Spring 2018

Validating Key-value Based Implementations of the Raft Consensus Algorithm for Distributed Systems

Deepthi Vishwanath
San Jose State University

Validating Key-value Based Implementations of the Raft Consensus Algorithm for Distributed Systems

A Thesis
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfilment
of the Requirements for the Degree
Master of Science

By
Deepthi Vishwanath
May 2018

The Designated Thesis Committee Approves the Thesis Titled

Validating Key-value Based Implementations of the Raft Consensus Algorithm for Distributed Systems

By

Deepthi Vishwanath

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2018

Dr. Robert Chun	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research

ABSTRACT

VALIDATING KEY-VALUE BASED IMPLEMENTATIONS OF THE RAFT CONSENSUS ALGORITHM FOR DISTRIBUTED SYSTEMS

By Deepthi Vishwanath

Distributed systems are a group of systems connected via a network, all working towards achieving a common goal. To achieve fault tolerance and reliability, all the systems should work towards achieving consensus. Paxos is the most widely used consensus algorithm since 2 or 3 decades, but the shift is now happening towards a new algorithm known as Raft. Raft is a consensus algorithm (paper published in the year 2014) which is easier to understand and works like Paxos in terms of fault tolerance and performance. Since Raft is new, there is a need for a tool that verifies systems built using the Raft algorithm.

This project proposes the development of a tool that validates different key-value based implementations of Raft: Ckite, Libraft and Kayvee with each other, and compares the output to give a result if the implementations are correct or not. The validation is done based on various test cases that a distributed system is tested for.

Index Terms – **Consensus, Distributed systems, Evaluating implementations, Paxos, Raft.**

ACKNOWLEDGEMENT

First of all, I would like to express my deepest gratitude to my project advisor Dr. Robert Chun for all his help and guidance throughout this one year duration of this project. I would also like to thank my committee members Dr. Sami Khuri and Dr. Thomas Austin for their valuable suggestions and their time for reviewing my work.

I would like to say a big thank you to my husband, Kiran Ramamurthy, for his unending support, understanding and encouragement during my studies. I would also like to thank all my friends who have stood by me and motivated me throughout. Lastly, I would like to thank my parents for all their support and blessings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Achieving consensus is the most important aspect in a distributed system. Consensus is where all the systems in a distributed network agree upon a certain value or a certain course of action. The process of achieving consensus makes data consistent across all the systems and hence, one can be sure that the system always returns the correct value and not any stale value. There are algorithms that are implemented to make sure that a system arrives at consensus. The most widely used consensus algorithm is the one proposed by Lesley Lamport which is Paxos [10]. But, many researchers and developers found Paxos to be complicated to understand and not quite implementation friendly. Hence, a couple of engineers from Stanford University developed Raft [6], a consensus algorithm which is both easy to understand and implementation-friendly. Raft's architecture is very like that of Paxos [1].

There are many challenges when it comes to verifying a distributed system. Firstly, implementing a distributed system is not easy because these systems must handle reliability, concurrency, network failure and many other factors; systems can go down or network failures can occur at any point in time. Further, the system behavior is too complex to perform exhaustive testing and even the slightest bug or error in the system can cause it to crash, resulting in loss of critical data and information [2]. All processors arriving at a consensus is another major problem in distributed systems.

Solving consensus provides solution to many problems like:

- **Total order broadcast:** This is a type of broadcast where all the processors in a distributed system send or communicate with the same set of messages sent in the same order.
- **Atomic commit:** In an atomic commit, a set of instructions or operations that changes the state of a system are grouped and committed to the system as a single operation. If the commit fails, all changes are rolled back.
- **Terminating reliable broadcast**: This is another type of broadcast where, if a correct processor sends a message, then all other correct processors in the system should communicate using that same message.

As our interest is in Raft consensus algorithm, we hope to find answers to the following research questions in this project:

i. *Is Raft as easy, simple and efficient as its authors claim?*
ii. *What tools and approaches are used to evaluate the correctness of distributed systems in general?*
iii. *Can systems built using Raft, be efficiently tested by modifying and improving on the above approaches?*

Raft is gaining quick popularity and there are already almost 56 open-source implementations which is mentioned in just one collection [8]. These implementations are all different level of complexity and quality. The implementers mention it in their work about the validity and reliability of their systems and they are sometimes not sure what would happen when certain

conditions changed. Hence, it is important to develop a tool that verifies the various implementations of the Raft consensus algorithm. This is quite a bit of a challenge because each implementation is written in a different language and the tool that we construct should be able to verify majority of the implementations, if not all.

The rest of this work is organized as follows: Chapter 2 gives a brief overview of the Paxos algorithm and explains the problems and assumptions of it. Chapter 3 gives an overview of the Raft protocol and explains how its correctness has been verified. Chapter 4 explains the related work done in the field of tools for verifying distributed algorithms. In Chapter 5, we will see the design and implementation details of this project. Chapter 6 explains the results and Chapter 7 is the conclusion and future work of this project.

2. PAXOS – A BRIEF INTRODUCTION

Paxos is an algorithm used to solve the problem of consensus in a network of systems. It was proposed by Leslie Lamport, who is now known as the 'Father of Distributed Computing' and it was first submitted in 1990 and was published as a journal article in 1998 which is [9]. It is one of the oldest and most implemented distributed consensus algorithms to date. But it is widely regarded as complicated and difficult to understand and implement. Hence, L. Lamport wrote another paper titled 'Paxos made simple' [10] and a brief explanation of how the algorithm works is as follows:

2.1 Problems and Assumptions:

Paxos proposes that in a distributed network, all the systems are capable of proposing values and a single value should be chosen among the proposed values. The algorithm should ensure that no other value is chosen and, if no value is proposed, no value should be chosen.

The algorithm is based on the following assumptions:

1. The chosen value should be among the proposed values.
2. At the end of the algorithm, only a single value is chosen.
3. No process learns that a value has been chosen, unless a value has actually been chosen [11].

Each system in the network belongs to these three roles:

- Proposers – Propose a value.
- Acceptors – Decide which value to choose.
- Learners – Eventually learn the chosen value.

A single system can perform more than one role and they communicate with each other via messages. While implementing, the system is assumed to be asynchronous non-byzantine where each system operates at its own speed and the message delivery can take any amount of time to get delivered, be lost or duplicated but it cannot get corrupted.

2.2 The Paxos Algorithm:

When a value is proposed by any system, the steps the algorithm follows are shown in the flowchart Figure 1.

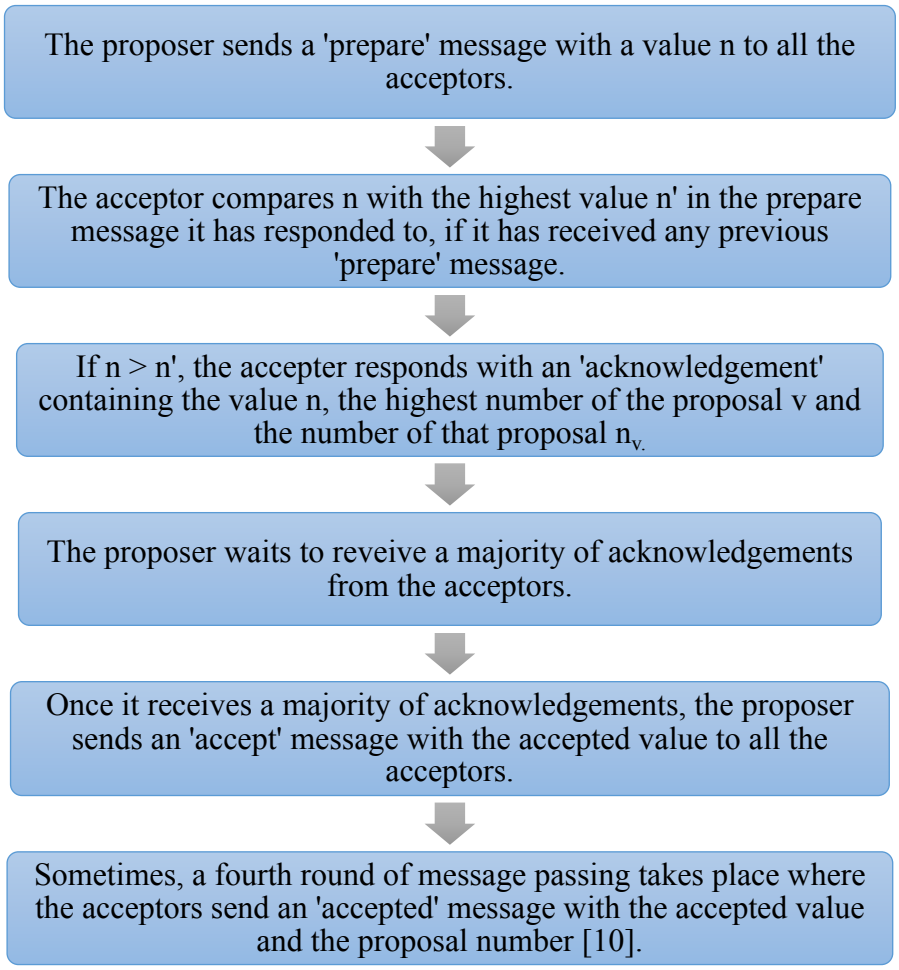

Figure 1. Paxos flowchart when a value is proposed

The system eventually reaches a consensus at the end of the algorithm as explained above. Though the algorithm might look simple on paper, it has been deemed difficult to understand and implement. This has led to the development of simpler consensus algorithms such as Raft.

