelus uses to ensure that every client has an identical view of the history of operations. Fourth, we describe how each client verifies that its local view of operations is consistent with the attested history of operations. Fifth, we describe how clients join and leave the Caelus system. Finally, we describe how the audit procedure works. The major elements of the Caelus protocol are illustrated in Figure 5.2—we will explain more details of each element below.

**Operations.** As depicted in Figure 5.3, each `Get` and `Put` operation transmits the following meta-data in the header of the request message: operation type (`Get` or `Put`), key, client ID, a timestamp, a sequence number and a hash of the data if it is for a `Put` operation. The entire header is signed with a private key specific to each client, and whose matching public key is already made publicly available. After the operation is performed on the DKVS, cloud servers reply to clients with the response message that contains the field telling if the operation has succeeded or failed (i.e., Ack or Nack), the hash of the entire request message header's meta-data and optionally the hash of the data if it was a response to the request for a `Get` operation. The response header is signed with a private key specific to the cloud service, and whose matching public key is already publicly known.

Each field in the meta-data of request and response headers contributes to achieving security guarantees that

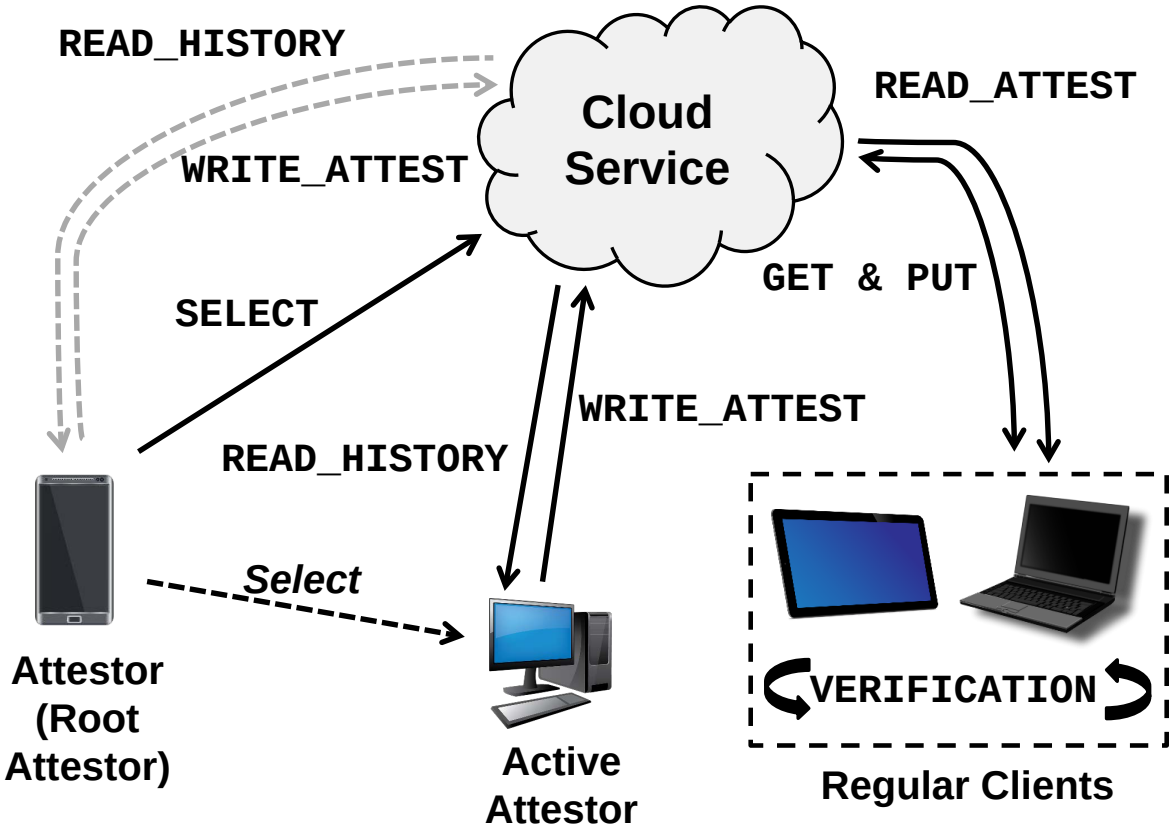


Figure 5.2: The Caelus Protocol. The Root Attestor selects the Active Attestor using the `Select` operation. The Active Attestor then reads the history using `Read_History` and signs the history to produce attestations, which are written back using `Write_Attest`. When no Active Attestor is available, the Root Attestor can perform attestation itself (shown in gray). Regular clients perform `Get` and `Put` operations and verify these operations by reading attestations using `Read_Attest` and running the verification procedure on the attested histories.

are goals of Caelus. How each field contributes to each security guarantee is summarized in Table 5.1 [2]. Sequence numbers are used to detect omission, replay and reordering of operations by checking if sequence numbers are continuously and monotonically increasing. Timestamps allow clients to detect truncation or delay by checking if operations are attested within the expected time bound. Thus, with sequence numbers and timestamps, a malicious cloud cannot covertly manipulate the history of operations applied to data and, therefore, cannot indirectly tamper with user data. Hashes are used to protect from a malicious cloud's attempt to directly tamper with stored data instead of tampering with the operation history. Hence, both sequence numbers and hashes can enforce Guarantee SRV2. Hashes may be tampered or forged by a malicious cloud; therefore, digital signatures are employed to provide the integrity of operation meta-data. Because a malicious cloud cannot forge digital signatures without knowing the private keys of clients, signatures can prevent operation forging and operation tampering. Moreover, signatures defend against repudiation attempts; therefore, neither the cloud nor clients can repudiate. Thus, Guarantee CLT1 is enforced by signatures. Finally, any data transmitted in a `Put` is encrypted by clients; therefore, it

[2] Operation truncation is different from operation omission because operation truncation means omitting all future operations between two or more specific partitions of clients while operation omission means omitting some operations but not all future operations and there is no permanent partition between two or more groups of clients

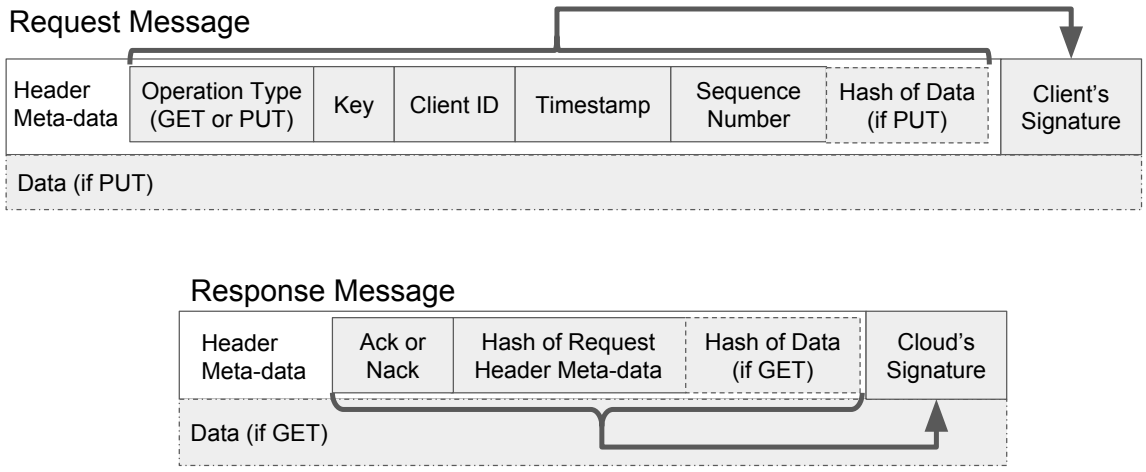


Figure 5.3: Operation Request Message and Response Message Formats. To make an operation, clients send a request message, and the cloud service replies with a response message. The format of request and response messages for `Get` and `Put` operations are illustrated.

enforces Guarantee SRV1. Guarantee SRV3 is achieved by a consistency verification check, and Guarantee CLT2 can be achieved by the audit procedure as we will discuss later in separate sections.

Figure 5.4 illustrates how client operations are passed from a client to the history server: (1) a client constructs a request message and fills in fields accordingly [3]; (2) the client sends the request message to the closest cloud server; (3) the cloud server receives the request and prepares it to forward to the DKVS and the history server; (4) the cloud server sends the request to the DKVS; (5) the DKVS processes the request while honouring [4] client timestamps; (6) the DKVS then replies to the cloud server with the result of processing the requested operation; (7) the cloud server checks the result of processing the client request on the DKVS; (8) if the result is successful, the cloud server sends the request to the history server to log it [5]; and (9) the history server gets the latest global sequence number, increments by one, prepends it to the signed request header meta-data to construct an *operation record* and appends the operation record into the *global history log*.

Cloud servers do not buffer any data; their main purpose is to provide a single interface to their clients and hide the details of the DKVS and the history server from clients. Clients' `Put` and `Get` requests are directly forwarded to the globally distributed DKVS, and the results are sent back to clients by cloud servers.

**Logging.** `Gets` and `Puts` are forwarded to the history server after a cloud server receives replies from the DKVS if operations were successfully committed. `Gets` are logged immediately, while `Puts` are only logged after the DKVS notifies a history server that the results of `Puts` have become globally visible. Alternatively, the history server can log `Puts` by itself after the visibility time bound has passed since the DKVS commits them.

[3] A client obtains the timestamp from its local physical clock and attaches the timestamp to the request message at the proper field in the request header meta-data.

[4] Honouring here means that the DKVS performs operations in the order reflected in clients' timestamps. For a more detailed discussion, refer to Section 5.8

[5] If the request is for a `Put`, the cloud server sends only the request header embedding the hash of the data and the client's signature but not the actual data

| Threat | Countermeasure | Security Guarantee |
|---|---|---|
| Operation Omission | Sequence Number | SRV2 |
| Operation Replay | Sequence Number | SRV2 |
| Operation Reordering | Sequence Number | SRV2 |
| Operation Truncation | timestamp | SRV2 |
| Operation Delaying | timestamp | SRV2 |
| Data Tampering | Hash | SRV2 |
| Operation Forging | Signature | SRV2 |
| Operation Tampering | Signature | SRV2 |
| Repudiation | Signature | CLT1 |
| Unauthorized Data Read | Data Encryption | SRV1 |

Table 5.1: Threats and Countermeasures Enabled by Caelus Operation Request and Response Formats

Having the history server log only those `Puts` that are fully visible to every client allows us to check if `Puts` are replicated within the time bound and, therefore, if subsequent `Gets` consistently read fresh values. Because fully visible `Puts` are required to be logged, a malicious server cannot omit, delay longer than the time bound or truncate.

The history server assigns global sequence numbers to logged operations, which are only used as a way for clients to request sections of the log. `Puts` that have been received but are not yet logged are not assigned global sequence numbers—such `Puts` are not yet visible to every client.

While the history server is shown as a single machine in Figure 5.1, it does not have to be and can be distributed. However, if distributed, one important caveat is that each operation must be assigned a unique global sequence number, so the history server must be at least sequentially consistent. This requirement only holds for keys that have common clients. If two sets of keys do not have any clients in common, the assignment of sequence numbers among those sets does not need to be sequentially consistent. Violating this requirement will result in clients detecting consistency violations, as it will result in operations with duplicate sequence numbers. Thus, it is of no benefit for a malicious cloud service to violate this requirement. In addition, the history server only stores hashes of data objects instead of the full data objects, so the amount of data stored is relatively small.

The Caelus's scalability can be degraded due to this centralized history server. The sequential unique global sequence number requirement only applies to each collection of keys shared by common clients. Thus, the careful distribution of different sets of keys with common clients can help with the scalability problem. In addition, for the large keyspaces, careful partitioning can also help to prevent a small number of nodes of the distributed history server from becoming a bottleneck. Moreover, storing just a hash of data instead of the entire data for each request further improves the scalability, as data transmission and processing get faster due to the smaller size. To further optimize, the history server may run slower than the DKVS by batching operations and logging at a slower pace as long as no time bound requirement is violated. However, the history server in Caelus cannot be as scalable as the eventually consistent distributed systems and this the limitation is left for future work.

