ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

IDENTIFY BOTTLENECKS IN FAULT INJECTION TEST FOR DISTRIBUTED SYSTEMS

EPFL SEMESTER PROJECT REPORT

Zhili Pan

January 5, 2025

CHAPTER 1

ABSTRACT

Fault injection testing is an important methodology for assessing the resilience of distributed systems. However, its performance is often suboptimal. This project aims to find the performance bottlenecks within individual fault injection test executions via a case study. The results identify that consensus protocol steps, particularly leader elections and data synchronization, are major bottlenecks. Concepts such as executing steps in advance and limiting test data size are potential approaches to improve the test performance.

CHAPTER 2

INTRODUCTION

2.1 MOTIVATION

Distributed systems are the backbone of modern computing and play a crucial role in contemporary applications, especially in the era of cloud computing. However, the complexity inherent in distributed environments introduces a range of bugs that compromise the reliability, availability, and safety of these systems. For instance, in September 2024, I reviewed the latest 200 open issues in Kafka [4], and approximately 6% of them were identified as concurrency bugs [13] or time-of-fault bugs [14].

To address these challenges, engineers and scholars have developed various bug-finding techniques such as model checking [11, 21], formal verification [5, 19], and dynamic testing [1, 3]. Among these methods, fault injection testing is particularly valuable for its ability to simulate real-world failure scenarios and uncover sequences of faults that lead to bugs during actual execution.

Previous research about fault injection tests for distributed systems has primarily focused on efficiently exploring different sequences of faults. However, there has been a lack of thorough evaluation regarding the performance of individual test executions. This gap is significant because unidentified bottlenecks in each test execution can severely limit the overall fault injection test efficiency in distributed systems.

2.2 RESEARCH STATEMENT

This project seeks to address the gap in understanding the performance bottlenecks in single fault injection test runs for distributed systems. By conducting a case study, this project aims to provide a quantified evaluation of the factors limiting test performance. Additionally, this project explores potential approaches to improve the performance of fault injection tests for distributed systems, combining analysis results with insights from other literature.

CHAPTER 3

BACKGROUND KNOWLEDGE

3.1 JEPSEN

Jepsen [15] is an open-source Clojure library designed for testing distributed systems. It simulates real-world failure scenarios and validates the correctness of distributed systems with the presence of failure. More specifically, it generates sequences of faults, such as node crashes and network partitions, and evaluates how well a system maintains consistency under these adverse conditions. Jepsen is highly configurable and allows users to define test scenarios in a flexible and expressive manner. Users can easily specify the duration of faults, introduce external inputs or faults, and collect test logs. Jepsen's test logs include observations of the system under test, such as request status and timestamps, providing valuable information for performance analysis.

3.2 MONGODB REPLICA SET

MongoDB [7] is a well-known NoSQL database. A replica set [10] in MongoDB maintains a data set via a distributed group of *mongod* processes, providing data replication and a level of fault tolerance. A replica set contains several data-bearing nodes and, optionally, one arbiter node. Only one of the data-bearing nodes is deemed the primary node, while the other nodes are secondary nodes. The primary node handles all write operations while secondary nodes replicate the primary's data set and can serve read operations. MongoDB offers strong consistency by default while supporting tunable consistency levels, allowing applications to balance between consistency and performance based on their specific needs.

From version 4.0 onwards, MongoDB uses a Raft-based consensus protocol to achieve strong consistency [23]. This protocol replaces the push-based data synchronization model in the vanilla Raft protocol [17] with a pull-based model. Data synchronization in MongoDB can be initiated by any replica member and can happen between any two members. When adding or recovering a member to an existing replica set, MongoDB requires the node to either contain no data or have a recent copy of data from an existing member [6]. In the latter case, data is recent if it corresponds to a version created later than a certain time point. The leader election process of MongoDB's consensus protocol is identical to Raft's, mainly triggered in the event of the primary node's failure or unavailability.