The next chapter describes Raft in detail and gives an overview of its verification for correctness.

3. RAFT CONSENSUS ALGORITHM

3.1 What is Raft?

Raft is a consensus algorithm that was developed for managing replicated logs [6]. It is very important for a distributed system to arrive at consensus, because it ensures the fault tolerance properties of a system. This means that even if a few nodes in the network fail, the system will continue to operate, as long as a majority of the nodes are up and healthy. The main aim of Raft was to improve understandability and encourage easy implementation of the algorithm to build practical systems.

Each node in the network will always be in one of the three states:

- Follower – Each node starts as a follower and it only responds to remote procedure calls (RPCs) from the candidate or the leader. A follower maintains a timer known as *election timeout*. It starts the election process if it does not receive any RPC at the end of the *election timeout.*
- Candidate – A candidate is a node which initiates the leader election process.
- Leader – A leader is the node which gets elected as the leader.

The Raft algorithm proceeds in time known as *terms*. A *term* is an arbitrary length of time which starts with a leader election process. The node that gets elected serves as the leader for that term. Figure 2 explains the entire Raft algorithm in a nutshell.

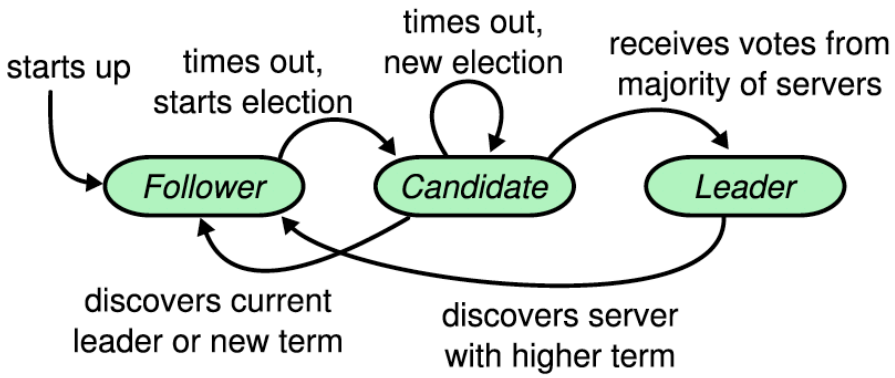

Figure 2. States of a node in the Raft algorithm [6]

Raft differs from Paxos mainly in its architecture. It is designed in such a way that it decomposes the consensus process to 3 phases:

1. **Leader election**

Raft uses a heartbeat mechanism to start the election process. Each node which starts in the *Follower* state waits until its timer *election timeout* expires and then sends a 'RequestVote' RPC to all the other nodes in the network. The follower node increments its term and transitions to become a *candidate.* The node remains in the candidate state until:

- It gets elected as a leader. Or,
- It gets notified of another node becoming a leader (The node then goes back to being a *follower*). Or,
- The term times out and no node has been elected as the leader [6].

Once a node is elected as the leader, it sends an RPC to all other nodes in the system to prevent any further elections. In Raft, the leader has all the responsibility to manage the replicated logs in the system. A distinguished leader is elected and if that leader goes down, the leader election

process starts once again. Once the leader receives the log entries from the clients, it routes it to the other nodes in the system and decides when it is safe to commit the logs [6].

2. Log replication

A node starts servicing the clients once it has been elected as the leader. The clients issue requests which contains commands which it needs executed by the replicated state machine. The leader first appends the request to its log entry and then sends an *AppendEntries* RPC to all the nodes in the system. The leader commits the entry in its state machine once a majority of the nodes have replicated the log entries in their state machines and responds to the client with the result. It then continues to issue *AppendEntries* RPC until all the nodes have the log entries replicated.

Each log entry consists of an *index* which identifies its position in the log entries. The leader keeps track of the highest *index* that has been committed and sends *AppendEntries* RPC with the appropriate *index* so that the nodes that are slow learn the current index term and make sure that every entry before that is committed before the current index is committed.

3. Safety

The most important property of Raft is the **State Machine Safety Property** as shown in Figure 3. This property states that, *L*

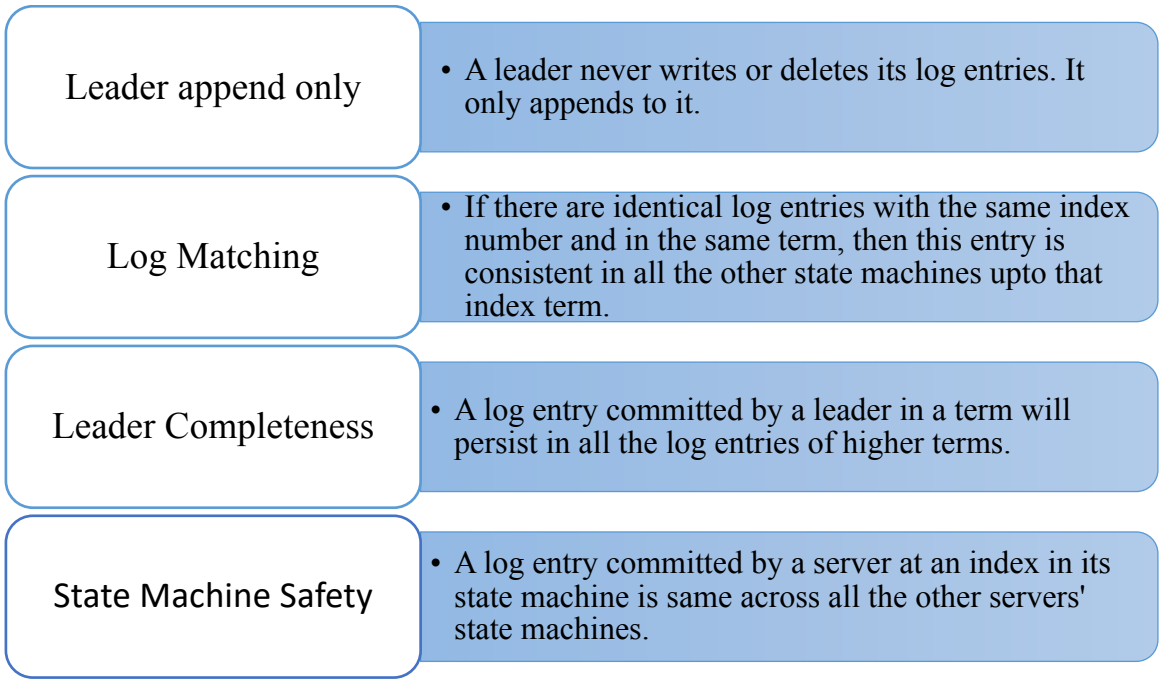

Figure 3. State Machine Safety Property [6]

The authors of Raft: Ongaro and Ousterhaut, mention about a study in their paper which was done on 2 different groups of students. In the study, the students were asked to read both Paxos and Raft and were asked questions about both the algorithms. Majority of the students easily answered questions on Raft, compared to questions on Paxos. The easy understandability of Raft was further verified by authors of [3], H. Howard et al. They mention in their paper that the high-level concepts of Raft were relatively easy to understand. The Raft protocol is designed by modularizing the different aspects of the protocol which improves the understandability. D. Ongaro's Ph.D. dissertation [19] further classifies the design decisions and provides a clear explanation of subtle details of the protocol.

The following section is aimed at verifying the correctness of Raft.

3.2 Verifying Correctness of Raft

H. Howard et al. [3], have done an extensive study of Raft and tried to validate its correctness. The steps they follow to verify Raft are as follows:

1. They first built an initial implementation of Raft to test the understandability of the protocol.
2. They then tested their implementation using an event-driven simulator.
3. Lastly, they calibrated and tested their results with the experimental setup given in the original paper of Raft [6].

The authors of [3] implemented the protocol using the same states as mentioned in the original Raft algorithm. They used a modular Ocaml-based approach to:

1. Separate the state-transitions of Raft.
2. Build trace checker and simulation framework, which are domain-specific front-ends.

The architecture of the simulator is as shown in Figure 4:

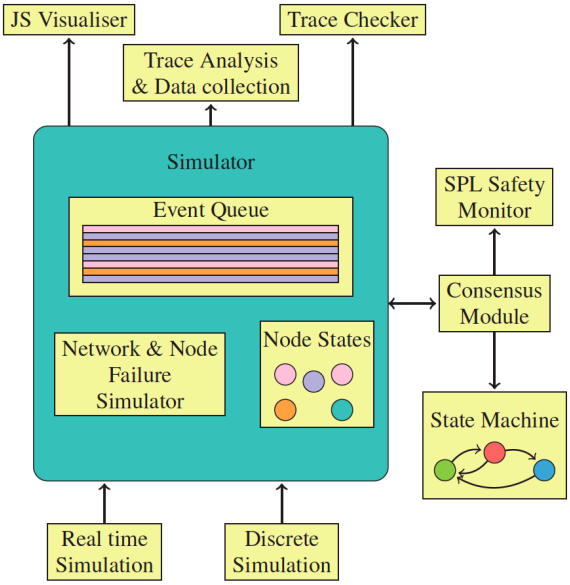

Figure 4. Overall architecture of the simulator [3]

The state transition models of Raft were encoded using the Statecall Policy Language (SPL) [4,5]. They used OCaml to build their simulator, instead of the original implementation of Raft which was done in C++ because OCaml provided static typing and a powerful module system. Their implementation could also test and detect bugs in the implementation.