The global history log on the history server is intended to act as "proof" that the cloud service is adhering to its promised consistency model. For example, suppose there are three clients: client 1, client 2 and client 3. Client 1 performs `Put(X,1)` initially. Then, client 2 performs `Put(X,2)`. Following that, client 3 performs `Get(X,2)`. Each client operation is logged by each client as a local history. From client 3's perspective, it

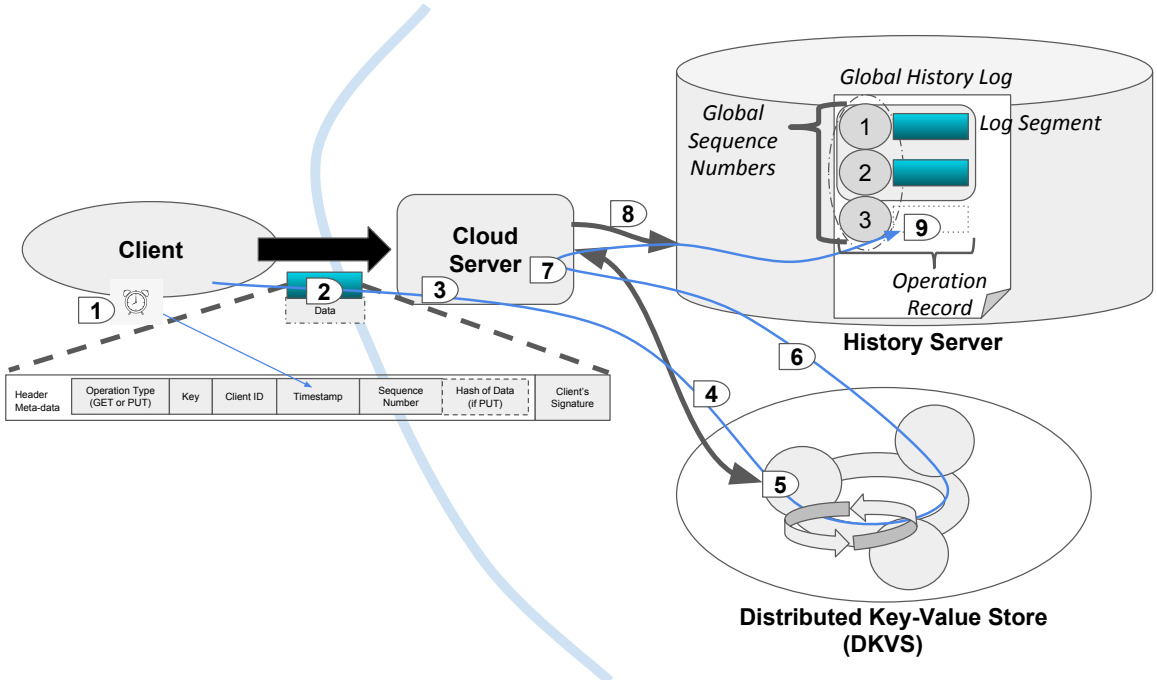


Figure 5.4: Operation Path from a Client to the History Server. Under normal circumstances with no errors, a request message is sent by a client to the cloud server, then to the distributed key-value store, and finally to the history server.

needs to know what other clients perform in order to check if the cloud service is adhering to its promised consistency model (e.g., strong consistency). The global history log on the history server will show the log containing `Put(X,1)`, `Put(X,2)` and `Get(X,2)`. Client 3 can identify that its own operation logged in its local history appears in the global history log and, referring to the global history log, it can see that its `Get` operation saw the result of the latest `Put` operation, which had wrote 2 to the key X. Therefore, client 3 can conclude that the cloud service correctly provided strong consistency for its operation. Also, clients can compare the global history log and their own local history to verify that their own operations appear in the global history log. If the global history log is inconsistent with any of the local history of a client, the client whose local history is inconsistent with the global history log can raise an alarm. For instance, if the global history log contains only `Put(X,1)` and `Get(X,2)`, then client 2 can detect that its operation is not properly reflected in the global history log. At the same time, client 3 can detect that its operation read the value that has never been written.

Thus, correctly constructing and serving the global history log is critical. In a distributed setting, assembling all information and providing them to every client can be challenging. Especially, if the target environment includes clients that are battery-powered devices, devices frequently and spontaneously come and go due to battery and network connectivity issues. Having clients directly exchange their local histories to construct the full global history log may not work without disruption if some clients become unavailable unexpectedly. Also, even if clients successfully constructed the full global history log, placing the global history log on clients soon to be disconnected will cause another availability problem for clients who want to access it. By having a highly available cloud service to assemble and serve a global history log, Caelus can ease this challenge and lift a significant amount of burden from client devices.

Depending on the consistency model, cloud servers and the DKVS may also include additional information about each operation into the global history log to facilitate the verification that the consistency model is met. We

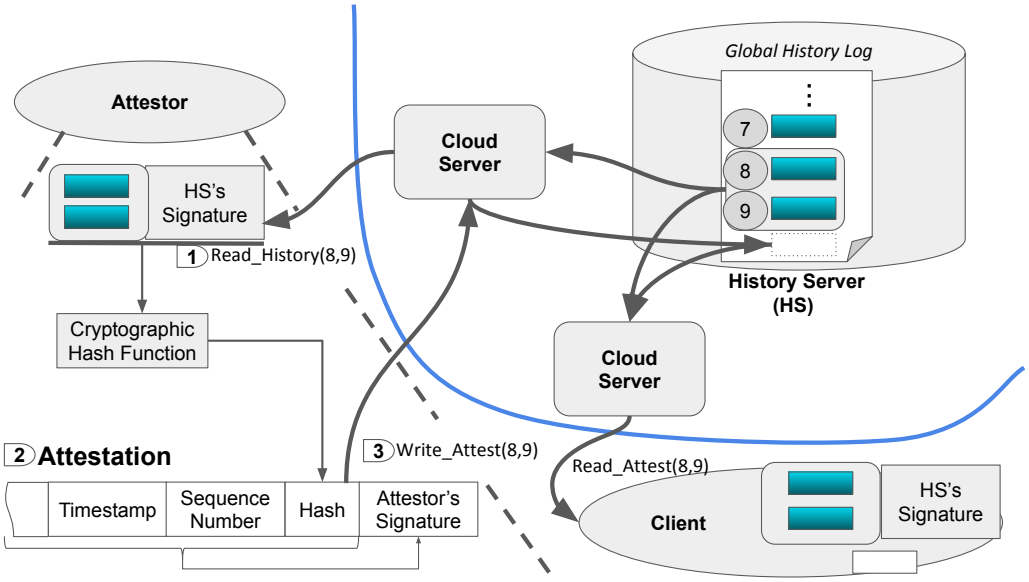


Figure 5.5: Attestation Procedure. The attestor, which is a special client, is responsible for performing attestation periodically. It fetches the log segment, computes the hash of it, attaches a sequence number and a timestamp and signs the hash along with the sequence number and the timestamp to generate an attestation. Lastly, the generated attestation is uploaded to the cloud service and logged into the global history log.

discuss the details of the consistency verification procedures below.

**Attestation.** One of the user's devices, acting as the attestor, periodically performs an *attestation procedure (attestation)*. For now, we assume that the attestor has no battery limitation, which we will remove by employing *attestor-partitioning*, the protocol described in Section 5.4.3. There are two requirements that the attestor must meet. First, the role of the attestor is permanently assigned to one and only one device, and the identity of the attestor should be known to all other devices. Second, the attestor should periodically perform the attestation procedure on a schedule that is also known to all devices.

The attestor performs attestation for a specific log segment as described by Figure 5.5. The attestor starts the attestation procedure by fetching a log segment from the history server. Then, the attestor generates an attestation using the fetched log segment. Firstly, the attestor computes the cryptographic hash value of the fetched log segment. The fetched log segment contains operation records, while global sequence numbers are stripped off. Moreover, the log segment is signed by the history server to ensure that a malicious client cannot tamper with them, which enforces Guarantee CLT2. Secondly, the attestor prepares the operation request message as described in Figure 5.3, and we extend it with adding the attestation as a new type of operation [6]. Like `Get` and `Put` operations, the attestor fills out a sequence number and a timestamp in the meta-data of the request message. Also, unlike `Put` operations, the attestor inserts the computed hash value of the log segment (including the

[6]Similarly, `Select`, `Status` and `Wake`, which we will describe later, can be added as the new type of operations

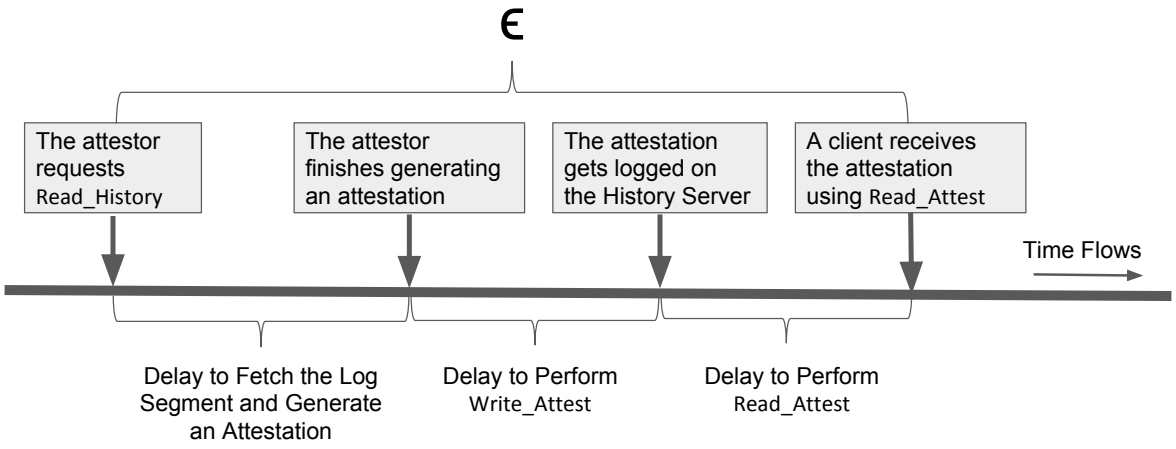


Figure 5.6: Networking and Processing Delay for Reading a New Attestation. Various delay factors involved in fetching a log segment, generating a new attestation, uploading the new attestation and reading the new attestation. Such delays must be considered whenever a client is measuring the delay between two distributed events, and we denote it by $\epsilon$.

history server's signature) into the hash field instead of the hash of data. Then, the attestor signs the entire header including the hash value of the fetched log segment, the sequence number and the timestamp with its private key; therefore, the malicious cloud cannot tamper, forge, omit, reorder, replay, delay or truncate attestations. The result of this procedure is also called an *attestation*. Thirdly, the attestor finally uploads the generated attestation to a cloud server. The cloud server forwards the attestation to the history server. Then, the history server stores the attestation in the global history log after attaching the easily-retrievable index based on global sequence numbers of attested operation records. Then, clients can download one or more attestations along with the corresponding log segment in order to check for the availability of the attestor and verify consistency of their own operations as we will explain in more detail below.

To request a log segment, the attestor uses a `Read_History`$(G_{Start}, G_{End})$ operation, which specifies a section of the log between two global sequence numbers $G_{Start}$ and $G_{End}$ to read. The attestor stores an attestation back to the history server by using a `Write_Attest`$(G_{Start}, G_{End})$ operation. Clients can read attestations from the history server by using a `Read_Attest`$(G_{Start}, G_{End})$ operation, which returns all operations and attestations in the requested range. The attestor performs attestation at every specific time interval defined by the parameter $T_A$.

Clients expect to be able to read a new attestation every $T_A + \epsilon$. $\epsilon$ accounts for variable delays due to network and processing and must be added any time a client is measuring the delay between two events distributed across the cloud service. $\epsilon$ can be broken down to several factors, and we provide one example of breaking it down in Figure 5.6. Figure 5.6 is a breakdown of $\epsilon$ for reading a new attestation. As $\epsilon$ is a variable delay, for different contexts, there can be different ways of breaking it down. For instance, if we are interested in how fast a `Put` operation will become visible after the client's `Put` request, then we can say it will take up to $T_S + \epsilon$ for the result

of the `Put` to become fully visible for every client, where $\epsilon$ will account the end-to-end round-trip delay between the client and the distributed key-value store.

Note that, as the attestation procedure is periodically performed on a schedule expected by every client, we call this attestation scheme *scheduled attestation*. This scheduled attestation prevents a malicious service from showing different log contents to separate partitions of clients. As explained above in Table 5.1, replay, omission, reordering, delay and truncation can be detected by sequence numbers and timestamps. Note that a malicious service may attempt to truncate all future log segments and attestations, but this will be detected because clients will not be able to read an attestation at the expected time. Clients cannot distinguish between this type of malicious cloud service and a failed attestor, but since the attestor is available most of the time and assumed to only experience failures for short periods of time, Caelus clients halt until they are able to read any missed attestations. If a client continues to miss attestations for an extended period of time, it can notify the user, who can then examine the state of their attestor device to determine if the device or the device's network connection has failed or not. If neither the device nor its network has actually failed, this indicates that the cloud service is acting maliciously.

Using scheduled attestation, all clients can safely assume that all attestations will be eventually made identically visible to all clients. By extension, this guarantees that all clients will see the same history of operations and, from this, detect if the cloud service is maliciously trying to intentionally violate consistency models using the verification procedure we describe next.