CHAPTER 4

TEST TOOLS AND ENVIRONMENT

This project employed Jepsen as the primary testing tool. Unfortunately, most sample test projects provided by Jepsen were unexecutable due to outdated software versions, language versions, and/or operating system versions. Therefore, in this project, the experiments borrowed the test environment from a recent research [16], where the researcher aimed to strategically generate fault sequences in the Jepsen framework. All tests in this project were executed in a virtualized environment, specifically a Debian 11 virtual machine with a containerized version of the Jepsen environment. The test virtual machine consisted of one "control node" container for orchestrating the tests and five "node" containers for running the system under test. The entire virtual machine was hosted on an Ubuntu 24.04 system with 16 CPU cores and 32 GB of memory.

To ensure the accuracy of the tests, no other resource-consuming services were running simultaneously during the testing process. The versions of the software used in this project align with previous research, with MongoDB version 4.4.9 (released on September 20, 2021) and Jepsen version 0.2.7 (released on June 30, 2022). By using the same baseline test environment settings, the analysis results of this project are more comparable and relevant to the findings of previous research.

CHAPTER 5

CASE STUDY ON MONGODB

5.1 TEST CASE OBJECTIVE AND SETUP

The objective of this case study is to understand the performance bottlenecks in fault injection tests for MongoDB replica sets under node failure scenarios. The focus was on the basic replica set feature of MongoDB, with all five members as data-bearing and voting members for simplicity.

The primary fault examined in this case study was node crashes, modeled by forcibly killing the *mongod* processes. Scenarios tested included the crash of a majority of nodes, a minority of nodes, and specifically the primary node to understand their impact.

Recovery behavior varied depending on the nature of the node failure and the role of the failing node. Thus test cases were designed to support several recovery processes, including recovering with all, partial, or none of the data present before the kill, gracefully changing membership via reconfiguration commands, and purely (re)starting the *mongod* process.

In each single test run, Jepsen initialized the MongoDB replica set and allowed MongoDB to run normally for 10 seconds. Afterwards, Jepsen repeated the fault injection and recovery cycles 3 times, causing MongoDB to switch between fault and no-fault states repeatedly. As supported in Jepsen, the duration of staying in each state and overall runtime of a test run were configurable and fixed for each test run. Different test runs had varying fault/recovery durations, MongoDB read/write request rates, and recovery processes. The Jepsen log was the main source for all performance analysis, with 3 recovery cycles per test run providing repeated measures for each tested scenario.

The code for this case study is available at

https://github.com/sasoripathos/fitbottle

5.2 TEST RESULT AND ANALYSIS

Bugs usually occur with unexpected system states, such as inconsistent data or unexpected request results. Fault injection testing examines the validity of status changes introduced by the faults. To identify potential bugs, it is necessary for the testing framework to simulate the entire status change. The key measurement for this case study is the time needed for a MongoDB

replica set to transition between statuses. Specifically, I measured the duration needed for a MongoDB replica set to recover nodes so that they can resume handling user requests.

The first set of measurements involves recovery by simply restarting the *mongod* process with all data present before the kill. In these test cases, each fault injection and recovery kills or recovers a majority of members in the replica set (at least 3 out of 5 nodes) simultaneously. The results are shown in Figure 5.1. There are three measurements for three recovery cycles in each test run. Each bar in 5.1 represents the average of three measurements, with error bars indicating the maximum and minimum values of the three measurements. The results indicate that user request rate and fault interval have a negligible effect on system recovery speed. On average, a MongoDB replica set can recover the majority of nodes in about 4-5 seconds if the nodes contain valid recent data. When losing a majority of members, the MongoDB replica set stops serving user requests, and leader election attempts will not succeed [23]. There is typically at least one successful leader election during the recovery process, taking about 4-5 seconds to complete.