To test the Raft protocol across diverse network environments, they built a message level network simulation function. They performed event-driven simulation, where the events occurred in a global priority queue and they were sequentially committed or applied on to the nodes. Each event contained information about the node it should run on, the time the execution should start and what operation should be performed. This would then decide the next state transition of the node.

After checking thousands of simulation traces and various tests, the authors found not one instance where the test results violated the results in the original Raft paper. Thus, they verified

the correctness of Raft and backed the claim of the original authors of Raft for easy understandability and easy implementable characteristics of Raft.

4. TOOLS FOR EVALUATING DISTRIBUTED SYSTEMS

Testing a distributed system is not an easy task because there are many factors that play a role like: node failure, network failure, external attacks, synchronization, deadlocks etc., and extensive and rigorous test cases should be written to identify any type of error that can occur. The following sections provide a literature review of tools and techniques that have been developed to test distributed systems in general, followed by the latest techniques that have been proposed to test systems built using Paxos and Raft.

4.1 Temporal Approach

A. Khoumsi, the author of [12] published in 2002, gives a temporal approach to testing distributed systems. He proposes *Implementation Under Test (IUT)*, which is nothing but generating the test cases and running them on the implementation of the system. But the main focus of the paper is on studying the *test execution phase* when IUT is distributed. Here the author discusses about the two main problems that occur while testing distributed systems: *Controllability and Observability* [12].

The solution the author proposes is that to resolve the above-mentioned problems, the test system should conform to certain timing constraints, even if the system is not real-time. These timing constraints are determined and applied during the test execution phase to improve the time it takes to run the tests and get results faster. This solution, though it might work for basic test cases, requires generating test cases for complex systems which can get complicated due to dependencies on the timing constraint.

4.2 Scenario-Based Object Oriented Testing

The authors of [13], W. T. Tsai, L. Yu, A. Saimi, published a paper in the year 2003 which proposes an Object-Oriented framework for testing distributed systems rapidly. It allows the programmer to adaptively test the implementation using scenario modeling, perform static analysis and dynamic simulation, remotely execute test cases via TCP/IP or SOAP protocols and finally, perform runtime verification.

The architecture of this framework follows three tiers:

1. **Front-end tier:** Here, the tester is provided a user-friendly graphics interface, where he can enter the test case based on the scenarios and also, view the final results.
2. **Middle-tier:** This tier organizes the test scenarios in an object-oriented fashion and creates the test case scenario objects. It then performs various types of analysis, executes tests and performs runtime verification and dynamic simulation. Access to the database is provided to store the test results.
3. **Target system tier:** In this tier, proxies which are nothing but the agents of the master, run tests on the target system and report the test results back to the master [13].

This approach looks very easy to use because the user need not write complex test cases as it gets automatically generated. The unanswered questions in the paper is how well it performs while testing complex systems and how easy is it to modify the tool to add a variety of test scenarios for a single test run.

4.3 Multi-Agent Framework for Testing

The authors of [14], H.F. Yamany, M. A. M. Capretz and L. F. Capretz propose a multi-agent framework for testing 3-tiered distributed systems. Agents are described as a piece of software that observe its surroundings through sensors and react using effectors. A 3-tier distributed system consists of server, middleware and various clients. A server contains all the data repository and the middleware is the connection between the server and the clients. Figure 5 shows a 3-tier distributed system.

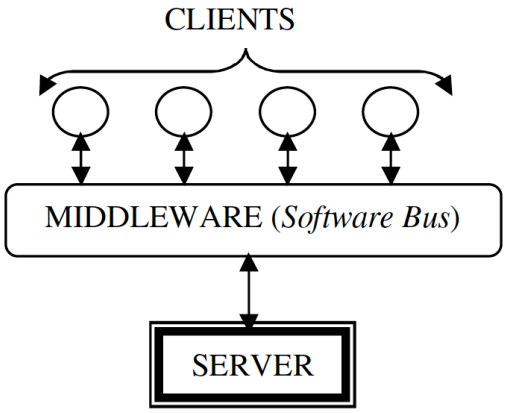

Figure 5. 3-tier distributed architecture [14]

The architecture of this framework is based on the following properties of an agent:

1. Autonomy: During its runtime, agents automatically monitor the distributed systems.
2. Intelligence: Agents generate fresh test cases and reveal most bugs that might occur by running a variety of testing techniques.
3. Flexibility: The conditions in a distributed system are continuously changing as many resources are being accessed simultaneously. The agents adapt well to this condition and change the test approach based on the current condition of the system.

4. Cooperation: Agents communicate among themselves and with the tester to ensure that the test cases are running fine.
5. Reactivity: The agents correct the system as soon as an error is found the system.

This approach looks promising because it is adaptive to the changing conditions in the system and generates automatic test cases to suit the conditions. But, a possible bottleneck can be when it comes to the cost and performance of this tool. A detailed performance analysis could have shed more light on this aspect to decide on its feasibility.

4.4 Verdi – A Framework for Verifying Distributed Systems

Verdi [2] is a framework built by J. R. Wilcox et al., to help programmers build a fault-tolerant and reliable distributed system. This framework implements a distributed system and verifies the implementation of that system.

A code that is tested often diverges from the real-world implementation. Hence, Verdi overcomes this by supporting testing of the real working code. An initial implementation of a system can first be tested using Verdi. Going further, parameters can be changed to match the real-life scenario and tested again to verify the correctness. Hence, Verdi is flexible enough to allow different implementations of the same system to be tested.

This framework is built upon the following 3 main ideas:

1. Verdi provides a toolchain named Coq to write executable distributed systems.

2. As mentioned above, it allows the user to tweak parameters in a system to match a fault model.
3. Verdi works on a *compositional model* i.e., separates the correctness of the system from fault tolerance. The compositional model works on a concept called *verified system transformer.* This means, given a set of input parameters, the framework assumes certain output parameters and provides a new implementation of the system as output.

Hence, Verdi can be used to verify the mutex property of the lock service of an application and safety properties of an application effectively.

Verdi is a promising tool for verifying distributed systems from end to end. The concept of verified system transformer presented in Verdi allows the user to visualize the system for a variety of parameters. But, one disadvantage is that this tool has a complicated architecture and a difficult learning curve. The users must spend some considerable amount of time to learn to use it effectively and tweak its parameters based on the needs.

4.5 Coracle – Evaluating Consensus

Coracle [7] by H. Howard and J. Crocroft is a tool that verifies distributed consensus and claims to do it at settings that are closer to realistic deployments. Coracle also claims that consensus algorithms like Raft and Paxos face an availability issue when deployed in scenarios closer to real-life.

Coracle is designed as an event-based simulator which uses state machine replication to verify the correctness of a system. It allows the users to select from a variety of network environments or even allows the user to build a new network. These are then stored in the framework to allow rerun of traces to check the safety and performance metrics. Also, it allows the user to select the best parameters for a particular deployment [7].

The advantage of this approach is that it allows the user to play with many parameters and test the system for all possible conditions. As this tool is very generic, it allows testing of entirely new algorithms against this test framework. A possible outline of a performance analysis of the tool when run against some famous algorithms would have rendered the tool more credible.

4.6 Test-Case Dependency Architecture

The authors A. Morroquin and D. Gonzalez of [15] propose a distributed testing architecture based on test-case dependencies to verify the conformance of distributed systems in a black-box context. They use the *European Telecommunication Standards Institute* and *Test Description Language standard* to test their systems. They use conformance testing which aims at detecting errors and abnormalities of the System Under Test (SUT) with regards to the standards [15]. They identify controllability, non-determining nature of the system and synchronization as the main problems to test distributed systems but, another major issue which no other paper identifies is the difficulty in correlating all the test results from different components.

They formally define the dependencies for test cases for testing complex systems and focus on logical dependencies. The final validation is done in a black box context, which is testing it on

real-time communication systems. They also provide a generic test framework on which the tests are executed and the results are collaborated to provide the final verdict.

The plus point of this approach is that it does a black box testing which is closer to the real-life implementation. But, a major drawback is that it vests a lot of responsibilities on the tester. The tester is responsible for writing all the test cases and defining the dependencies. An error from the tester can render the whole tool useless.

4.7 Flotsam – Evaluating Implementations of Raft

Flotsam [8] by C. Gilbert is a tool which works by comparing various systems implemented using Raft against one another and compares their output to verify the systems. This tool currently works only on the key-value store implementation of systems. The protocol is executed in a virtualized Docker environment and the configurations of the system to be tested are specified in a special Dockerfile. There is a test generator in place, that generates and runs test plans for each implementation of the system. Writes and reads are performed during the test plan execution and once all the tests are complete, the test results are verified for any inconsistencies.