**Verification.** To distribute the verification tasks, each client is responsible for verifying the consistency of its own operations. Verification happens asynchronously to `Put` and `Get` operations when clients periodically fetch attestations using the `Read_Attest` operation. Caelus verifies that operations are inconsistent by, at most, some time bound $T_{Caelus}$, thus enforcing Guarantee SRV3. Verifying the consistency of `Put`s means that clients check if their `Put` operations are globally visible and attested within the time bound $T_{Caelus}$. Verifying the consistency of `Get`s means checking if `Get` operations actually read valid values under the specific consistency model given the global history of operations within the time bound $T_{Caelus}$.

Clients verify their operations in three steps as illustrated by Figure 5.7. First, clients verify the correctness of the fetched log segment against the accompanying attestation. The sequence number of attestations should be checked whenever a new attestation is fetched to make sure there is no reordering, omission or replaying of attestations. Timestamps are also checked to make sure the attestation is actually performed at the expected schedule. The first step ensures the log segment fetched along with the accompanying attestation is valid to use. Second, clients perform a *presence check*, where they verify the individual signatures on each operation in the log to detect tampering. Once the integrity of each operation is checked, operations should be sorted based on timestamps in an increasing order. Then, we check if operations are not reordered, omitted or replayed using the sequence numbers embedded in the operations. Timestamps are used for the next step which is the consistency model specific check in order to compare the timing of two operations. This second step is to ensure each client's operation is attested on time and its meta-data is valid to use. Finally, clients verify the consistency of their `Put` and `Get` operations. The exact method that clients use to verify the consistency of `Put`s and `Get`s depends on the consistency model of the cloud service. We will discuss the third step in more detail below.

Caelus currently supports three consistency models: strong consistency, bounded eventual consistency and bounded causal consistency with some time bound defined by the *visibility time bound* $T_S$. Strong consistency is defined as every `Get` should see the result of the latest `Put`. Bounded eventual consistency is defined as every `Get` may see the result of either the latest `Put` or of any `Put` that is written but still being replicated. Bounded causal consistency is defined as every `Get` should see the result of the latest `Put` according to the causal order.

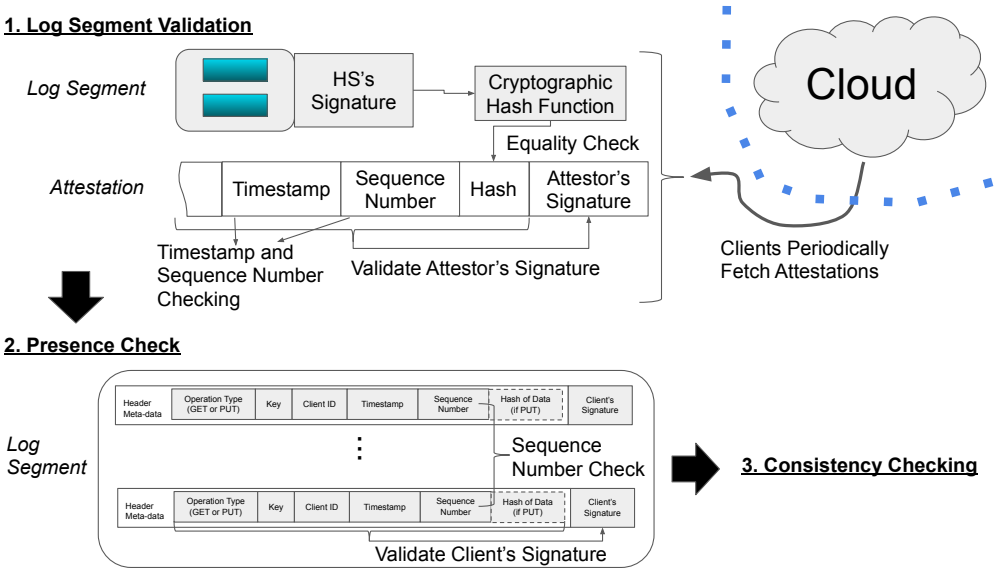


Figure 5.7: Caelus Consistency Verification Procedure Overview.

For the definition of all three consistency models Caelus supports, we additionally include the requirement that `Puts` become globally visible within $T_S + \epsilon$.

Under strong consistency, all operations must appear to execute in a single global order. Then, for the given global order, every `Get` must receive the value of the immediately preceding `Put` to the same key. In addition, all `Puts` should be globally visible as soon as they are acknowledged. This makes the verification of strong consistency the simplest of all three models. Clients verify the consistency of `Puts` by checking that the `Put` appears in the next attestation signed by the attestor. This means that a cloud service can, at most, delay the effects of a `Put` by $T_A + \epsilon$. Clients verify the consistency of `Gets` by checking that the value returned matches the value of the immediately preceding `Put`.

In the bounded eventual consistency model, the results of `Puts` do not need to be immediately visible to all clients but may instead take up to the visibility time bound $T_S$ to become visible to all clients. This is equivalent to the definition of the bounded read guarantee used by Pileus [98]. The checks that clients do to verify the aforementioned bounded eventual consistency are illustrated in Figure 5.8. To verify the consistency of `Puts`, the client will check that (1) the attestation time of all of its `Puts` are, at most, $T_S + T_A + \epsilon + \delta$ after the corresponding `Put` has been acknowledged by the cloud service, where $\epsilon$ accounts for the end-to-end delay taken until the client received the attestation for its `Puts` and $\delta$ accounts for clock skew between clients when comparing timestamps.

Under bounded eventual consistency, checking the consistency of `Gets` is slightly more complex than under strong consistency. Fundamentally, for each unverified `Get` to be verified without a violation, the matching `Put` performed before the `Get` should be found. To find all `Puts` performed before the `Get`, clients will collect log

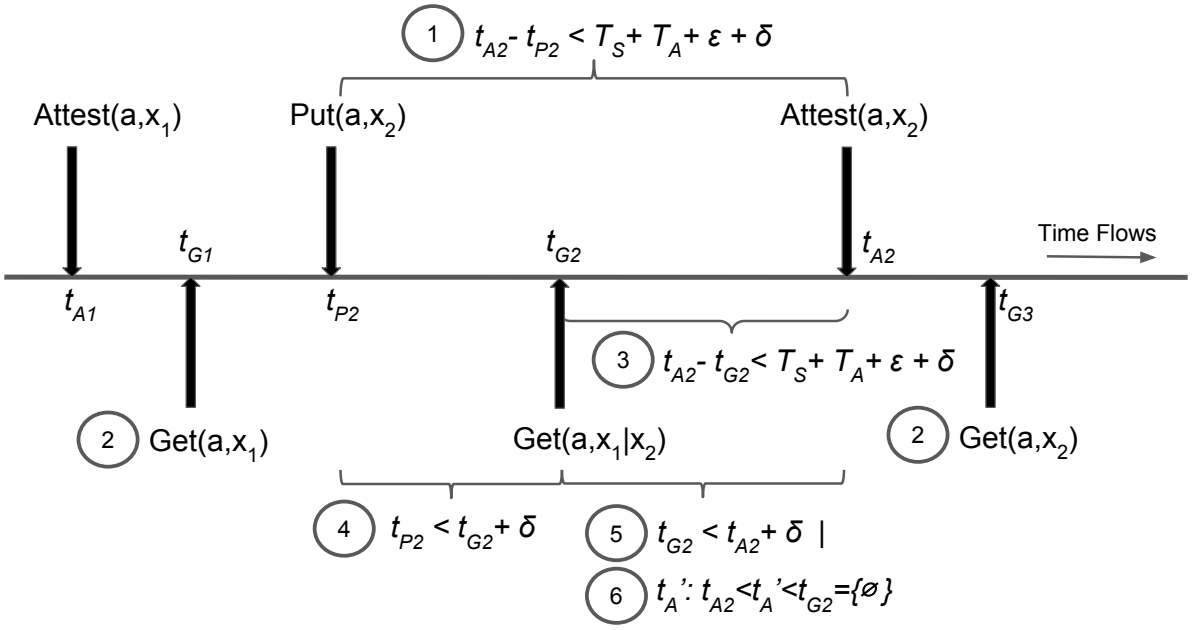


Figure 5.8: Verification of Eventual Consistency. We denote operations as $operation(key, value)$. $x_1|x_2$ means that the `Get` may legally return either $x_1$ or $x_2$.

segments and attestations until $T_S + T_A + \epsilon + \delta$ since the `Get`. Within this time duration, all `Puts` performed before the `Get` will be fully replicated and attested. Then, it ensures that there is no newer `Put` than the matching `Put` that the unverified `Get` has read.

Figure 5.8 illustrates a set of checks for bounded eventual consistency. At time $t_{A1}$, the attestor attests a `Put` with value $x_1$. Then, at time $t_{P2}$, a client performs a `Put` with value $x_2$ to the same key. Because of the consistency model, $x_2$ is not attested until time $t_{A2}$. The `Get` at $t_{G1}$ must return $x_1$ because $x_1$ has been attested (i.e., globally visible) and the latest value to read, while the `Get` at $t_{G3}$ must return $x_2$ for the same reason. However, the `Get` at $t_{G2}$ may return either $x_1$ or $x_2$. Thus, to verify the consistency of a `Get`, ② the client first checks whether the value returned by the `Get` matches the most recent attested value. If not, it is either a violation or the `Get` has read the value of a `Put` that has yet to be attested. The client maintains a list of unverified `Gets` and ③ waits for an attestation for a matching `Put` to appear within the *timeout period* $(t_{A2} - t_{G2}) < T_S + T_A + \epsilon + \delta$ [7]. If an attestation for a `Put` does appear in time, then the client checks that ④ the timestamp of the `Get` is later than the timestamp of the `Put`, i.e. $t_{G2} > t_{P2} - \delta$. It must also check that ⑤ the `Get` is before the `Put`'s attestation, i.e. $t_{G2} < t_{A2} + \delta$. If not (i.e., the `Get` is after the `Put`'s attestation), then the client must instead check that ⑥ there was no newer attested `Put` that the `Get` should have read, i.e. $t'_A : t_{A2} < t'_A < t_{G2} = \{\emptyset\}$. ⑥ handles the case where the attestation happens before the `Get`, but is not fetched and verified by the client until after the `Get`. If the `Get` passes these checks, then it is verified and removed from the unverified `Get` list. Otherwise, the `Get` remains on the list and will be checked against other attested `Puts` until either it is verified or it times out and becomes a violation.

A cloud service that implements bounded causal consistency for Caelus enforces causal consistency on the values read by `Gets`, and will eventually make all `Puts` visible to all clients via the global history log. Bounded

[7] Note that within this time, the client can use the value of the `Get` as any violation will be detected within the timeout period. If the attestation does not appear before the timeout period, either the cloud service is taking too long to replicate results or the service returned stale results, both of which are consistency violations.

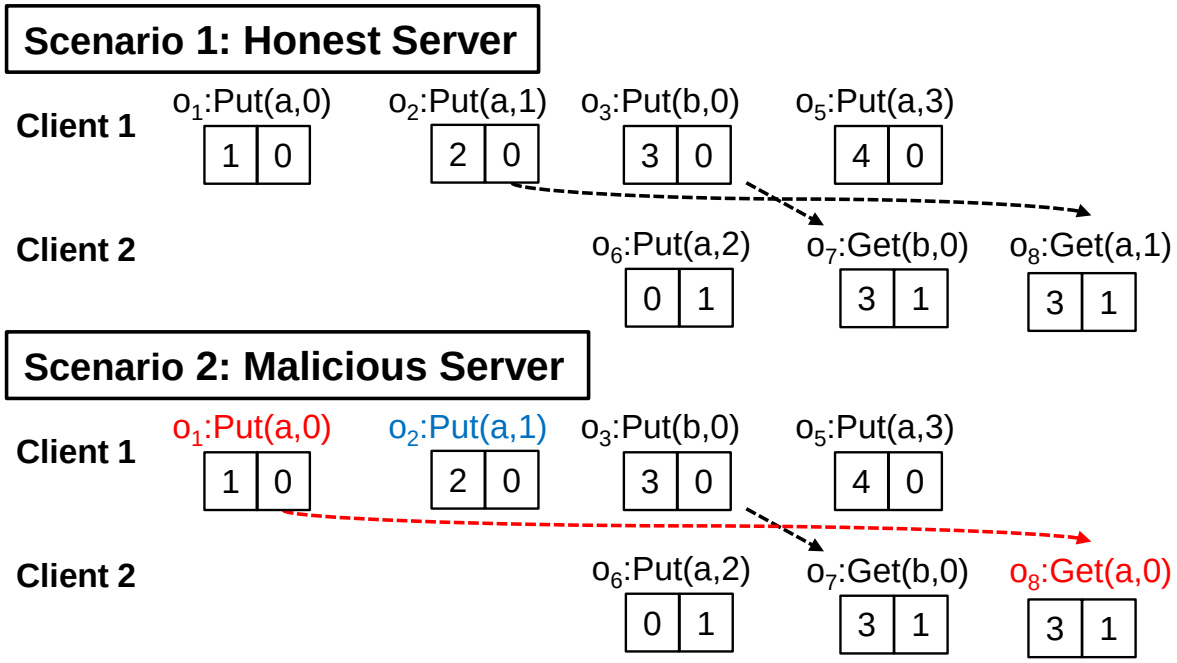


Figure 5.9: Verification of Causal Consistency. The vector clock is shown below each operation as [C1 | C2].

causal consistency is also referred to as Causal+ consistency [67] in the literature. Because `Puts` must be made globally visible in a bounded amount of time, verifying the consistency of `Puts` in bounded causal consistency is the same as verifying `Puts` in bounded eventual consistency.