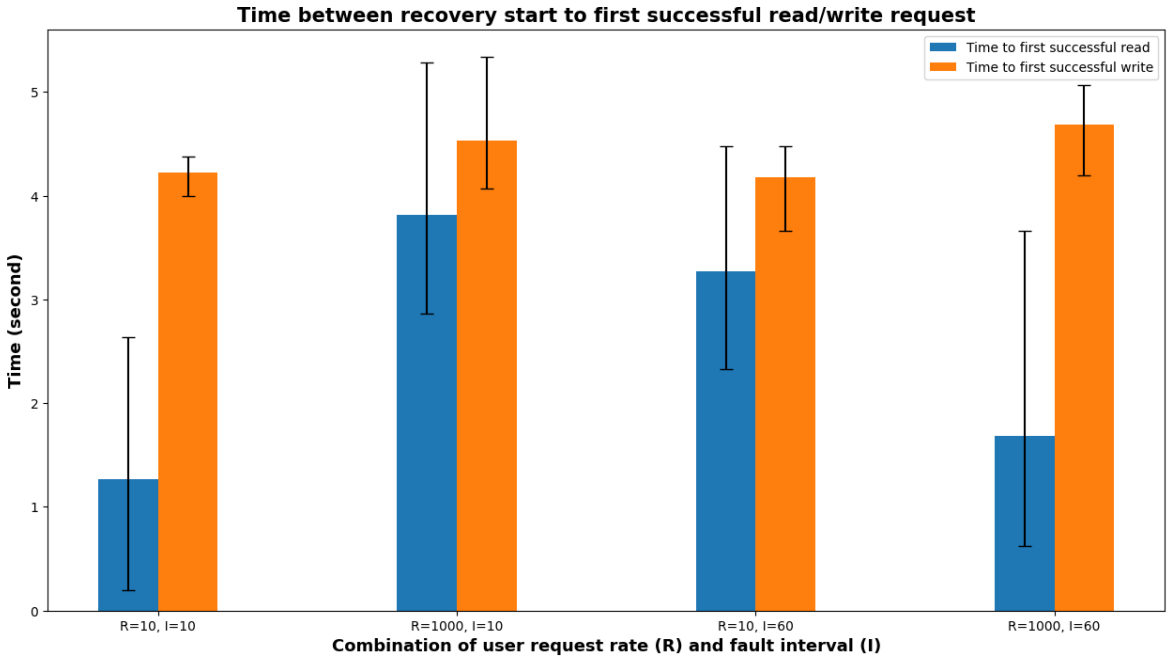

FIGURE 5.1: Measurements for simply restart with data present

The second set of measurements involves recovery by using MongoDB reconfiguration (reconfig) with all data present before the kill. This set covers cases where recovery is triggered by administrative actions. When each fault injection kills a majority of members in the replica set, the reconfiguration has to be forced as per MongoDB documentation [8]. The forced reconfiguration doesn't introduce much overhead in the measurements. The MongoDB replica set still takes around 5 seconds before serving both read and write operations (as shown in Figure 5.2).

Based on the two sets of measurements, one can estimate a leader election process taking approximately 5 seconds to complete in a MongoDB replica set with 5 members. It's important to note that MongoDB may require additional time to initiate an election. When a fault injection results in the loss of a minority of members (including the primary), the remaining nodes will not mark the primary as unavailable until the configured timeout elapses. Only then is a leader election process triggered. MongoDB indicates that the total time for marking a primary as