Flotsam is the latest tool which is designed to test distributed systems built using Raft. It uses the latest technologies like Docker to run the tests. Using Docker is an advantage here because the Raft implementations can be downloaded as an image with all the runtime environments encapsulated in the image. This helps in easy plugin of the implementations to do the tests. This tool only works on a basic key-value implementation of a system and hence, making it work on different types of implementations can prove the tool to be more powerful.

5. DESIGN AND IMPLEMENTATION

The main idea for the implementation is to compare the execution of different Raft implementations. As mentioned in the official website of Raft, [17], there are many different implementations of the Raft algorithm. These implementations are written in different languages and most of them are key-value implementations of Raft. For some, the type of implementation is not specified. For this project, we chose only the implementations that had a key-value based implementation because most implementations belonged to this category and it is the most developed type of distribution.

Key-value implementation is a storage method designed to store and query the data (value), based on a key. Each key is a unique value that is associated with the corresponding value. Most of the modern systems are designed using this model and many of the databases like DynamoDB, Oracle NoSQL etc., are internally key-value stores. Among the many implementations of Raft, we chose Literaft, Kayvee and CKite implementations.
Literaft is a javascript implementation, Kayvee is part of a Java implementation of Raft known as Libraft and CKite is a JVM implementation of Raft written in Scala. All these implementations are consistent and distributed.

Each of these implementations are downloaded as Docker images. A brief overview of Docker is as follows:

5.1 Docker

Docker is the world's leading containerization software platform. Containerization is nothing but virtualization at the operating system level. Docker produces components known as containers which are light-weight and which encapsulates an entire environment for a program to be run. It can be run on any system to instantly set up all the necessary environments required for that program.

Docker is gaining huge popularity nowadays. Until recently, it was the era of virtual machines, which allowed one to mount an entirely new OS on top of our existing base system, which, in turn, would share the system configurations on the underlying base system (As shown in Figure 6). These virtual machines eventually become very heavy for the host system to handle. But, Docker solves this problem by using shared operating system. This means, it is more efficient in using host system resources and allows many more applications to run on a single machine than which would be possible with the use of virtual machines.

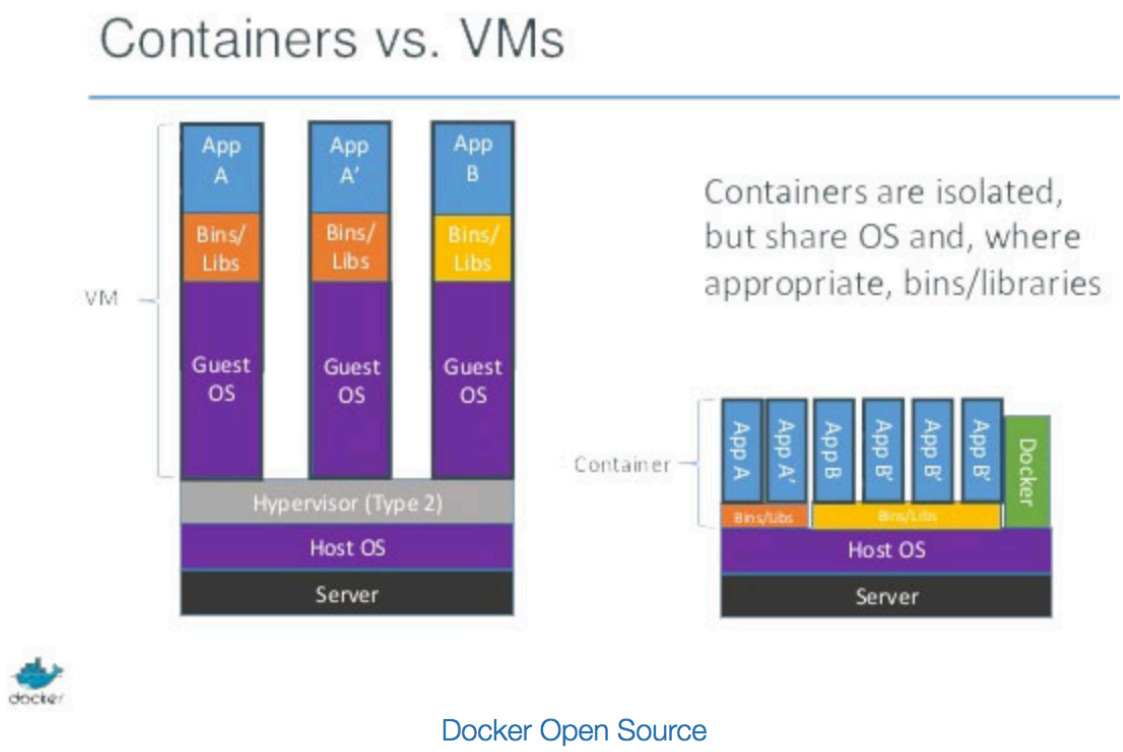

Figure 6. Difference between VMs and Docker containers [18]

In this project, we use Docker to download the Key-value store implementations of Raft as Docker images.

5.2 Workflow

This project is implemented on an Ubuntu 14.04 virtual machine using the Trusty image. An entire virtual machine was dedicated just for this project. A memory of 8 GB has been dedicated to this machine and it is mounted on a base machine which is a MacBook of 16 GB memory.

This project is divided into the following components as shown in Figure 7.

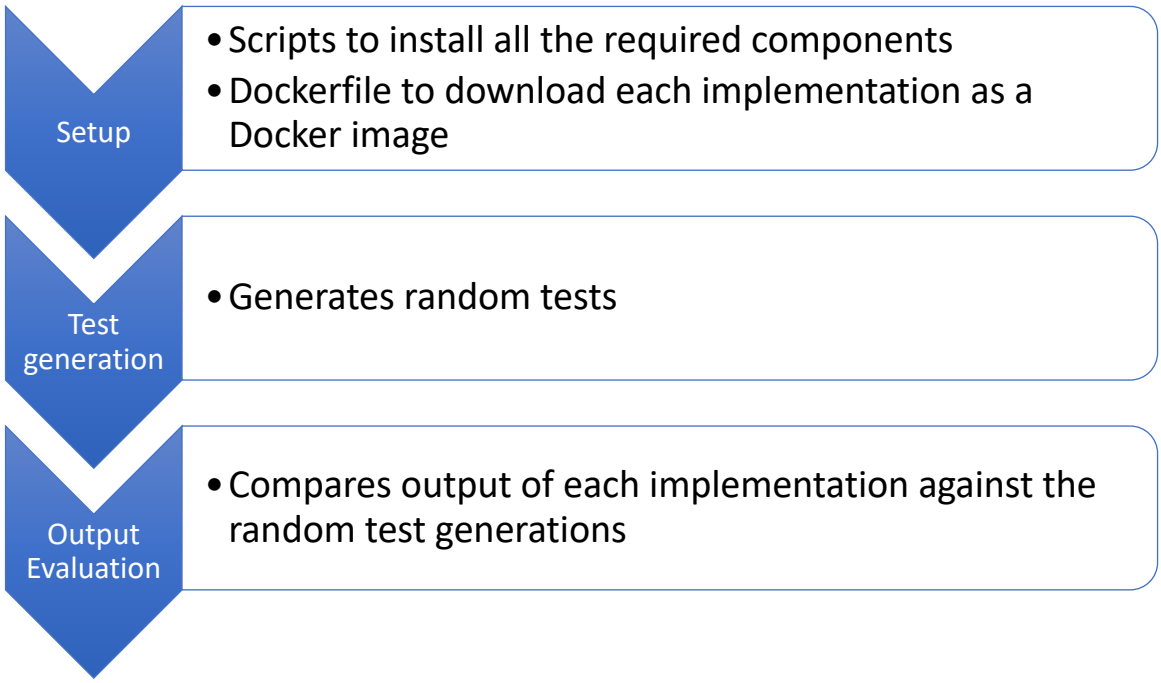

Figure 7. Project components and their functionality

5.2.1: Setup

The Setup consists of the scripts for the following purposes:

- To install the latest Docker version – using version 1.12.6 in this project.
- As this entire project is coded in Python2.7, scripts to install pip and other Python dependencies.
- To setup a domain name forwarder for Docker
- To build and download Docker images based on the Dockerfile.
    - A Dockerfile is a script in which the environment to be setup is specified with all the links provided to build the image and the dependencies.

5.2.2: Test Generation

A typical distributed system should be tested to check the following basic conditions:

- Kill a node and revive it.
- Kill the leader node.
- Route a request to a node that has just been revived.
- Create network partition
- Heal network partition

The test generation script generates above tests to test the implementations of Raft. These test scripts are run against each Raft implementation that we downloaded as a Docker image.

The result of the test generation module is passed on to the output evaluation module to compare the outputs.

5.3.3 Output Evaluation

The Output Evaluation is a comparator that compares the results obtained from running the tests on the Raft implementations. For a key-value store implementation, the output will be a log of activities that occurred across the cluster.

Hence, we designed this module to compare the results of each implementation to test for any mismatch or discrepancy. If the log entries across the implementations under test match, the results return that the components tested conforms to the protocol. If not a match, then

appropriate messages will be printed to inform the user which component failed and what might have happened.