As with eventual consistency, some `Gets` may see the result of `Put` operations before they become globally visible in the global history log. Thus, clients perform the same verification steps in causal consistency as eventual consistency. However, while a `Get` in eventual consistency may return either the most recent attested value or *any* written but unattested value, `Gets` in causal consistency must return the most recent value on which it is causally dependent. One option for verifying `Gets` would be for clients to extract the chain of causal operations it is dependent on and then verify that the value read matches that of the most recent `Put` in the chain. However, if the client only knows the value that the `Get` read, it may be ambiguous as to which `Put` it is actually dependent on if there are several `Puts` with the same value.

To uniquely identify each operation, we enhance the cloud servers to attach a vector clock to each operation in the log [59]. Clients verify the correctness of the vector clocks by checking that they increase along with the sequence numbers on operations, which indicate the program order. Clients can then use the vector clocks to verify the freshness of the value read by checking if there are any newer `Puts` to the same key between the vector clock of the `Get` and its associated `Put`. Like in eventual consistency, a client may have to defer verification for up to $T_S + T_A + \epsilon + \delta$ to ensure all necessary attestations have occurred.

To illustrate, consider Figure 5.9. The `Get`, $o_8$, by client $C2$ reads the result of the `Put`, $o_2$, by $C1$. We denote a vector clock of an operation using the notation $vc(o_i)$. Note that vector clocks only increase on `Put` operations. $C2$ verifies the consistency of the `Get` by verifying that there is no `Put` operation on the same key with a vector clock greater than $o_2$ and less than $o_8$. In Scenario 1, there is no violation because all operations between $o_2$ and $o_8$ modify other keys. $o_1$, $o_5$ and $o_6$ modify the same key, $a$, but since $vc(o_1) < vc(o_2)$, $vc(o_5) \parallel vc(o_8)$ and $vc(o_6) \parallel vc(o_2)$ ($\parallel$ means "incomparable", the two values have no defined order), their results may legitimately

be invisible to $o_8$. However, in Scenario 2, the cloud service, either maliciously or erroneously, returns the result of $o_1$ instead of $o_2$ to $o_8$. In this case, client verification will fail because it finds that $vc(o_1) < vc(o_2) < vc(o_8)$ and that $o_2$ is a `Put` to the same key read by $o_8$. A malicious cloud service cannot assign $o_2$ a vector clock less than $vc(o_1)$ because the order of the vector clocks must match the sequence number embedded in the operations. Neither can it omit $o_2$ since the presence check done by clients will detect that the $o_2$ operation is missing from the log.

For a large number of clients, vector clocks can be expensive since the length of the vector is determined by the number of nodes in the system [67]. However, in Caelus, vector clocks do not need to span users who do not share data. Instead, the size of the vector only has to accommodate the number of clients a user has (or is sharing data with), which we expect to be generally fewer than 20.

**Client join and leave.** When a client joins or rejoins the system after a period of being asleep, it must verify that the attestor is available before performing any operations. It does this by checking that the timestamp of the most recent attestation posted by the attestor is less than $T_A + \delta$ old. Once it establishes that the attestor is available and making attestations properly, it can proceed to access values on the cloud service.

If the client has been disconnected from the cloud service for a long period of time, it may have to download a significant portion of the log to verify the consistency of `Gets` that read values written many operations ago. To bound the length of the log, the attestor can periodically checkpoint the entire key-value store by performing a `Get` and `Put` on every key, attesting to the new key values and having the history server discard all log entries before the checkpoint. To safely checkpoint a key, Caelus must ensure that there are no `Puts` in flight so that the latest value is checkpointed. If large key-values are anticipated, Caelus can provide special `Checkpoint` operations that avoid transmitting the value since the value itself does not change. Checkpointing requires all keys with conflicting values to be resolved, though data loss can, of course, be avoided by assigning conflicting values to new keys.

When clients intentionally leave, for example, to go to sleep, they may have to delay their leaves by up to $T_S + T_A + \epsilon + \delta$ so that they can verify any operations made just prior to sleeping. Note that $T_S$ is effectively zero in strong consistency since replication is immediate. Unfortunately, if clients have an unexpected failure that they cannot delay, a malicious server can truncate data written in the last $T_S + T_A + \epsilon + \delta$. However, a malicious server cannot omit operations since omissions will be caught by the presence checks done by other clients.

**Audits.** If a consistency violation is detected, either the cloud service acted maliciously or a device incorrectly reported a consistency violation. To differentiate between these two cases and enforce Guarantee CLT2, an audit procedure is required to verify that the device is truthfully reporting misbehavior by the cloud service. The audit procedure is fairly straightforward, as all the information required to perform the verification procedure is contained in the logs and attestations on the history server, and thus no information or interaction is required with the device that is accusing the cloud server. Thus, the user can perform the audit procedure by repeating the verification procedure on a device whose integrity is known to be good. We envision the requirement to perform such audits to be rare, so it is reasonable to assume the user will be willing to expend some effort to acquire such a device. For example, they may boot a device from a CD or USB image that is known to be safe or use a device capable of verified trusted-boot [73, 75]. The audit procedure can even be performed publicly if all signature verification keys are available in case the user wants to prove to a third party that the cloud service behaved maliciously.

### 5.4.3 Battery-friendly system

Our basic system described above is secure but not battery-friendly because it requires the attestor to produce attestations continuously. To solve this problem, we introduce *attestor-partitioning*, which partitions the single attestor into a *root attestor (RA)* and an *active attestor (AA)*, each fulfilling one of the requirements on the single attestor. The device that fulfills the RA role takes on that role permanently, and its identity as RA is known to all devices. However, once it selects a device to take on the role of the AA, it can sleep and conserve battery, and the selected AA will then actively create attestations every $T_A$. Consequently, the role of the AA is not permanently attached to one device but can be changed as necessary to minimize the impact on battery life. In Section 5.7, we show that Caelus has minimal battery impact on a device that is already awake. The main battery cost of Caelus results from preventing devices from sleeping. Thus, the RA should select a device to be the AA that must be awake for other reasons. For example, the RA could select devices that the user is actively using or devices that are awake due to their processing background tasks, such as downloading updates or synchronizing data. The RA can even select itself as the AA if it is the only device that is awake. If all devices are asleep, then no AA needs to be selected because no operations are being performed. Thus, there is nothing to attest. If operations are being performed on the cloud service, then at least one device must always be awake to perform them.

While the RA can be any device, we generally envision that the user may use their smartphone as the RA for their devices. As of Jan 2020, there are approximately 3.5 billion smartphone users worldwide [16, 94]. Accordingly, even in instances where users only own one or a small number of devices, we can likely assume that at least one of them is a smartphone. Smartphones also have several other advantages that make them suitable for use as an RA. First, they have a cellular data connection, meaning they are likely to be reachable and able to respond to network requests. Second, users generally keep their smartphones with them, so they are more able to fix a failed or disconnected smartphone than a non-portable device. Finally, a malicious cloud provider could drop messages from the RA to make it appear that it has failed. However, since the user usually has the smartphone-RA with them, such an attack is unlikely to work, as the user can easily verify the state of the smartphone.

The key security invariant that attestor-partitioning protocol must uphold is that there must not be more than one AA at any given time. Otherwise, a malicious cloud service can fork the AAs into two partitions that are not aware of each other, which would violate all the consistency models that Caelus tries to guarantee. Although the attestations are performed by the AA, the RA still has an important role in keeping security guarantees by maintaining only one AA at any given time and securely handling instances where the AA fails unexpectedly.

The RA selects the AA by writing a *selection message* to the history server using a `Select` operation[8]. Subsequently, the AA will perform attestations every $T_A$, allowing the RA to sleep. Selection messages are signed and include a sequence number just like regular operations and thus cannot be forged or replayed. They contain a timestamp and unambiguously select the client to be the AA. If the AA leaves or fails, the RA must then select another client to be the AA. Thus, while an AA is active, the RA must wake up every $T_R$, where $T_R \gg T_A$, to check for the presence of AA attestations in the log. If a current attestation exists, the AA is still present, and the RA renews the selection by writing a new selection message. This renewal message is important, as it serves to tell the AA and clients that the AA has not been isolated from the RA. If the AA does not see a renewal at $T_R + \epsilon$ after the last renewal, it must stop acting as the AA and wait until a new AA is selected.

When a client wants to join the cloud service, it checks for the presence of an AA by ascertaining if the last selection message is less than $T_R + \delta$ old. If selection has expired, there is no current AA, and the client must wake the RA. To do this, we enable clients to wake the RA by adding a `Wake` operation that causes the cloud service to

[8]The operation request message format can be extended to support new operations like `Select` and then the new operation will be protected against the malicious cloud in a same way `Get` or `Put` operations were protected as we described earlier.

wake the RA using a *push notification*. Push notifications are a facility that is universally available in essentially all battery-powered devices, such as phones and tablets, that allows a remote host to send a message to a mobile device, such as an RA device, and ensures it is received in a timely manner, even if the device is sleeping. They utilize special hardware that puts the main processor on the device to sleep while the network interface remains awake but allows the network interface to wake the device if a message arrives. A variety of push notification services exist, such as Google Cloud Messaging, the Apple Push Notification Service and the Amazon Simple Notification Service. Before waking the RA, the client indicates that it is awake by writing a status message to the history server using a `Status` operation. Like all other operations, status messages include a sequence number and are signed so they cannot be forged or replayed. Then, the cloud service wakes the RA, which checks the status messages on the history server to see which devices are awake. It next selects an active device to be AA and goes to sleep for $T_R$.

If the AA intends to leave, it must give up its role as the AA. Similar to the join procedure, the AA writes a status message indicating it is going to leave and asks the server to wake the RA. At that point, the RA can select a different device if there are other devices awake or go to sleep if there are no other devices awake.

### 5.4.4 Handling failures

One of the drawbacks of attestor-partitioning is that it can increase the likelihood of unavailability because if the AA fails, the system will become unavailable for up to $T_R + \epsilon$ for the RA to wake up, at which time the RA will detect that the AA is not making attestations. Recall that $T_R$ could be on the order of several minutes. If other clients are awake, the RA will select a new AA; otherwise, it will go to sleep. At first, it might appear that clients could avoid having to wait by waking the RA once they detect that the AA has failed (i.e., after $T_A + \epsilon$ has passed without an attestation). However, this is not safe, as neither the RA nor clients can differentiate between a failed AA and a malicious cloud service that is dropping AA attestations. If the RA incorrectly assumes the AA has failed and selects a new AA when, in fact, the cloud service is dropping attestations, this will result in two simultaneous AAs. Without a trusted communication channel between clients, the only way to avoid this is for the RA and all clients to wait until $T_R$ has passed. Then, even if the cloud service is malicious, the AA will stop acting as an AA unless it sees a selection renewal from the RA, which the RA will not issue unless it can see the operations of the AA.

An AA can potentially suffer from a variety of failures, such as benign failures due to WiFi disconnection, battery depletion or failed hardware as well as malicious failures such as malware infection or remote compromise. Such a failed AA affects the availability of Caelus as mentioned above. Moreover, if the failure is malicious, a compromised AA (or any other compromised client) could falsely accuse the cloud provider, requiring an audit to be run, which will impact the availability of Caelus as well. Thus, Caelus should try to minimize the chance of AA failure as much as possible. Preventing attackers from compromising devices is beyond the scope of this work.

The easiest way to reduce benign AA failures is to have more reliable devices and networks. Enterprise-grade devices, while more expensive than consumer-grade devices, are often of a higher quality, providing more reliable networks and less failure-prone hardware. Thus, while Caelus is not restricted to only enterprise settings, it will perform better in such settings where the probability of failures is low. In lieu of using higher-cost devices, Caelus can still mitigate failures by managing and configuring the system more carefully. For instance, when more than one candidate for AA is available, the RA can preferentially select an AA that is more reliable by using attributes such as the device's previous failure history, the current battery-level of the device, the network signal strength and error-rate of the device, the software patch level and whether the device is in the physical presence of the user

so failures can be more quickly addressed. We leave the details of an algorithm that correctly balances these and possibly other attributes for future work.