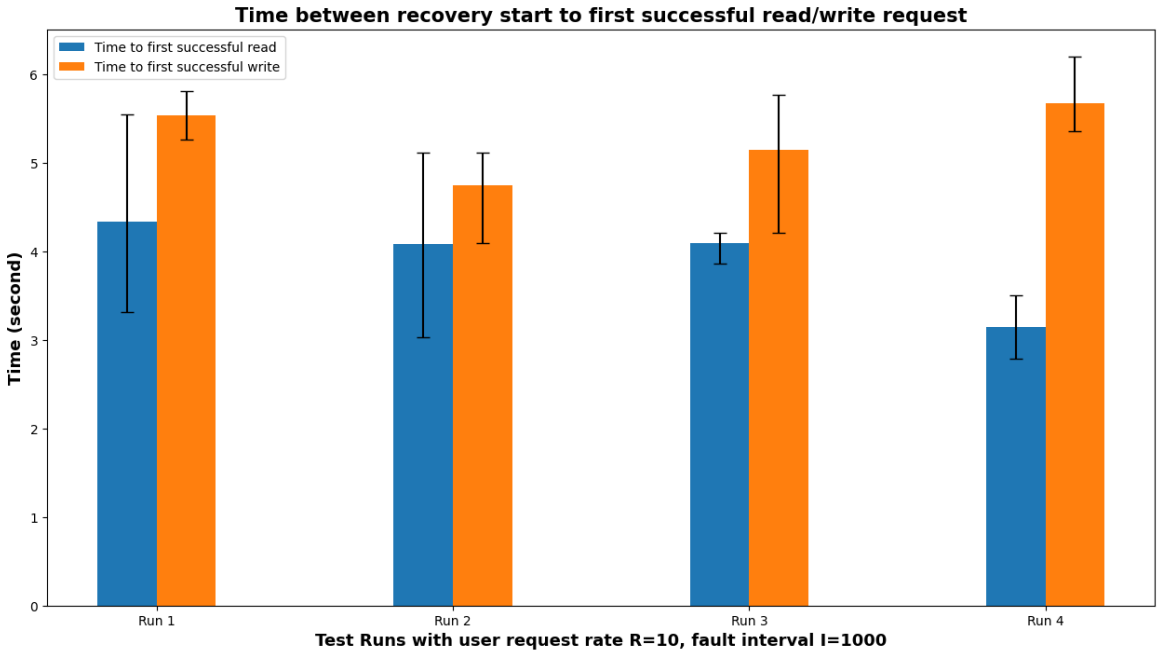

FIGURE 5.2: Measurements for reconfigure majority with data present

unavailable and completing an election typically does not exceed 12 seconds [9]. This upper bound is confirmed by the third set of measurements (as shown in Figure 5.3), which involved losing only the primary member in the fault. The time when continuous write request failure ends varies in each fault and recovery cycle, but roughly remains within 12 seconds.

The fourth set of measurements is for recovery which purely starts the *mongod* process with partial or no data present before the kill. Specifically, the fault recovery process involves removing some of the data before restarting the *mongod* process. In this case, the replica set is unavailable to respond to any new requests after the recovery, regardless of fault/recovery duration. Jepsen logs show an error saying "sessions not supported by cluster". This result indicates that mongodB requires additional steps other than restarting the *mongod* process when recovering a member with impaired data, reflecting MongoDB's constraint about the data directory during adding/recovering members. Additional data resynchronization is required.

The fifth set of measurements involves recovery by gracefully reconfiguring MongoDB without any data present before the kill. This set corresponds to the case where a brand new server is added as a member without any prior data. MongoDB automatically performs a full data synchronization for all new members. When each fault injection kills a majority of members in the replica set, it takes around 5-7 seconds to complete both data full synchronization and the leader election (as shown in Figure 5.4). The default user request generation logic in Jepsen was used, simulating a mix of small read and write requests. The overall data size in these test runs was very small compared to real-world scenarios. Even with the small data size, it is clear that full data synchronization adds noticeable overheads to the recovery process.