6. RESULTS

As mentioned earlier, this project is executed in 3 stages. First, the setup is run which gets the entire environment ready. All the required Docker images of the Raft implementations are downloaded as specified in a Dockerfile.

Each test case was executed at least 10 times to record any anomalous behavior such as inconsistent reads/writes that might occur.

Test Case 1: Kill a node and revive it

In this test case, the test simulates the failure of a node to observe how the system behaves. When a node fails in a system, it will not be able to respond to any requests. The goal of this test case is to check how each of the implementations handle when a node failure occurs and after it has been revived.

The test plan for Test Case 1 is as shown in Figure 8.

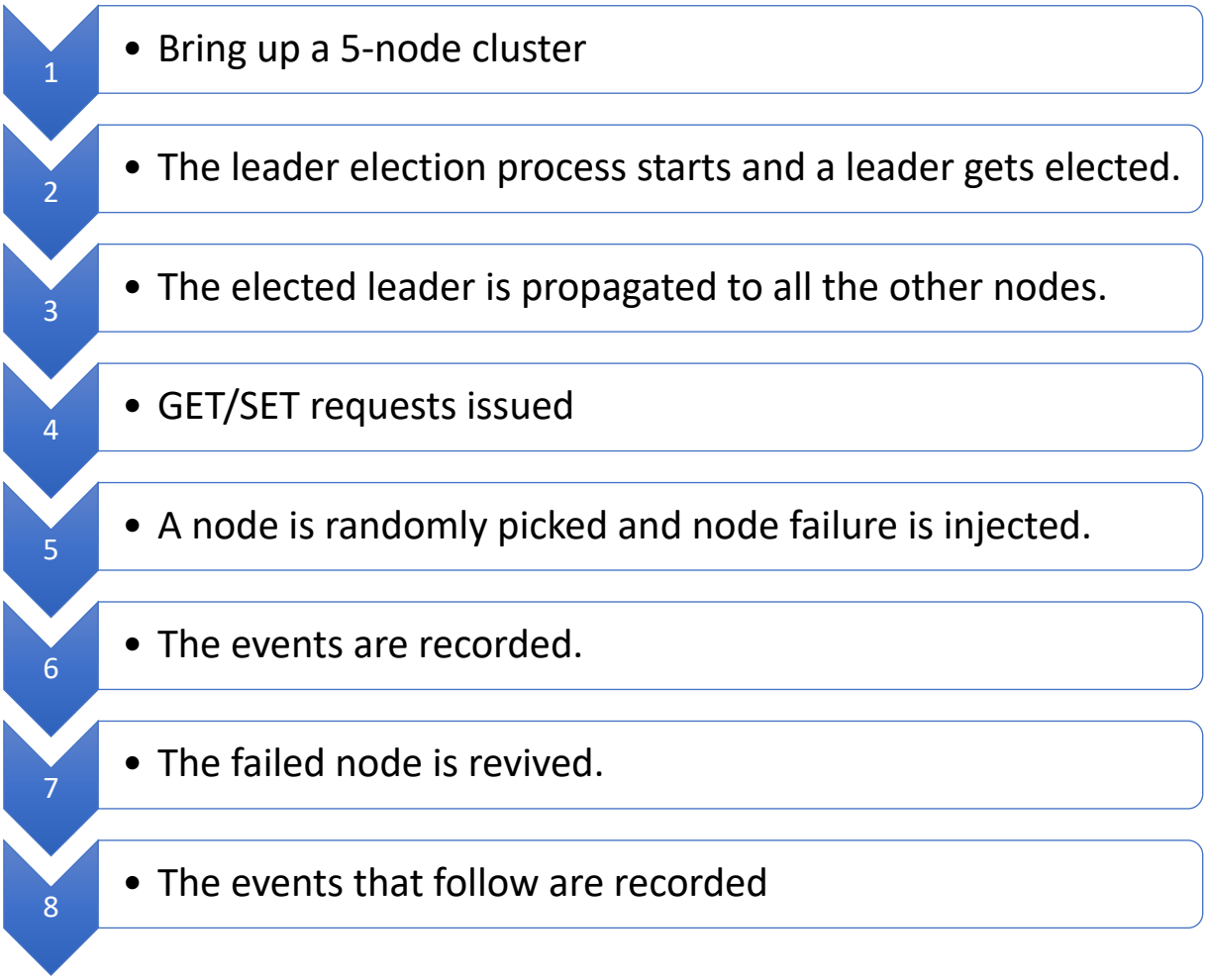

Figure 8. The test plan for Test Case 1

The events that occur and the logs collected in each implementation are shown in Table 1.

Table 1 The events that occur in Test Case 1

	SET (key0,10)	SET (key1,15)	FAIL (node)	GET (key1)	SET (key3,20)	REVIVE (node)	GET (key3)
Lite-raft	n3:10	n3:15	n1 fail	n2:15	n3:20	n1 revive	n1:20
Kayvee	n1:10	n1:15	n3 fail	n4:15	n1:20	n3 revive	n3:20
CKite	n2:10	n2:15	n4 fail	n3:15	n2:20	n4 revive	n1:20

This test was executed on all the implementations: Literaft, Kayvee and CKite. As we can see in Table 1, the leader node handles the SET requests. Once the SET request is committed on

majority of the nodes, the leader node commits the log and sends a reply to the client. These implementations reported the event of node failure in their logs and the following GET/SET requests were not propagated to this failed node. When the node was revived, the traffic again started propagating to this node to fulfil requests, achieving consistency. As seen, each of these events were reported and the Output Evaluator compared the GET/SET results after and before the node failure injection. There were no inconsistencies recorded in any values for all the implementations.

Test Case 2: Kill a leader node

In this test, the failure of a leader node is simulated and the events that occur are recorded. The test plan for Test Case 2 is as shown in Figure9.

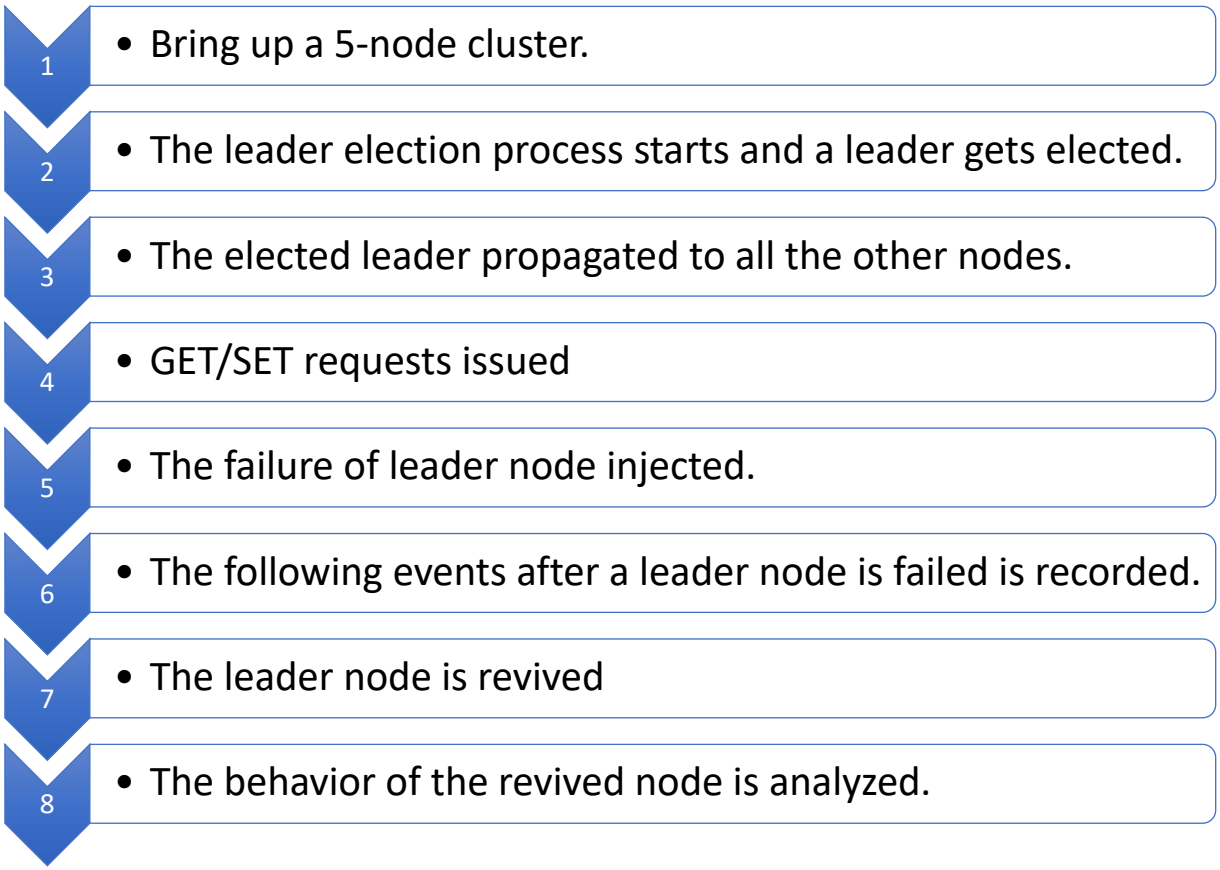

Figure 9. The test plan for Test Case 2

The events that occur and the logs collected in each implementation for Test Case 2 are shown in Table 2.