A malicious push notification service could arbitrarily delay, drop or forge notifications. Delayed or dropped notifications will reduce the responsiveness of the system, since the RA will not wake up as intended. Alternatively, forged notifications will cause the RA to wake up unnecessarily, affecting the battery life. Both of these attacks reduce availability, thus affecting Guarantee AV1, which is not intended to withstand malicious activity. However, all other security guarantees hold against a malicious notification service. Moreover, both of these attacks are not stealthy and can easily be detected, for example by having clients detect delayed or dropped notifications by timing the delay for the RA to respond after a `Wake` or having the RA detect forged or delayed notifications by checking for a valid `Status` operation upon receiving a notification.

Guarantee AV1 ensures that Caelus does not increase unavailability unless the cloud service or devices fail, or there is malicious activity. The only time Caelus will stop clients from performing operations is if a scheduled attestation or selection message is missed, which can occur only when either the AA or RA fail, the network they are connected to fails or if a malicious server drops those messages.

### 5.4.5 Operating parameters

Caelus has several time-based operating parameters, some of which are dictated by the cloud service or operating environment, and some of which are set by the user. $T_S$ is the visibility time bound for the cloud service and is a property of the distributed key-value store. We expect environmental parameters $\epsilon$, which represents network and processing delay, to be on the order of 10s to 100s of milliseconds (for connectivity over cellular networks) and $\delta$, which represents clock skew between devices, to be a few milliseconds. Since the history server should be composed of a single machine or a set of tightly coupled machines, we expect log, attestation and select operations to take on the order of $\epsilon$.

Caelus guarantees that clients cannot unknowingly use data that is inconsistent by more than $T_{Caelus}$, where $T_{Caelus} = T_S + T_A + \epsilon + \delta$. Caelus may also detect some operations that violate the shorter $T_S$ bound, but its ability to do this is limited by how short $T_A$ is. This is because operations logged by the history server are only attested every $T_A$, so an operation that violates $T_S$ by some amount $\phi$ will evade detection if the time it waits on the history server for the next attestation cycle is less than $T_A - \phi$. Thus, a short $T_A$ increases Caelus's ability to detect violations, but at a slightly higher network and computational cost to the AA and clients who must process attestations more often. Note that to have $T_A = 0$, which implies $T_{Caelus} = T_S + \epsilon + \delta$, an AA that checks infinitely often would be required, or a trusted history server would be needed to implement the AA.

While an RA that is never unavailable can never cause the system to be unavailable, real devices do become unavailable, so they can cause system unavailability. Attestor-partitioning mitigates the effects of RA unavailability by allowing the AA to hide some of the times the RA is unavailable. While the value of $T_R$ does not affect the system security, a longer $T_R$ reduces the likelihood that temporary unavailability of the RA will affect the unavailability of the entire system. To illustrate, consider an RA running on a smartphone with an availability of 97.81%, as found by our informal measurement study detailed in Section 5.8. Modelling the phone as a random variable with an expected value of 97.81% and subject to a trial every $T_R$, it would take about $32 \times T_R$ before the probability that phone unavailability impacts Caelus's availability increases to 50%. Considering an average period of unavailability of approximately 94 seconds (again from the study), partitioning gives the user a 50% chance of experiencing roughly 7 minutes of unavailability every 24 hours with a $T_R$ of 10 minutes, which compares favorably to the expected 30 minutes of unavailability the same phone is expected to experience every 24 hours. Moreover, the expected unavailability decreases exponentially as the phone's availability increases.

Therefore, we think the smartphone RA can be reliable and highly available for our target environment.

In general, a longer $T_R$ will also improve the RA's battery life, as the device the RA is on can spend a longer proportion of time sleeping. Mobile push notification services typically require the device to wake and send keep-alive messages every five to ten minutes. Consequently, we generally expect that $T_R$ is set to coincide with these keep-alive periods.

## 5.5 Security Analysis

Now that we have presented the design of Caelus, we describe how the individual guarantees that Caelus provides are upheld by elements in its design. These guarantees hold when either the cloud service is malicious or clients are malicious. We further show that, even if several malicious clients collude, they do not gain any abilities beyond those of a single malicious client.

### 5.5.1 Analysis of guarantees

**SRV1.** Since all clients encrypt data before sending them to the cloud provider and the encryption key is not known to the cloud service, a malicious cloud provider cannot read user data.

**SRV2.** A cryptographic hash of data sent in `Puts` is computed and included in the header of the `Put`. The header is signed by the device making the update. The same header is returned by the cloud service to a client performing a `Get` on the same key. The client then verifies the signature on the header and then uses the hash to verify the integrity of the returned data. Since the cloud service cannot forge the signatures, the data stored on the cloud are protected from tampering by a malicious cloud provider.

**SRV3.** The scheduled attestations produced by the AA in combination with the consistency model-specific verification checks the guarantee that consistency violations are detected within $T_{Caelus}$. Scheduled attestations ensure that all clients are notified of the history of an operation within $T_{Caelus}$ after the operation occurs. Since all clients see the same history, one can view the attested history as a "global history" of all operations. The verification checks then guarantees two properties: 1) that each client's observed history of operations matches the attested global history and 2) that the global history is consistent with the promised consistency model. For the second property, the verification checks verify that all operations are made visible within the promised time bound by comparing the timestamps on operations (i.e., no stale data are read) and that operations are made visible according to the ordering constraints specified by the promised consistency model (i.e., no malicious reordering).

**CLT1.** Because all key-value updates must be signed by the client making them, a client cannot later deny that it made the update. Consequently, data modifications are non-repudiable.

**CLT2.** To falsely accuse the cloud service of a consistency violation, a client must show that one of the verification checks has failed, even when it has not in reality. This can only happen in one of two ways: either the accusing client can alter the contents of the attested history so a check fails, or it can convince the user that a verification check has failed, even when it has not.

All attested history segments are signed twice: once initially by the cloud service and again by the attestor. Accordingly, a regular client would have to forge the signatures of both the cloud service and the attestor to tamper with the attested history. If either the RA or the AA is malicious, it could try to tamper with the attested history before signing it. However, to successfully tamper with it, the malicious attestor would still need to forge the

cloud service's signature, which is impossible according to our attack model. As a result, no malicious client can tamper with the attested history to falsely accuse an honest cloud service.

The other alternative is for the client to incorrectly evaluate a verification check and declare that a check has failed. However, since all consistency violations can be publicly audited, a malicious client on its own cannot falsely accuse the cloud service.

**AV1.** Clients expect the AA to sign an attestation every $T_A$. If this does not happen, clients will halt, affecting the system's availability. Similarly, a malicious RA can refuse to select AA and also refuse to sign attestations. If the cloud service is malicious, it can affect availability by simply refusing to respond to requests for clients. Thus, a single malicious client, if it happens to be the AA or RA, or a malicious cloud service can invalidate Guarantee AV1.

Under normal circumstances, where there are no failures and all components adhere to the protocol, an attestation is produced every $T_A$. If all clients are asleep and a new client joins, it must wake the RA. Normally, the new client would have to wait for up to $T_R$ for the RA to wake, thus affecting availability. However, because Caelus uses push notifications, this waiting period is shortened to latency of a push notification, which is on the order of one second. Thus, under normal circumstances, Caelus does not affect availability.

### 5.5.2 Colluding clients

Multiple colluding clients do not have any capabilities beyond a single malicious client. Like a single client, they are only able to corrupt or leak data by virtue of their ability to access and modify the data. However, both guarantees against malicious clients, CLT1 and CLT2, hold.

**CLT1.** Malicious clients could share their signing keys, thus allowing any malicious client to forge signatures that could have been made by another malicious client. Thus, malicious actions can no longer be traced to the client that made them but only to a member of the group of colluding malicious clients. While this changes the actual terms of Guarantee CLT1, it does not change the intent. Actions made by an adversary in control of several clients are still traceable back to that group of clients. Thus, Guarantee CLT1 still holds in the face of client collusion.

**CLT2.** To falsely accuse the cloud service, clients must be able to forge cloud service signatures. Having more than one malicious client does not make it any more possible to forge signatures, so Guarantee CLT2 will also hold against colluding clients.

In summary, colluding clients, regardless of whether they are regular clients or include the AA or RA, do not invalidate any guarantees except Guarantee AV1, which can already be invalidated by a single malicious client if that client happens to be an AA or RA.

## 5.6 Implementation

In this section, we describe our Caelus prototype, which implements the cloud server and history server components in the cloud service and clients for PCs and Android devices.

### 5.6.1 Cloud service

Cloud server and history server components are implemented in 3K lines of Java. Communication between the server components and clients is implemented using Apache XML-RPC. To implement different consistency models, our prototype is modular and can use distinct key-value store backends. For strong consistency, our prototype

uses a single cloud server with a local key-value store implemented with the LevelDB library [39]. Because all clients communicate with a single server, the server can make all client operations atomic, thus providing strong consistency. To implement eventual consistency, our prototype uses Amazon's cloud infrastructure. Multiple cloud servers run as EC2 instances and use Amazon's S3 service as the key-value store backend. A single history server typically shares one of the EC2 nodes with a cloud server but can also run on a dedicated node. We are currently not aware of any open-source or commercial cloud service that implements a causally consistency key-value store. Consequently, our prototype does not implement causal consistency at this time. However, if one were available, we believe it would be fairly straight forward to integrate it with our prototype, as our prototype only assumes a `Put` and `Get` interface.

It is possible to implement Caelus without cloud server components by having clients communicate directly with the key-value store and history server. However, having the cloud server allowed us to easily abstract the different LevelDB and S3 interfaces from the clients, letting us have an identical client code for all experiments.

### 5.6.2 Clients

We implement two types of clients for our prototype, one for PCs and one for Android devices. Both are written in Java and consist of about 7K lines of code. To reduce the number of client-server round trips, the server piggybacks recent attestations to the responses for `Put` and `Get` requests.

Each time a client performs a `Put`, it is enqueued on a *deferred verification list*. Occasionally, a `Get` can be verified when it occurs because it reads the latest attested value, but other times, it reads a value that has yet to be attested, so it must also be enqueued on the deferred verification list. The verification of deferred operations is performed asynchronously by a *verification thread*, which periodically wakes up every $T_A$, processes any new log segments that have been received and then verifies operations on the deferred verification list. Operations that remain unverified for longer than $T_{Caelus}$ are flagged as violations. Any delays between when the AA posts attestations and when clients process them must be accounted for in $\epsilon$. Thus, we synchronize both the period and phase of the verification thread with that of the AA.

We use the Google Cloud Messaging (GCM) service to implement push messages on Android clients. GCM generally takes about one second to deliver a message to the phone because it requires an additional network hop to Google's servers. This latency could be reduced by implementing our own dedicated push service and collocating the notification server with the cloud server the phone is connected to, but for the purposes of our prototype, we found one-second latency to be reasonable and perhaps more realistic since most cloud services would be more likely to use a third-party notification service than implement their own. GCM does not use a fixed period for keep-alive messages but varies their timing depending on network conditions. Caelus can be modified to allow for a variable $T_R$ by having the phone embed the length of the current AA selection period in each selection message, but our prototype does not implement this. Hence, we currently do not synchronize $T_R$ with the GCM heartbeat period.

## 5.7 Evaluation

We evaluate four properties of our Caelus prototype. First, we evaluate Caelus's effectiveness at detecting consistency violations. Second, we evaluate the computational costs for Caelus clients. Third, we evaluate the battery costs of Caelus on a smartphone, as well as the battery savings of attestor-partitioning. Finally, we evaluate the network bandwidth overhead of sending and retrieving attestations in Caelus.

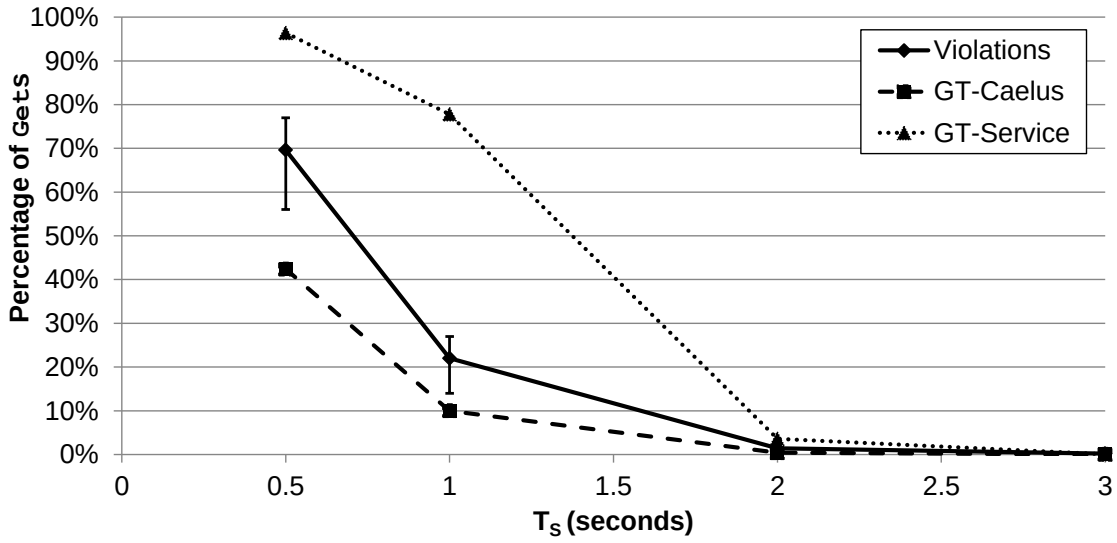


Figure 5.10: Percentage of `Gets` with Consistency Violations on S3 as a Function of $T_S$. GT-Caelus represents the true number of $T_{Caelus}$ violations, and GT-Service represents the true number of $T_S$ violations.