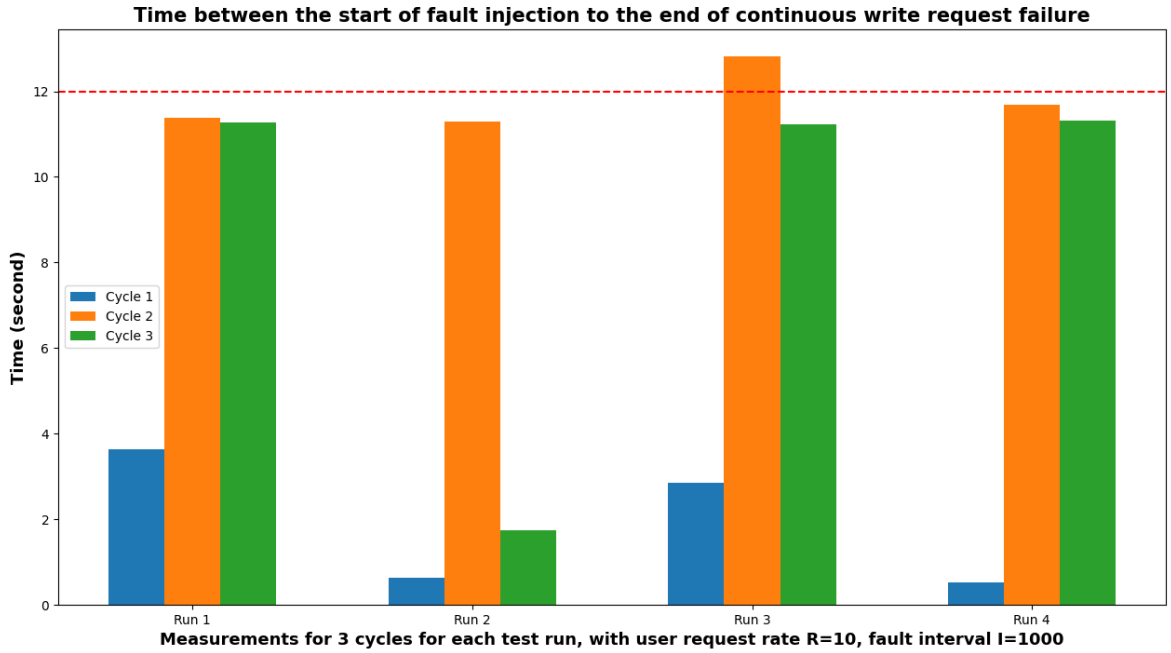

FIGURE 5.3: Measurements for losing primary

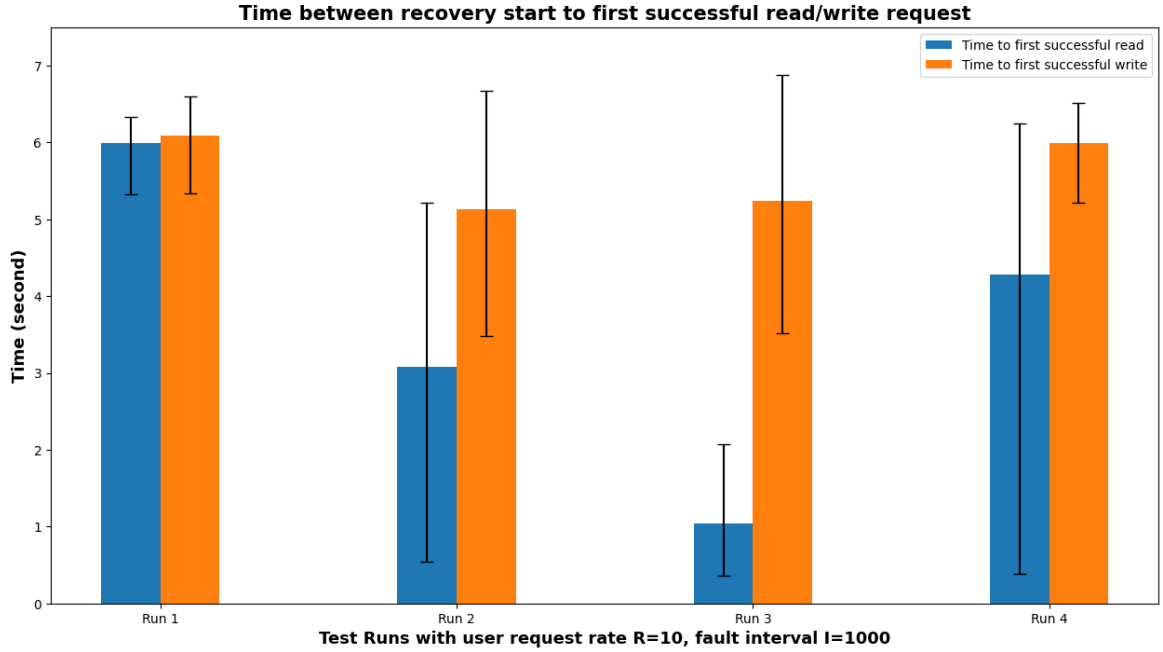

FIGURE 5.4: Measurements for reconfigure majority without data present

CHAPTER 6

BOTTLENECKS FOUND FROM CASE STUDY

In fault injection tests, the primary concern is the validity of status transitions in the presence of faults within distributed systems. From the MongoDB case study, it becomes evident that the status transition within a consistency-focused distributed system primarily involves executing the required steps according to the consensus protocol. This makes the performance of the consensus protocol itself a bottleneck for fault injection tests.