Table 2 The events that occur in Test Case 2

	SET (key0,10)	SET (key1,15)	FAIL (L-node)	GET (key2)	GET (key2)	SET (key1,15)	REVIVE (node)	GET (key1)	SET (key3,20)
Lite-raft	n3:10	n3:15	n3 fail	Leader elec.	n2:15	n5:20	n3 revive	n1:20	n5:20
Kayvee	n1:10	n1:15	n1 fail	Leader elec.	n4:15	n3:20	n1 revive	n3:20	n3:20
CKite	n2:10	n2:15	n2 fail	Leader elec.	n3:15	n1:20	n2 revive	n1:20	n1:20

This test was executed on all the implementations: Literaft, Kayvee and CKite. According to the algorithm, when a leader fails and when the other nodes in the cluster do not receive any communication from the leader for a pre-defined time, the other nodes (followers) initiate a leader election process and become candidates to the leader election.

As shown in Table 2, once a leader node failure is injected, a leader election process is triggered in all the implementations and a new leader is elected. This new leader now elected serves all the client requests. When a previously leader node is revived, it should no longer be a leader and this is confirmed by the last SET(key3,20) operation, which is served by the newly elected leaders. All these events were aptly recorded in the logs and the Output Evaluator verified the results and reported no inconsistencies.

Test Case 3: Route a request to a node that has just been revived

In this test case, the test simulates an event where a GET request is sent to a node that is reviving from failure. When a reviving node is issued a request, it should either forward the request to the leader node or respond with a message equivalent to "Not a leader".

The test plan for Test Case 3 is as shown in Figure 10.

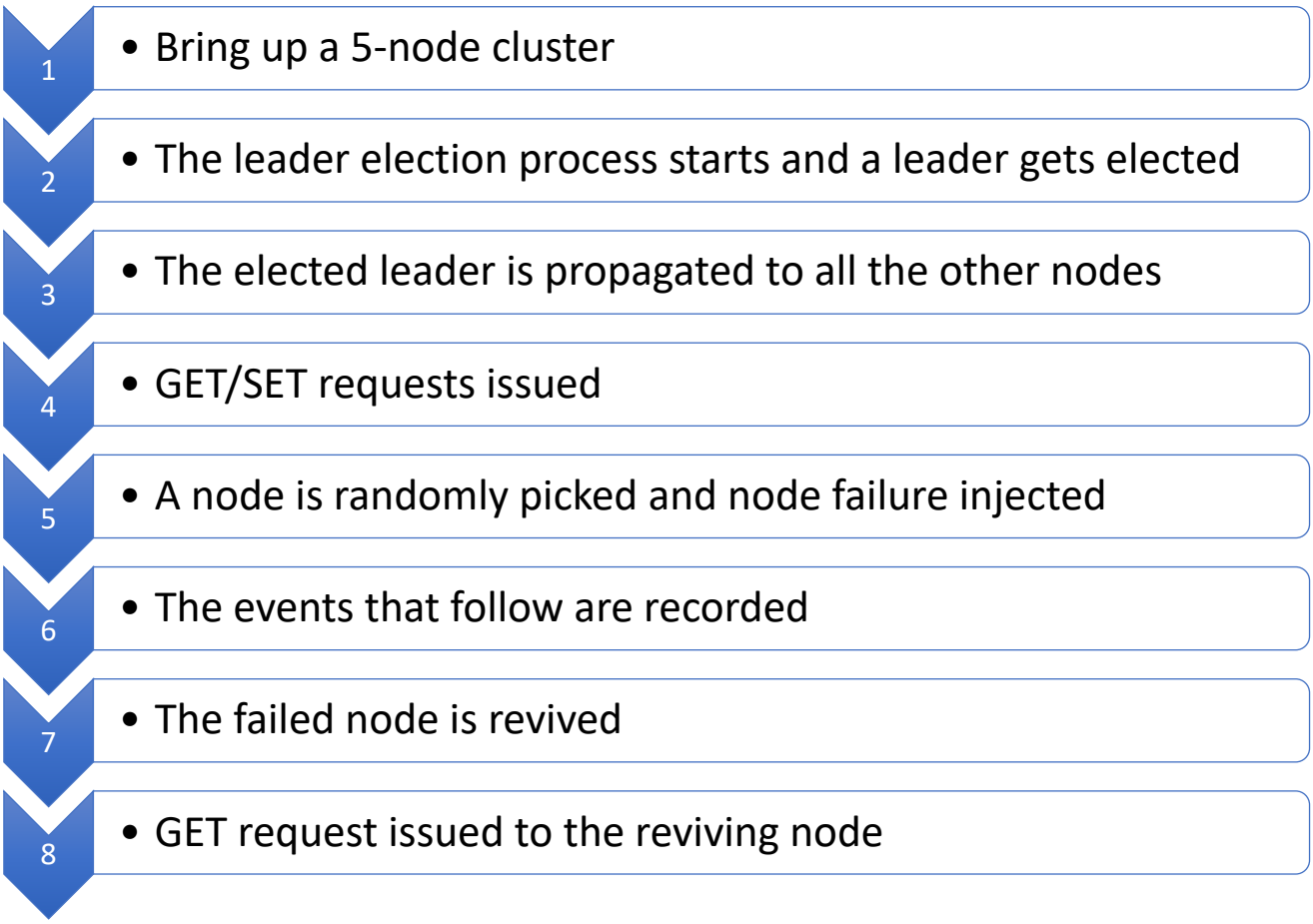

Figure 10. The test plan for Test Case 3

A sample set of events that occur and the logs collected in each implementation are shown in Table 3.

Table 3 The events that occur in Test Case 3

	SET (key0,10)	SET (key1,15)	FAIL (node)	GET (key1)	SET (key2,20)	REVIVE (node)	GET (key1)
Lite-raft	n2:10	n2:15	n1 Fail	n3: 15	n2:20	n1 revive	n3:15
Kayvee	n3:10	n3:15	n2 Fail	n1:15	n3:20	n2 revive	n4:15
CKite	n4:10	n4:15	n3 Fail	n5:15	n4:20	n3 revive	n3: null

This test was implemented on all the implementations: Literaft, Kayvee and CKite. The Literaft and Kayvee implementations did not serve this GET request, according to the expected behavior. But, the CKite implementation arbitrarily reported a NULL value for the request, which denotes that the value for the KEY in the request is not set. This behavior was noticed only once. It was not possible to re-create this condition again because such an anomalous behavior depends on a variety of factors such as network connectivity, network speed etc. Otherwise, even the CKite implementation successfully denied the request. These events were all recorded and the Output Evaluator verified the results to report consistent / inconsistent results.

Test Case 4: Create and heal network partition

In this test, a network partition is simulated. Figure 11 shows a cluster of 5 nodes before the network partition occurs.

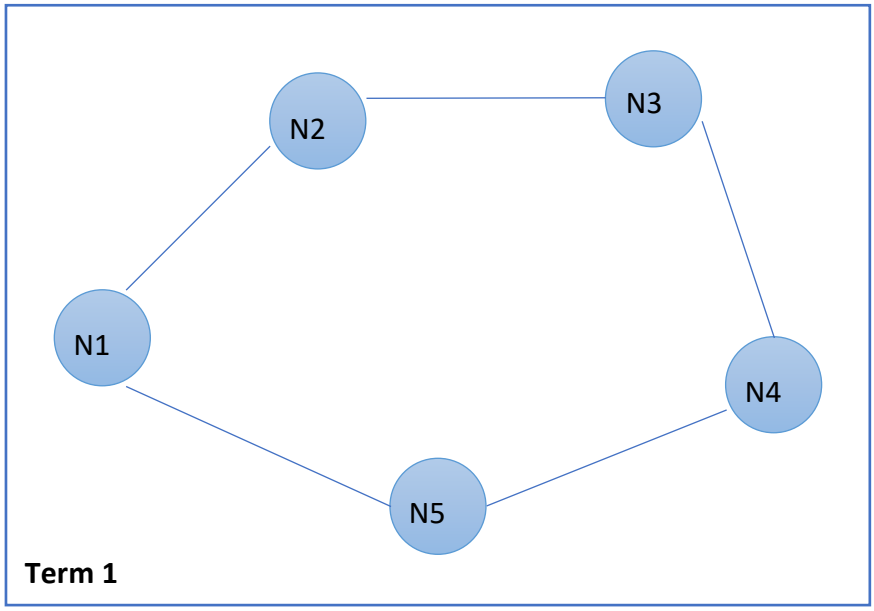

Figure 11. A cluster before network partition

Figure 12 depicts the network partition. Here, the system is partitioned to contain 3 nodes in 1st partition and 2 nodes in the 2nd partition, with leader in the 1st partition. The purpose of this test case is to understand how the system behaves under network partition and how consistency is achieved.