### 5.7.1 Detecting consistency violations

We begin by evaluating Caelus's effectiveness at detecting consistency violations using our eventual consistency prototype on S3. Amazon does not publish a visibility time bound for S3. Thus, we vary $T_S$ and measure the effect on the number of consistency violations detected by Caelus. Using a $T_S$ smaller than what S3 supports simulates a malicious cloud provider that tries to claim a shorter visibility time bound than what they can deliver.

We deploy Caelus on S3 in the US Standard Region, which automatically replicates data across Amazon data centers in the USA. We then deploy cloud servers on EC2 in the Oregon data center on the west coast using a t1.micro instance with 2 GHz Intel Xeon E5-2650 cores and 600 MB memory and the Northern Virginia data center on the east coast using a m3.2xlarge instance with an 8-core 2.5 GHz Intel Xeon E5-2670 Processor and 30 GB of memory. Four "writer" clients are running on the Northern Virginia server and repeatedly perform `Puts` of non-repeating 1 MB values on a key. A "reader" client is connected to the Oregon server and repeatedly performs `Gets` on the same key. The reader client runs on the smaller t1.micro instance with 2 GHz Intel Xeon E5-2650 cores and 600 MB of memory. We set $\epsilon$ to be 100 ms, $\delta$ to be 5 ms, $T_A = 500ms$ and vary $T_S$ between 0.5 and 3 seconds, taking the average over five runs. We also log the time of every `Put` and `Get` and perform an offline analysis to extract the ground truth (GT) number of $T_{Caelus}$ and $T_S$ violations. We then plot the results in Figure 5.10. As stated by its guarantees, Caelus detects all $T_{Caelus}$ violations and some but not all $T_S$ violations. As the $T_S$ increases, more operations are replicated by S3 in time, resulting in fewer true and detected violations.

In Figure 5.11, we hold $T_S$ fixed at 0.5 seconds while varying $T_A$ between 0.5 and 3 seconds. The number of true violations of $T_S$ stays the same, but the number of true $T_{Caelus}$ violations and those detected by Caelus decreases as $T_A$ increases, illustrating how a larger $T_A$ decreases Caelus's ability to detect $T_S$ violations.

### 5.7.2 Client verification costs

Since Caelus verification operations occur asynchronously, they are not on the critical path of any `Put` or `Get` operations and thus do not affect the performance of these operations. However, Caelus increases CPU utilization

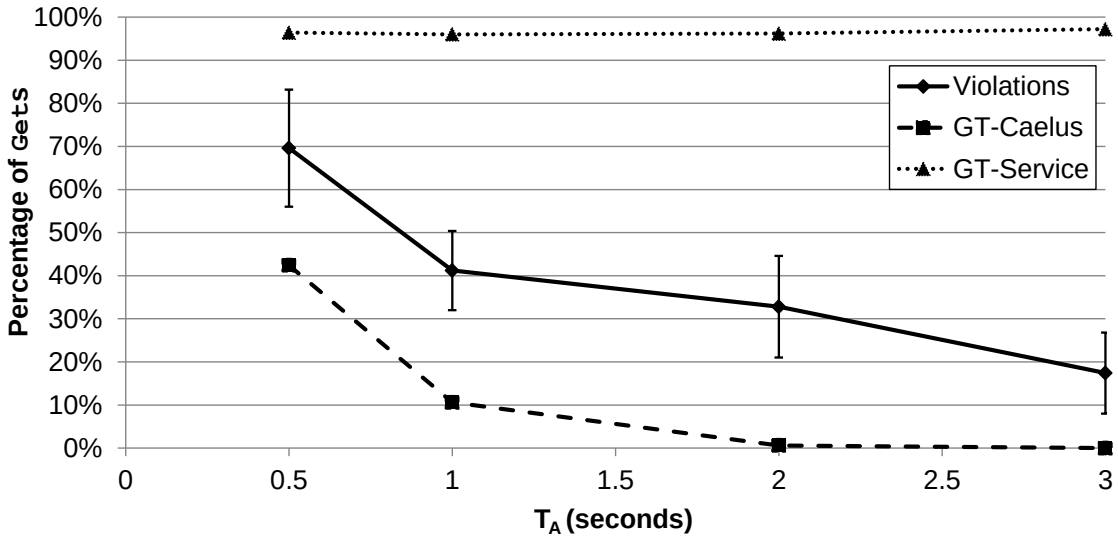


Figure 5.11: Percentage of Gets with Consistency Violations on S3 as a Function of $T_A$.

Table 5.2: Consistency verification performance on a PC.

| | Attest ($\mu s$) | Presence ($\mu s$) | Consistency ($\mu s$) |
|---|---|---|---|
| Strong-Get | 85.7±2.02 | 399.25±11.64 | 86.1±13.21 |
| Strong-Put | 85.7±2.02 | 402.38±26.08 | 9.52±2.19 |
| Eventual-Get | 67.5±2.26 | 392.59±12.57 | 17.79±2.71 |
| Eventual-Put | 67.5±2.26 | 410.78±8.45 | 18.47±6.68 |
| Causal-Get | 97.6±2.88 | 307.44±16 | 595.69±50.43 |
| Causal-Put | 97.6±2.88 | 319.03±8.15 | 2.7±1.75 |

as both verification and attestation contain cryptographic (2048-bit RSA with SHA256) and logical computations. We evaluate the computational costs of the different consistency model verification procedures by running them against our strong consistency prototype. The strong consistency server never causes consistency violations and evaluates the worst-case computational costs because operations must pass all tests to verify correctly, while Caelus will not perform any further checks on an operation once it detects that an operation violates consistency.

We measure the time to perform verifications on both a PC with a 3.4 GHz Intel i7-2600 Processor and 16 GB of memory and on a rooted stock Google Nexus 5 phone with a 2.3 GHz processor and a 2300 mAh battery. We run measurements in our lab to minimize network variability. Therefore, machines are connected to a local

Table 5.3: Consistency verification performance on a smartphone.

| | Attest ($ms$) | Presence ($ms$) | Consistency ($ms$) |
|---|---|---|---|
| Strong-Get | 1.49±0.06 | 2.24±0.33 | 0.05±0.03 |
| Strong-Put | 1.49±0.06 | 2.15±0.14 | 0.01±0.01 |
| Eventual-Get | 1.40±0.12 | 1.91±0.11 | 0.74±0.22 |
| Eventual-Put | 1.40±0.12 | 2.22±0.11 | 0.03±0.01 |
| Causal-Get | 1.75±0.13 | 1.79±0.13 | 2.53±0.39 |
| Causal-Put | 1.75±0.13 | 2.18±0.09 | 0.022±0.01 |

Table 5.4: Battery savings and percentage time sleeping comparison between when the phone acts as an attestor and when attestor-partitioning is used.

| | **Battery (mAh)** | **Sleeping (%)** |
|---|---|---|
| Idle | 20.85 | 98.5 |
| Single Attestor (WiFi) | 90.2 | 0 |
| Single Attestor (LTE) | 90.29 | 0 |
| Root Attestor (WiFi) | 22.57 | 98.3 |
| Root Attestor (LTE) | 22.57 | 97.7 |

Caelus service, also in our lab. We run YCSB [26] with a 50/50 mix of `Put` and `Get` operations, with no delay between operations on both machines, resulting in an applied workload of 26 ops/s. $T_A$ is set to one second, and we take the average over five runs.

The per-operation cost of the individual steps in the verification procedure are tabulated for the PC in Table 5.2 and for the Nexus 5 in Table 5.3. The consistency column records the cost of all the model-specific consistency checks, which are generally fast with the exception of `Gets` under causal consistency. This check requires an iterative search through the log to find all operations with vector clocks between the `Get` and matching `Put`. Cryptographic operations are the main source of overhead for Caelus. Out of the three components of the verification operation, the presence check component dominates the overall cost because there is a public-key signature verification performed on each operation in the log segment. These relative trends hold on both the PC and the Nexus 5, except that the PC is roughly 5-18$\times$ faster at cryptographic operations, which is to be expected. We also evaluate the cost of performing the signing operations and attestations and found that they are dominated by the cost of the RSA signature operation, which takes about 11 ms on the PC and 60 ms on the Nexus 5, regardless of the type of operation being signed.

Overall, the cost of Caelus operations is not high, and we find that these operations take about 8.8-16.4% of the CPU time on our test devices. Currently, our Caelus prototype signs and verifies individual cloud operations and this makes up the bulk of the CPU overhead. Batching signing cloud operations would reduce both the number of signatures and verifications and, thus, lessen the CPU overhead of Caelus.

### 5.7.3 Phone battery consumption

When regular Caelus client devices have no operations to perform on the cloud service, they can perform a client leave and go to sleep, so Caelus imposes no battery cost on normal client devices. The only devices that have additional duties in Caelus, even if they have no operations to perform, are the RA and the AA. Thus, we measure the battery impact on the RA when it could be otherwise idle. In addition, recall that the RA should select an AA that is already awake, so we also measure the battery impact of Caelus on an AA that is running other tasks.

We use the same phone used for verification cost measurement. We use battery-level readings from the OS and the percentage of time the phone spends sleeping to measure the benefits of attestor-partitioning. To get a baseline, we first perform measurements on an idle phone in its default configuration with basic services and applications running and background synchronization disabled. We then compare this to the battery consumption of the phone acting as a single attestor in the basic system and the Root Attestor using attestor-partitioning. For these experiments, we have clients run a simulated image browsing and editing workload with a mix of random 330 `Gets` and 30 `Puts` of 1 MB values every 30 minutes. $T_A$ is set to 1 second, and $T_R$ is set to 5 min. We perform measurements when the phone is on a WiFi network and a cellular LTE network. We run each experiment for at least 30 minutes or longer until the workload is finished and normalize the results to a 30-minute period in

Table 5.5: Battery drain and average CPU frequency of an active phone with and without the attestor role.

| | **Battery (mAh)** | **CPU (GHz)** |
|---|---|---|
| Active Attestor (WiFi) | 431.01 | 1.66 |
| Active Attestor (LTE) | 433.87 | 1.52 |
| No Caelus (WiFi) | 366.17 | 1.64 |
| No Caelus (LTE) | 343.66 | 1.61 |

Table 5.6: Network bandwidth consumed by Caelus operations.

| **Operation** | **Cost (Bytes)** |
|---|---|
| `Read_History` | $1411 + 1087 \times \|Puts\| + 695 \times \|Gets\|$ |
| `Write_Attest` | 756 |
| `Read_Attest` | $2582 + 1087 \times \|Puts\| + 695 \times \|Gets\|$ |
| `Select` | 1421 |

Table 5.4. Our battery consumption measurement tool rounds up to the nearest percentage of battery capacity (i.e., 23 mAh). The results show that attestor-partitioning uses negligible battery charge over a completely idle phone with only a slight increase in battery usage and a slight decrease in the time spent sleeping. However, compared to the phone acting as a single attestor, running as an RA with attestor-partitioning reduces the additional battery drain over an idle phone by about 40×.

In cases where a device is acting as an AA, the device is assumed to be running some other workload that prevents it from sleeping. To evaluate the battery impact on such a device, we run the same image viewing workload as above on the phone. To simulate UI events, we use the *monkey* tool, which generates random UI events. We compare the battery drain and average CPU frequency when the phone is acting as an Active Attestor with when Caelus on the phone is completely disabled and tabulate the results in Table 5.5. To isolate the cost of Caelus components, fetching and verifying attestations is disabled in all "No Caelus" cases. The results show that acting as an attestor on an active phone adds roughly a 17-26% increase in battery consumption.

### 5.7.4 Network cost

On top of existing `Get` and `Put` operations, Caelus adds operations that fetch log segments, read and write attestations, and assigns the attestor role through select operations—all of which consume network bandwidth. To measure this cost, we measure the amount of data before it is encoded by XML-RPC. Because XML-RPC transmits data in ASCII (Base64 encoding), the size of data to transmit increases by 1.5×. Using a more efficient binary packet format in our prototype would have avoided this unnecessary artifact. We note that this measurement method also does not consider transport protocol overhead, but these costs are well understood (usually about 40-60 bytes per packet).