In a distributed data system like MongoDB, two major steps in its consensus protocol are leader elections and data synchronization. The leader election process involves less communication compared to the interaction between the system and its users, but it takes a significantly longer time to fulfill the semantics behind the communications. Data synchronization faces similar challenges, and it is also affected by factors such as simulated data size and network latency. These 2 steps are common in most consensus protocols, making them general bottlenecks for fault injection tests on consensus-based distributed systems like blockchains.

CHAPTER 7

POTENTIAL IMPROVEMENT

Generally speaking, the more efficient the consensus protocol is, the better performance it can achieve. Researchers have proposed many improvements on consensus protocols. For example, CRaft [18] employs erasure coding to reduce the amount of data that needs to be stored at the follower nodes. X-Raft [12] integrates a vitality system to evaluate node status in real-time, enhancing leader election and data replication. LRD-Raft [20] enables follower nodes to participate in data replication. There are also leaderless protocols like Racos [22], which eliminates the need for a leader. While these approaches offer significant improvements, they are unsuitable for test scenarios because the semantics of the system under test should remain the same as in the production environment.

While we cannot change the consensus protocol under test, we can borrow the concept from speculative execution to execute some steps ahead. In a fault injection test framework like Jepsen, developers have access to crucial information, such as the exact nature of the fault, the status of nodes, the likely new leader, the recovery logic, fault duration, and environmental guarantees. With this information, developers can make deterministic predictions about when recovery steps will be executed.

For instance, in the case where a MongoDB replica set loses a majority of members after a fault injection, the old leader will likely remain as the leader after a successful election during recovery. This is because the replica set cannot respond to user requests during the fault, ensuring no new data is added to the system. The old leader, having the latest data, will remain as leader after the election. With this knowledge, it is theoretically possible to simulate the leader election step immediately after the fault injection, rather than waiting for the recovery to begin. The practicality of such an approach depends on the specific protocol implementation and requires further exploration.

For data synchronization, MongoDB suggests copying sufficiently recent data from other active members to avoid full synchronization. This approach aligns with the concept of executing ahead. The feasibility of obtaining sufficiently recent data varies across different systems and scenarios. When the test data size is large, efficient data transition in the underlying storage becomes crucial. Techniques like storage snapshots may help but depend on the support available in the underlying storage. Advanced knowledge of the underlying storage is often necessary to develop or utilize an efficient data copy approach.

Another relatively straightforward approach is to limit the size of test data. Ideally, we want to use the least amount of data combined with correct fault sequences to find potential bugs. In test

frameworks like Jepsen, data size is configurable via setting user request rates. With the help of test frameworks, the focus shifts to finding correct and short fault sequences that reveal bugs. There is a wealth of research that explores optimizing fault sequences.

CHAPTER 8

LIMITATION AND FUTURE WORK

Within the time limitation, the case study above was conducted in a relatively ideal virtual environment and only explored one type of fault. There are many types of faults, such as network partitions, which represent valuable test cases for distributed systems. One possible follow-up project is to identify other bottlenecks with the presence of other types of fault in a real-world network environment like the cloud.

Finding new bugs was not the purpose of this project, but many validation results in above test runs are invalid. Jepsen's consistency checker, Elle, cannot guarantee the validity of the validation results [2]. Additionally, the software version used in this project is not the latest, so it is unclear if there are actual bugs in MongoDB. It would be valuable to manually investigate the validation results and redo the experiments on a newer version of MongoDB.

Nevertheless, implementing the executing ahead concept in a distributed system has great potential. An implementation attempt will reveal unnoticed technical difficulties. It may also prompt code reengineering in the target distributed system for better modularization. With a successful implementation, the fault injection testing performance will be further improved.

CHAPTER 9

CONCLUSION

This project provides valuable insights into the challenges and potential improvements for fault injection testing in distributed systems. The case study on MongoDB reveals that certain consensus protocol steps, particularly leader elections and data synchronization, are significant bottlenecks in single fault injection test runs. While optimizing the consensus protocol itself is not feasible in test scenarios, the concept of speculative execution has the potential to mitigate some delays. Additionally, limiting test data size and employing advanced storage techniques can enhance testing efficiency. The findings and proposals in this project may inspire future research on fault injection testing for distributed systems and contribute to the development of more reliable distributed systems.