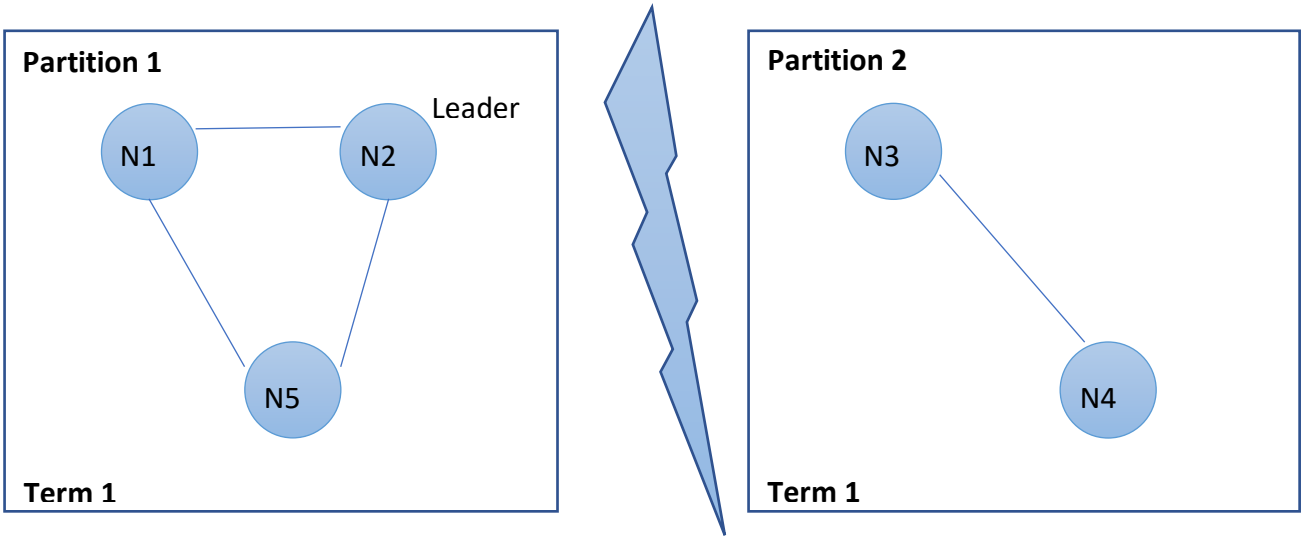

Figure 12. A cluster divided into 2 partitions after network partition

The test plan for Test Case 4 is as shown in Figure 13.

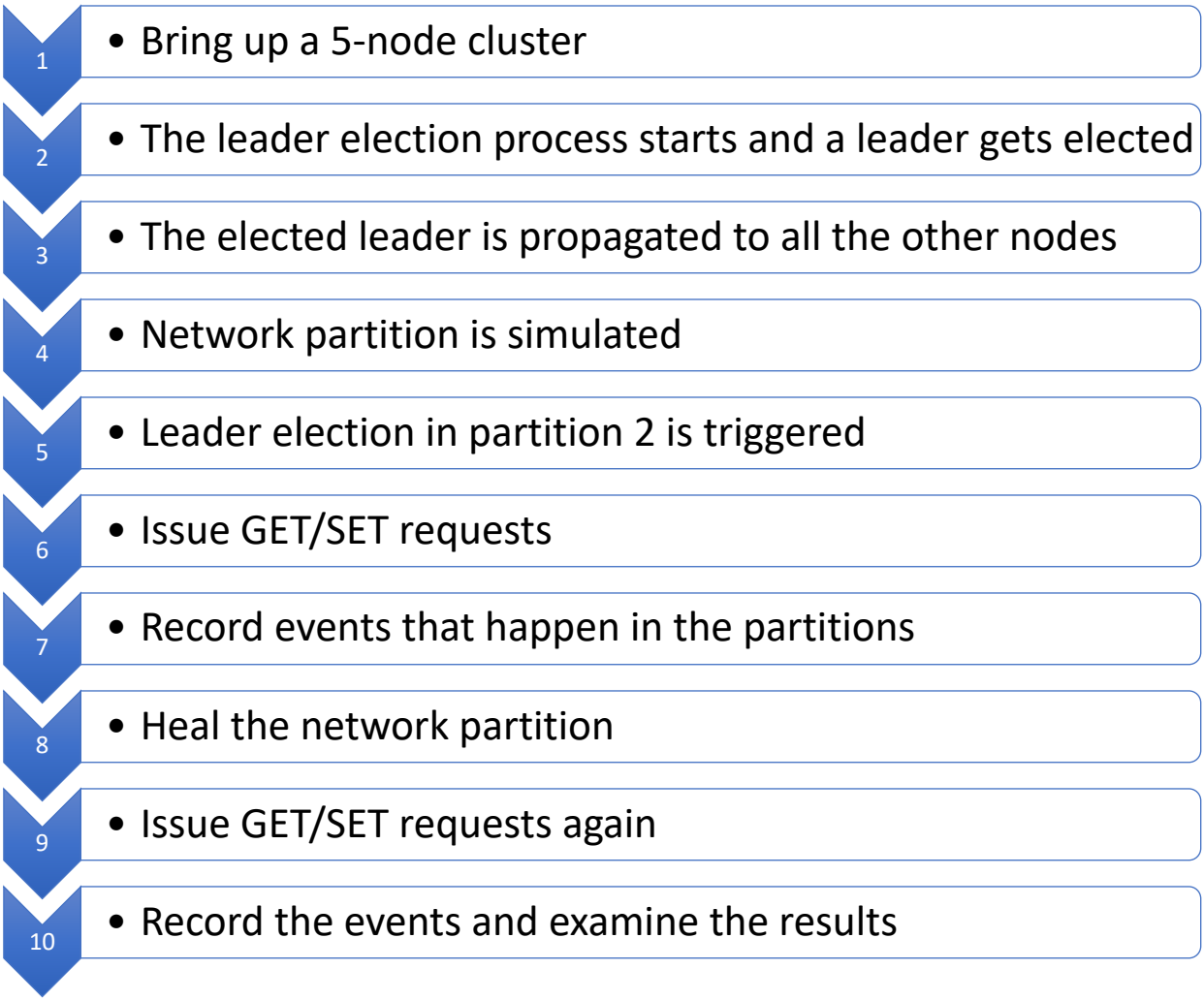

Figure 13. The test plan for Test Case 4

This test was executed on all the implementations: Literaft, Kayvee and CKite.

When there is network partition, each partition will have to have leader nodes and hence, a leader election happens even in partition 2 and a node gets elected as the leader. When the algorithm

began execution, all the nodes were in Term 1. Due to a second leader election, now all nodes in partition 2 will move to Term 2 as shown in Figure 14.

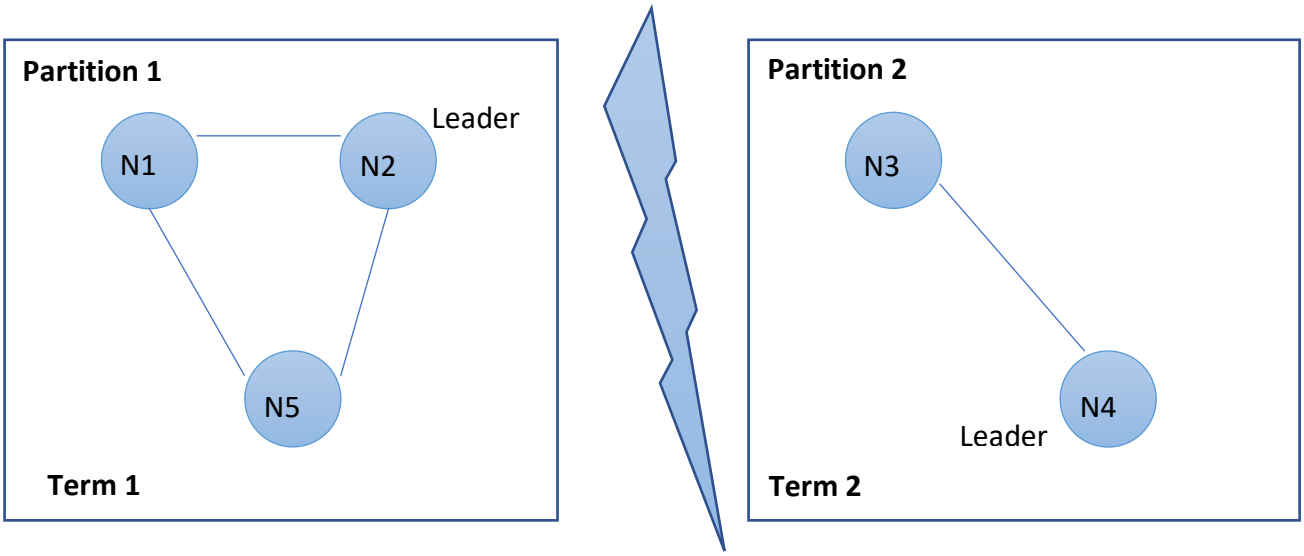

Figure 14. Leader election in partition 2

These events are reported in the log of each node. Now, any SET requests to partition 2 will not get committed because of lack of majority (To achieve majority, at least 3 nodes are required). Hence, even after a series of SET requests, the result of the nodes in partition 2 will be null because the leader node cannot commit those values to the state machine.