Table 5.6 gives the cost of various Caelus control operations. Note that the cost of `Read_History` and `Read_Attest` operations depend on the number of `Put` and `Get` operations attested, as this affects the size of the history log segment that is read. For the image browsing workload in the previous section, the client uses about 1.14 MB on `Read_Attest` messages, the AA uses about 3.65 MB on `Read_History` and `Write_Attest` messages, and the phone uses about 8.33 KB on `selection` messages. When amortized over the 360 MB of data transferred in the workload, this works out to about 13 KB of network bandwidth overhead per megabyte of transferred data or about 1.3%. While these costs are fairly small, they are actually smaller in practice, since they only exist if clients are active and using the cloud service. If the cloud service is not being used, the clients use no network bandwidth at all.

## 5.8 Discussion

**Smartphone Connectivity.** While smartphones are designed to be constantly connected, and cellular coverage is available in most populated areas of the world, momentary gaps in cellular connectivity are still a common occurrence. To better understand this phenomenon, we performed an informal smartphone connectivity study. We acknowledge that our study has limitations—the participants are from the same geographical area, so the study is limited to the four or five carriers who service the area. However, given that cellular coverage will only continue to improve in all parts of the world, we believe that the results we attain here should be representative of what most populated areas of the world can achieve in the near future.

To record the availability of phones, we built a simple Android application that records the periods when the phone is not connected to the Internet. The application continually monitors network connectivity on both cellular and WiFi interfaces by registering for network status events. The application was installed on the phones of 12 participants in our lab over a 7-month period. We only collected connectivity data but no personal information is collected. The total time the phones had network connectivity over the total monitored time is 97.81%. The average duration of a disconnection is roughly exponentially distributed, with a mean of 94 seconds, and the longest measured period of disconnection is 5.7 hours. About 90% of disconnections last for less than 2 minutes, suggesting that, even if smartphone unavailability is encountered, it does not last long enough for a human user to perceive much inconvenience. In addition, we found that most disconnection events tend to be clustered, suggesting that they are related to the user's physical location. Thus, if the user is trying to access the cloud service while in an area of poor reception, they can likely remedy the situation by moving to a different location.

Various industry measures indicate that smartphone usage is rising, so one would assume that cellular networks must be fairly reliable to have fostered such heavy use. Our findings do not contradict this notion, and they suggest that smartphones have a high enough level of connectivity so that episodes of connectivity loss are short and isolated.

**Honouring Timestamps.** Caelus requires the DKVS to honour client timestamps to avoid violating consistency models for benign reasons such as slow processing or network congestion. Without having the DKVS honour client timestamps, Caelus may suffer from false positives. Assuming there is an upper bound for the end-to-end delay between clients issuing operations and the DKVS processing operations, having Caelus delay the start of processing those operations decreases the number of false positives.

Figure 5.12 illustrates an example of an issue that can occur when we do not enforce the DKVS to order operations based on clients' timestamps. Suppose that the DKVS in the figure provides strong consistency. At first glance, the log in the figure may look like there is a consistency violation because the `Get(X,1)` does not read the latest `Put(X,2)`. However, the actual ordering is the `Put(X,2)` and then the `Put(X,1)` due to the long processing and networking delay for the path of the `Put(X,1)`. Therefore, it is a false positive if Caelus raises an alarm for the `Get(X,1)` as a consistency violation. This situation can occur as follows: (1) Client 1 sends a `Put(X,1)` at time 1; (2) Client 2 sends a `Put(X,2)` at time 2; (3) Client 3 sends a `Get(X,_)` at time 3; (4) the `Put(X,2)` arrives at and gets committed by the DKVS at time 4 before the `Put(X,1)`; (5) after experiencing a long delay, the `Put(X,1)` arrives at and gets committed by the DKVS at time 5; (6) the `Get(X,_)` arrives at and gets committed by the DKVS at time 6 and, therefore, the `Get` reads the value 1 for the key X; (7) the `Put(X,1)` gets logged at time 7; (8) the `Put(X,2)` gets logged at time 8; and (9) the `Get(X,1)` gets logged at time 9.

If the DKVS honours the client timestamps, this issue will not occur. One method is modifying the DKVS to delay before committing operations. Figure 5.13 illustrates this solution. Again, suppose strong consistency

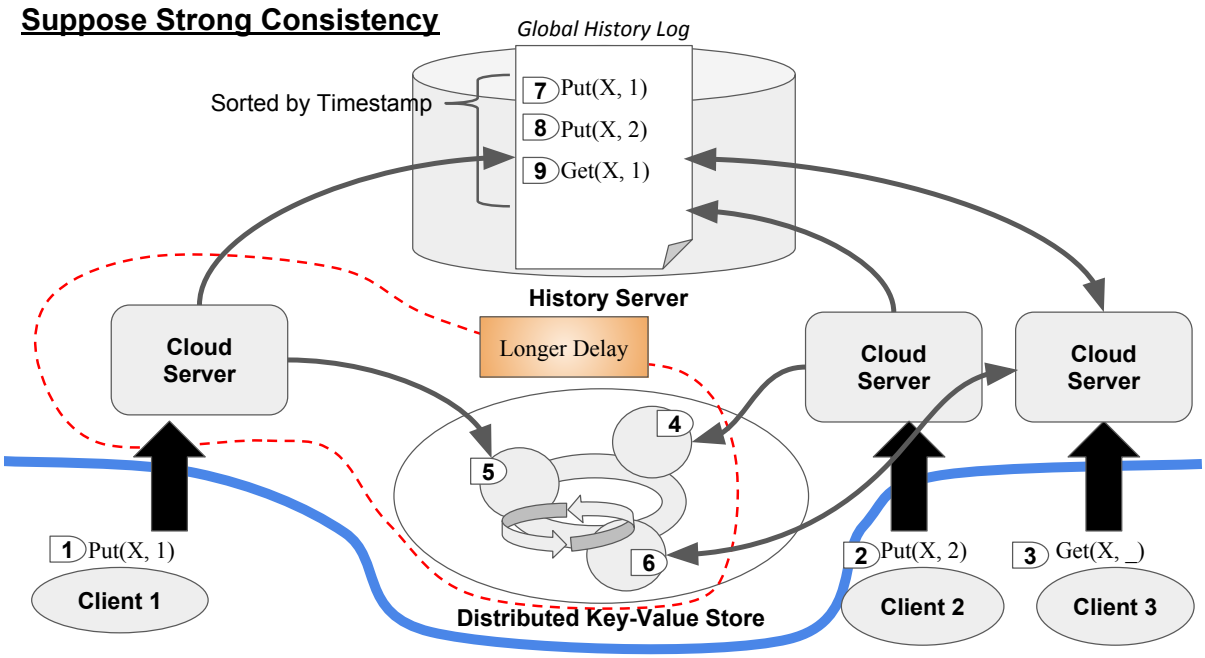


Figure 5.12: Operation Reordering due to Benign Delay May Lead to False Positives. Assuming Strong Consistency is provided, two arbitrary operations may be processed in reverse order due to benign delays taking longer than usual. This can happen regardless of the order in which the clients sent them. From the client's perspective, a benign delay is indistinguishable from the attack by the malicious cloud. (The underscore for `Get(X,_)` indicates that the value has not yet been read by the `Get` operation.)

is provided here. Note that even if the DKVS receives the `Put(X,2)` before the `Put(X,1)`, the distributed key-value store should wait until the upper bound of the delay has passed since the `Put(X,2)` is issued. While waiting, the `Put(X,1)` must have been arrived because it has been issued earlier than the `Put(X,2)`, and we assume the end-to-end delay from clients to the DKVS is bounded. By the end of waiting, the `Put(X,1)` must have been arrived and committed. Then, the DKVS can finally commit the `Put(X,2)`. The `Put(X,1)` and the `Put(X,2)` are performed in the order of client timestamps. Likewise, the `Get(X,_)` gets committed after two `Puts` and the `Get` reads 2 instead of 1, which is valid under strong consistency. Waiting for the upper bound of the delay before committing operations removes variation in the rate of arrival for client requests on the DKVS. Although this solution trivially introduces additional latency per operation, it allows clients to order operations performed on the DKVS without requiring a trusted component on the cloud service and without false positives. Any additional reordering or delaying of operations is can only be due to malicious reasons. Thus, it can be seen as a trade-off between latency for each request and security. The duration of waiting may be shortened for weaker consistency models due to more relaxed restrictions on the ordering and timing of operations. There may be better solutions for this issue, and exploring them is left for future work.

**Adversarial Model.** Caelus assumes active attackers who attempt to mount consistency attacks to trick client applications to make incorrect decisions. Therefore, in Caelus, nodes can have Byzantine behaviors. Comparing with Modulo, which assumes a fail-stop failure model, Caelus takes Byzantine failures into account and attackers are considered to be more active.

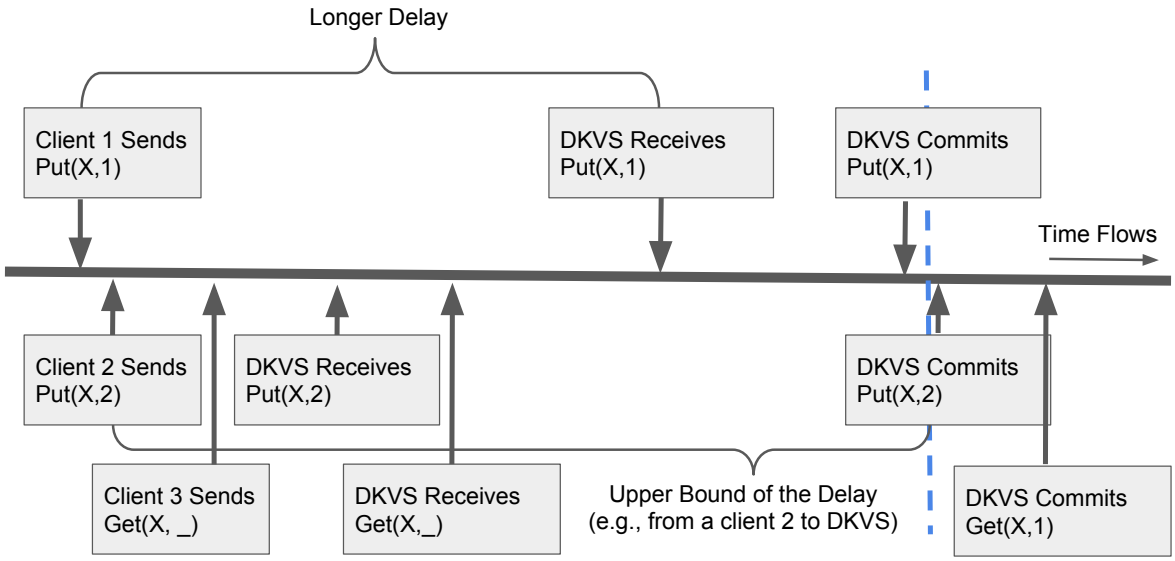


Figure 5.13: Waiting for the Upper Bound for the Delay and Requiring the Distributed Key-Value Store to Honour Client Timestamps Allows to Distinguish Benign Delay from Malicious Delay.

# Chapter 6

# Conclusion

Replicated distributed storage systems are becoming a more and more essential component of cloud services, on which our daily life increasingly heavily depends. These systems distribute and replicate data to support a large number of clients spread across different geographic locations. Distributing and replicating data contributes to providing various benefits such as reliability, durability, scalability, availability and short latency, but it can pose significant risks regarding data consistency. First, software bugs can be present in the implementations of resync mechanisms of replicated distributed storage systems. These are critical bugs because they can violate virtually every existing consistency model in practice by infringing on the most fundamental consistency guarantee—convergence. Second, replicated distributed storage systems can be compromised by adversaries who can mount subtle cyberattacks, causing consistency violations. Because end-user devices are constrained in terms of their battery power and network connectivity, it is very difficult to detect consistency violations using these devices. In this thesis, we discuss solutions for these issues and present novel techniques to mitigate them.

First, we built a bug-finding tool, Modulo, that can automatically find convergence failure bugs in distributed storage systems. Those bugs cause failures to achieve convergence between diverged replicas. By focusing on exploring divergence and convergence behavior with Divergence Resync Models, we can more effectively stress systems to manifest such bugs. Divergence Resync Models are generic enough to allow Modulo to be integrated with three different well-known open-source distributed storage systems: ZooKeeper, MongoDB and Redis. As a result, we showed that Modulo could effectively find software bugs in all three systems.