REFERENCES

[1] Gluster community. *Test cases and Verification*. In: Development workflow of Gluster, https://docs.gluster.org/en/main/Developer-guide/Development-Workflow/#test-cases-and-verification.

[2] Elle contributors. *Elle*. https://github.com/jepsen-io/elle.

[3] Loic Dachary and Nathan Cutler. *Testing - Unit Tests*. In: Contributing to Ceph: A Guide for Developers, https://docs.ceph.com/en/reef/dev/developer_guide/tests-unit-tests/.

[4] Apache Software Foundation. *Apache Kafka*. https://kafka.apache.org/.

[5] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty and Brian Zill. IronFleet: proving practical distributed systems correct. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 1–17. ISBN: 9781450338349. DOI: 10.1145/2815400.2815428.

[6] MongoDB Inc. *Add Members to a Self-Managed Replica Set*. In: MongoDB Manual, https://www.mongodb.com/docs/manual/tutorial/expand-replica-set/#add-members-to-a-self-managed-replica-set.

[7] MongoDB Inc. *MongoDB*. URL: https://www.mongodb.com/.

[8] MongoDB Inc. *Reconfigure Replica Set with Unavailable Members*. In: MongoDB Manual, https://www.mongodb.com/docs/manual/tutorial/reconfigure-replica-set-with-unavailable-members/.

[9] MongoDB Inc. *Replica Set Elections*. In: MongoDB Manual, https://www.mongodb.com/docs/manual/core/replica-set-elections/.

[10] MongoDB Inc. *Replication*. In: MongoDB Manual, https://www.mongodb.com/docs/current/replication/.

[11] 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)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 399–414. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa.

[12] Fengqi Li, Jiaheng Wang, Weilin Xie, Ning Tong and Deguang Wang. X-RAFT: Improve RAFT Consensus To Make Blockchain Better Secure EdgeAI-Human-IoT Data. In: *IEEE Transactions on Emerging Topics in Computing* (2024), pp. 1–12. DOI: 10.1109/TETC.2024.3472059.

[13] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi and Chen Tian. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud

Systems. In: *SIGARCH Comput. Archit. News* 45.1 (Apr. 2017), pp. 677–691. ISSN: 0163-5964. DOI: 10.1145/3093337.3037735.

[14] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye and Chen Tian. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In: *SIGPLAN Not.* 53.2 (Mar. 2018), pp. 419–431. ISSN: 0362-1340. DOI: 10.1145/3296957.3177161.

[15] Jepsen LLC. *Jepsen*. URL: https://jepsen.io/.

[16] Ruijie Meng, George Pırlea, Abhik Roychoudhury and Ilya Sergey. Greybox Fuzzing of Distributed Systems. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1615–1629. ISBN: 9798400700507. DOI: 10.1145/3576915.3623097.

[17] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[18] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu and Dongsheng Wang. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 297–308. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/wang-zizhong.

[19] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 357–368. ISSN: 0362-1340. DOI: 10.1145/2813885.2737958.

[20] Heru Yang, Yuming Feng and Weizhe Zhang. LRD-Raft: Log Replication Decouple for Efficient and Secure Consensus in Consortium Blockchain-Based IoT. In: *IEEE Internet of Things Journal* (2024), pp. 1–1. DOI: 10.1109/JIOT.2024.3506601.

[21] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 213–228.

[22] Jonathan Zarnstorff, Lucas Lebow, Christopher Siems, Dillon Remuck, Colin Ruiz and Lewis Tseng. Racos: Improving Erasure Coding State Machine Replication using Leaderless Consensus. In: *Proceedings of the 2024 ACM Symposium on Cloud Computing*. SoCC '24. Redmond, WA, USA: Association for Computing Machinery, 2024, pp. 600–617. ISBN: 9798400712869. DOI: 10.1145/3698038.3698511.

[23] Siyuan Zhou and Shuai Mu. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 687–703. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/zhou.