A sample set of events that occur and the logs collected in each implementation for each partition when the network is partitioned are shown in Table 4.

Table 4 The events that occur in Test Case 4 when the network is partitioned

PARTITION 1					
	SET (key0,10)	**SET (key1,15)**	**GET (key1)**	**SET (key2,20)**	**GET (key0)**
Lite-raft	n2:10	n2:15	n1: 15	n2:20	n5:10
Kayvee	n1:10	n1:15	n2:15	n1:20	n4:10
CKite	n5:10	n5:15	n1:15	n5:20	n2: 10
PARTITION 2					
	SET (key0,10)	**SET (key1,15)**	**GET (key1)**	**SET (key2,20)**	**GET (key1)**
Lite-raft	n3: null	n3: null	n4: null	n3: null	n4: null
Kayvee	n4: null	n4: null	n3: null	n4: null	n3: null
CKite	n4: null	n4: null	n3: null	n4: null	n3: null

Once the network partition is healed, since partition 2 has the higher term, the leader of the cluster will now be the leader elected for the entire network, as shown in Figure 15.

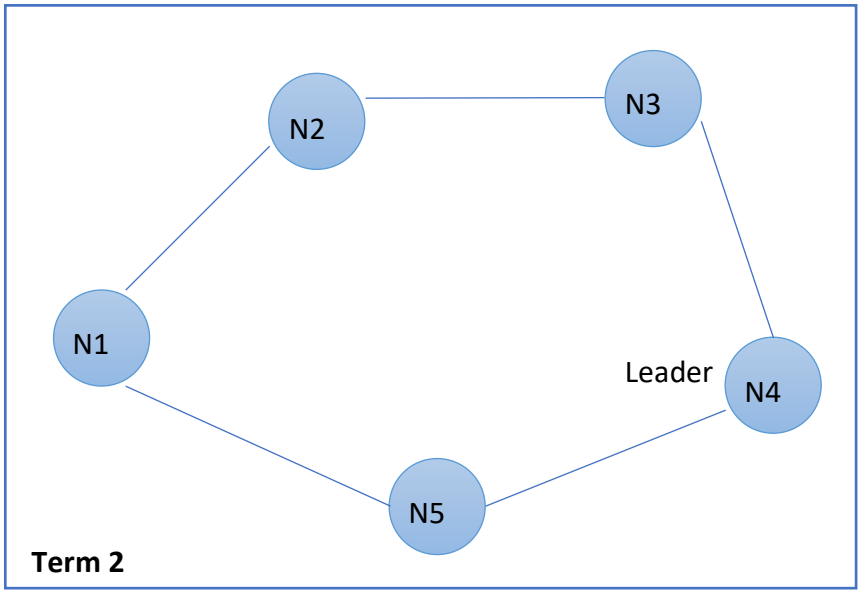

Figure 15. A cluster after the network partition is healed

Now, when GET/SET requests are issued, all the nodes have values updated in the result.

A sample set of events that occur and the logs collected in each implementation for each partition when the network is partitioned is shown in Table 5.

Table 5 The events that occur in Test Case 4 when the network partition is healed

	SET (key0,10)	SET (key1,15)	GET (key1)	SET (key2,20)	GET (key0)
Lite-raft	n2:10	n2:15	n1: 15	n2:20	n5:10
Kayvee	n4:10	n4:15	n2:15	n4:20	n3:10
CKite	n5:10	n5:15	n3:15	n5:20	n1: 10

For this test case, all the 3 implementations worked as expected and provided consistent results.

7. CONCLUSION AND FUTURE WORK

Paxos was the most widely used consensus algorithm for the last 2 decades but now, Raft is catching up and is almost replacing Paxos. Many companies now use Raft in the implementation of their systems. As seen in Chapter 4 on related works, there has been many tools in place that verify the implementation of the distributed systems, but there is still a lot of research going on for building a tool that verifies systems built using Raft algorithm. Testing a distributed system has never been easy. Shifting to a new algorithm, such as Raft, from a tried-and-tested algorithm like Paxos is a huge undertaking and new methods and ways must be found to test such systems.

In this project, we have implemented a tool that tests various Raft implementations (only key-value store implementations) by comparing the outputs of the tests run on them. We have defined all the basic test cases that a distributed system should be verified for and reported the results. The results show that the implementations Libraft, Kayvee and CKite, passed all the test cases, except for 1 test run that CKite failed in Test case 3. The use of Docker to obtain Raft implementations makes it easier to plug new Raft implementations to test. Hence, one of the advantages is the easy usability of the tool. This tool is helpful in validating new Raft implementations that anybody will develop in the future and to confirm if it is working as expected.

This tool can further be extended to implement the concept of time (clocks) to test the simultaneity of events and, to understand and detect the dependencies between events. As we did not have access to multiple distributed clusters, generating performance evaluations were not

very helpful. All the numbers came up to be similar because all our nodes resided on the same cluster. Hence, this tool can be executed on multiple distributed clusters to get performance analysis of the implementations in each phase, like, the leader election phase, log replication phase etc. This will help to understand which component of the implementation can be optimized.

As possible extensions to this work, more complex test cases can be written to identify synchronization bugs. These bugs are very subtle and difficult to solve. Hence, future work in this direction can prove useful.

REFERENCES

[1] P. Uthamarajan, "Analysis of Systems Using Distributed Consensus Algorithms", M.S. Thesis, C.S, Texas A&M Univ., Kingsville, TX, 2017, [Online]. Available: https://search-proquest-com.libaccess.sjlibrary.org/docview/1946683768?pq-origsite=primo.

[2] J. R. Wilcox, D. Woos *et al.,* "Verdi: A framework for implementing and formally verifying distributed systems", *Proceedings of the 36th ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation,* Portland, OR, 2015, pp. 357-368 [Online]. Available: https://dl.acm.org/citation.cfm?id=2737958

[3] H. Howard, M. Schwarzkopf *et al.,* "Raft refloated: do we have consensus?", *ACM SIGOPS Operating Sys. Review – Sp. Issue on Repeatability and Sharing of Experimental Artifacts,* Vol. 49, 2015, pp. 12-21, [Online]. Available: https://dl.acm.org/citation.cfm?id=2723876

[4] A. Madhavapeddy, "Creating high-performance statically type-safe network applications", Ph.D. thesis, Univ. of Cambridge, Cambridge, TN, UK, 2006, [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-775.pdf

[5] A. Madhavapeddy, "Combining static model checking with dynamic enforcement using the statecall policy language", in *Proc. 11th Intl. Conf. on formal engineering methods: formal methods and software engineering*, 2009, pp. 446–465, [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-10373-5_23

[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm", in *USENIX Annual Technical Conf.*, Philadelphia, PA, 2014, pp. 305-320, [Online]. Available: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

[7] H. Howard, J. Crocraft, “Coracle: evaluating consensus at the internet edge”, in *SIGCOMM '15 Proc. of the 2015 ACM Conf. on Spl. Interest Grp. on Data Comm.,* pp. 85-86, [Online]. Available: https://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p85.pdf

[8] C. Gilbert, “Flotsam: evaluating implementations of the raft consensus algorithm”, unpublished, [Online]. Available: http://www.scs.stanford.edu/14au-cs244b/labs/projects/flotsam.pdf

[9] L. Lamport, “Part-time parliament”, *ACM Transactions on Comp. Systems (TOCS),* vol. 16, no. 2, pp. 133-169, May 1998, [Online]. Available: https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf

[10] L. Lamport, “Paxos made simple”, Nov. 2001, unpublished. [Online]. Avalilable: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf.

[11] L. Lamport, “New approach to proving the correctness of multiprocess programs”, in *ACM Trans. of Prog. Lang. and Systems (TOPLAS),* vol. 1, no. 2, pp. 84-97, Jul. 1979, [Online]. Available: https://lamport.azurewebsites.net/pubs/new-approach.pdf

[12] A. Khoumsi, "A temporal approach for testing distributed systems," in *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1085-1103, Nov 2002, [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1049406

[13] W. T. Tsai, L. Yu, A. Saimi and R. Paul, "Scenario-based object-oriented test frameworks for testing distributed systems," *The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, 2003. FTDCS 2003. Proceedings.*, 2003, pp. 288-294, [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1204349

[14] H. F. El Yamany, M. A. M. Capretz *et al.*, "A Multi-Agent Framework for Testing Distributed Systems," *30th Annual International Computer Software and Applications*

Conference (COMPSAC'06), Chicago, IL, 2006, pp. 151-156, [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4020160

[15] A. Marroquin, D. Gonzalez and S. Maag, "Testing distributed systems with test cases dependencies architecture," *2015 7th IEEE Latin-American Conference on Communications (LATINCOM)*, Arequipa, 2015, pp. 1-6, [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7430116

[16] C. Oliveira, L. C. Lung, H. Netto *et al.,* "Evaluating raft in docker on kubernetes", *ICSS* 2016, pp. 123-130, [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F978-3-319-48944-5_12.pdf

[17] D. Ongaro. Raft consensus algorithm. [Online]. Available: https://raftconsensus.github.io/

[18] [Online]. Available: https://www.sdxcentral.com/cloud/containers/definitions/what-is-docker-container-open-source-project/ [Last retrieved on March 20, 2018]