Second, we built a distributed system, Caelus, that can detect consistency violations of untrusted cloud storage services with end-user devices in near real-time. Cealus utilizes an untrusted cloud provider for communication between clients without requiring direct client-to-client communication channels. With scheduled attestation along with cryptographic techniques, the communication via untrusted cloud services can be secured. Also, Caelus partitions the role of the attestor to flexibly delegate the job of frequently generating attestation to an arbitrary user device to conserve battery, while timely detection is still maintained. Our evaluation showed that Caelus can detect violations of bounded eventual consistency simulated on Amazon's S3 storage service. Also, it is demonstrated that attestation-partitioning can reduce the battery impact of the root attestor by about 40×, and the cost in CPU time and network bandwidth overhead is small.

With this thesis, we showed that it is possible to significantly improve the detection of convergence failure bugs and consistency attacks that cause consistency violations. We made contributions for the detection of consistency violations. Even if there are frequent failures while maintaining a high degree of asynchrony, it is viable to effectively find CFBs by focusing on exploring divergence and convergence behaviors with our targeted state-

space exploration approach. Also, even with client devices that are limited in terms of battery power and network connectivity issues, it is possible to provide a timely battery-friendly detection mechanism for consistency violations of untrusted cloud storage services using a combination of cryptography, network security and distributed systems techniques.

## 6.1 Future Work

**Automating and Generalizing Modulo.** Although Modulo can find CFBs more directly than previous approaches, it has challenges that limit it from becoming a more automatic and general tool. It still requires users to write abstract DRMs that can effectively look for bugs. Also, Modulo is limited to finding bugs that can manifest with a small set of external events only. There are many bugs, including CFBs, that Modulo cannot find reliably and require tighter control for internal events. For these bugs, it is not known if Modulo's targeted state-space exploration approach can be useful or practical.

One solution for the automation of writing DRMs is using model inference. By generating a large number of random workloads to the systems, we may infer abstract models automatically by looking at inputs and outputs. In addition, an underlying OS may be enhanced with a generic event-monitoring feature. System-level events can be monitored using support from the underlying OS, so an application-agnostic model can be inferred. With the inferred abstract model, we may enable automated input generation.

Another potential line of future work is to see if Modulo's targeted approach can be effective in finding bugs that rely on internal event interleaving. Similar to fuzzing, we may repeatedly run a schedule to manifest bugs with some probability. This approach may require some additional control over internal events to interleave them in a specific order and timing. We can adjust the set of events we interleave. Thus, Modulo can be more generalized in terms of the types of events to interleave and types of bugs it can find while maintaining its targeted state-space exploration.

**Relaxing Assumptions and Optimizing Caelus.** Caelus has a significant limitation that stems from its requirement of honouring client timestamps for enforcing the total order of operations. It relies on the assumption on the upper bound for the end-to-end delays of the network path between two distributed nodes. The implication is that distributed key-value store servers should wait up to this bound before committing operations to ensure all previous operations to them are already committed. This assumption was necessary for us to allow clients to correctly reconstruct the order of operations based on clients' timestamps. Consequently, the server architecture must guarantee that operations are actually ordered based on timestamps. Thus, artificial delays are unnecessarily introduced on the cloud for more relaxed consistency models.

Alternatively, we may allow the cloud to determine the order of all operations for weaker consistency models. This will reduce the level of consistency in detecting consistency violations because the total order determined by the cloud may no longer reflect the true order of operations issued by clients. The cloud is then allowed to change the timing and order of operations as long as such changes do not violate weak consistency models promised. However, with this slight relaxation, we can expect a significant gain in the performance and scalability because we do not need to have a distributed key-value store to inject artificial delays to honour the order of client timestamps. This approach requires a complete redesign of Caelus. We need to modify the distributed key-value store and the history server to disclose the timing and order of operations. Also, clients should consider this redesign in their checking algorithms. We can explore the optimal design and can measure the performance gain compared to the original design.

# Bibliography


[1] Apache HBase. https://hbase.apache.org.

[2] Couchbase Server. https://www.couchbase.com.

[3] The ieee 802.1 working group. https://1.ieee802.org/tsn/.

[4] The ieee 802.1 working group. https://datatracker.ietf.org/wg/detnet/about/.

[5] MemcacheDB. http://memcachedb.org.

[6] MongoDB. https://www.mongodb.com/.

[7] Riak. http://basho.com/products.

[8] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, February 2012.

[9] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar 1995.

[10] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.

[11] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 51–68, Berkeley, CA, USA, 2018. USENIX Association.

[12] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What Consistency Does Your Key-Value Store Actually Provide? In *The 6th International Conference on Hot Topics in System Dependability (HotDep)*, October 2010.

[13] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Softw. Syst. Model.*, 4(3):326–345, July 2005.

[14] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable Data Possession at Untrusted Stores. In *The 14th ACM Conference on Computer and Communications Security (CCS)*, November 2007.

[15] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *The VLDB Endowment*, 5(8):776–787, April 2012.

[16] Bankmycell. How many smartphones are in the world? https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[17] David Bermbach, Sherif Sakr, and Liang Zhao. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In *The 5th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2013)*, August 2013.

[18] David Bermbach and Stefan Tai. Eventual Consistency: How Soon Is Eventual? An Evaluation of Amazon S3's Consistency Behavior. In *The 6th Workshop on Middleware for Service Oriented Computing (MW4SOC)*, December 2011.

[19] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, November 2013.

[20] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *The 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[21] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 194–208, London, UK, UK, 2001. Springer-Verlag.

[22] H.C.H. Chen and P.P.C. Lee. Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage: Theory and Implementation. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):407–416, Feb 2014.

[23] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 147–158, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[24] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[25] J. Cook. Google Drive Now Has 10 Million Users: Available on iOS and Chrome OS. http://techcrunch.com/2012/06/28/google-drive-now-has-10-million-users-available-on-ios-and-chrome-os-offline-editing-in-docs/.

[26] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *The 1st ACM Symposium on Cloud Computing (SOCC)*, pages 143–154. ACM, 2010.

[27] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak,

Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[28] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[29] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 249–262, Berkeley, CA, USA, 2016. USENIX Association.

[30] David L. Dill. The murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[31] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, August 2003.

[32] fdingiit. Questions about potential data inconsistency / unexpected FSYNC base on 4.0.2, 2017. https://github.com/antirez/redis/issues/4407.

[33] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *The 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[34] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[35] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, pages 174–, Washington, DC, USA, 1999. IEEE Computer Society.

[36] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, October 1999.

[37] Gartner. Gartner forecasts worldwide public cloud services revenue to reach $260 billion in 2017, 2017. https://www.gartner.com/newsroom/id/3815165.

[38] GeorgeBJ. Replication inconsistent issue, 2015. https://github.com/antirez/redis/issues/2694.

[39] Sanjay Ghemawat and Jeff Dean. Leveldb: A Fast and Lightweight Key/Value Database Library by Google. https://code.google.com/p/leveldb/.

[40] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, August 1997.

[41] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[42] W. Golab, M.R. Rahman, A. Auyoung, K. Keeton, and I. Gupta. Client-Centric Benchmarking of Eventual Consistency for Cloud Storage Systems. In *The 34th International Conference on Distributed Computing Systems (ICDCS)*, June 2014.

[43] Wojciech Golab, Xiaozhou (Steve) Li, Alejandro López-Ortiz, and Naomi Nishimura. Computing weak consistency in polynomial time: [extended abstract]. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 395–404, New York, NY, USA, 2015. ACM.

[44] Rachid Guerraoui and Maysam Yabandeh. Model checking a networked system without the network. In *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, page 225, 2011.

[45] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.

[46] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 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)*, 2014.

[47] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 265–278. ACM, 2011.

[48] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[49] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[50] Ari Juels and Burton S. Kaliski Jr. PORs: Proofs of Retrievability for Large Files. In *The 14th ACM Conference on Computer and Communications Security (CCS)*, October 2007.

[51] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of NSDI'07: 4th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, 2007.

[52] Beom Heyn Kim. Data inconsistency occurs if follower has uncommitted transaction in the log while synchronizing with the leader that has the lower last processed zxid, 2018. https://issues.apache.org/jira/browse/ZOOKEEPER-2832.

[53] Beom Heyn Kim. Initial sync not replicating old oplog entries may have a stale node give up resync and permanently stay in recovering state, 2018. https://jira.mongodb.org/browse/SERVER-35774.

[54] Beom Heyn Kim. The truncate() function in filetxnlog.java may fail to properly remove an uncommitted write resulting in data inconsistency, 2018. https://issues.apache.org/jira/browse/ZOOKEEPER-2946.

[55] Beom Heyn Kim, Wei Huang, and David Lie. Unity: Secure and Durable Personal Cloud Storage. In *The 2012 ACM Workshop on Cloud Computing Security Workshop (CCSW)*, October 2012.

[56] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. *IEE Proceedings - Software*, 146(4):187–192, Aug 1999.

[57] Kyle Kingsbury. Distributed systems safety research. https://jepsen.io/.

[58] Sriram Krishnan and Eva Tse. Hadoop platform as a service in the cloud, 2013. https://medium.com/netflix-techblog/hadoop-platform-as-a-service-in-the-cloud-c23f35f965e7.

[59] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, November 1992.

[60] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[61] Leslie Lamport. Specifying concurrent systems with tla+, 1999.

[62] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. Samc: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.

[63] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data repository (SUNDR). In *The 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[64] Jinyuan Li and David Mazieres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *The 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2007.

[65] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *The 28th International Symposium on Computer Architecture (ISCA)*, July 2001.

[66] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: transparent model checking of unmodified distributed systems. In *Proceedings of NSDI'09: 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, 2009.

[67] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *The 23rd ACM Symposium on Operating Systems Principles (SOSP)*, November 2011.

[68] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[69] T. Lodderstedt and M. Scurtescu. OAuth 2.0 Token Revocation. IETF RFC 7009, August 2013.

[70] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 20:1–20:16, New York, NY, USA, 2019. ACM.

[71] Peter Maass and Laura Poitras. Core Secrets: NSA Saboteurs in China and Germany. https://firstlook.org/theintercept/2014/10/10/core-secrets.

[72] Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. In *The 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[73] Mohammad Mannan, Beom Heyn Kim, Afshar Ganjali, and David Lie. Unicorn: Two-Factor Attestation for Data Security. In *The 18th ACM Conference on Computer and Communications Security (CCS)*, October 2011.

[74] Juan Martinez. The 50 best cloud services for smbs, 2017. https://www.pcmag.com/article/345308/20-of-the-best-cloud-services-for-smbs.

[75] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. *SIGOPS Opering Systems Review*, 42(4):315–328, April 2008.

[76] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[77] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. Cmc: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.

[78] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. Automatic test generation: A use case driven approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, March 2006.

[79] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[80] A. Jefferson Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.

[81] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.

[82] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *The 2nd ACM Symposium on Cloud Computing (SOCC)*, October 2011.

[83] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *The 2011 USENIX Annual Technical Conference (ATC)*, June 2011.

[84] Muntasir Raihan Rahman, Wojciech Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. Toward a Principled Framework for Benchmarking Consistency. In *The 8th USENIX Conference on Hot Topics in System Dependability (HotDep)*, October 2012.

[85] Research and Markets. $92.48 billion cloud storage market - forecasts from 2017 to 2022 - research and markets, 2017. https://www.researchandmarkets.com/research/lf8wbx/cloud_storage.

[86] Juniper Research. Cloud services to be adopted by 3.6bn consumers globally by 2018, juniper research finds, 2014. https://www.juniperresearch.com/press-release/cloud-computing-pr1.

[87] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.

[88] Salvatore Sanfilippo. Redis. https://redis.io/.

[89] Wendell Santos. 76 Storage APIs: Box.net, Amazon S3, Dropbox. http://www.programmableweb.com/news/76-storage-apis-box.net-amazon-s3-dropbox/2012/01/31.

[90] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for Untrusted Cloud Storage. In *The 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW)*, October 2010.

[91] Jiri Simsa, Randy Bryant, and Garth A Gibson. dbug: Systematic evaluation of distributed systems. In *SSV*, 2010.

[92] soloestoy. Redis 4.x PSYNC2 & RDB: data inconsistency between master and slave, bug located, 2017. https://github.com/antirez/redis/issues/4316.

[93] Sang Hyuk Son and Kwang Moo Choe. Techniques for database recovery in distributed environments. *Information and Software Technology*, 30(5):285 – 294, 1988.

[94] Statista. Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[95] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, November 1996.

[96] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[97] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013.

[98] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *The 24rd ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[99] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pages 491–515, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[100] Werner Vogels. Eventually consistent - revisited. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

[101] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[102] Brian Voo. 20+ Tools To Supercharge Your Dropbox. http://www.hongkiat.com/blog/dropbox-tools/.

[103] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *The 10th USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[104] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers' Perspective. In *The 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2011.

[105] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, volume 9, pages 229–244, 2009.

[106] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.

[107] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 651–662, New York, NY, USA, 2016. ACM.

[108] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.