Model Checking Guided Testing for Distributed Systems

Dong Wang* Wensheng Dou*† Yu Gao Chenao Wu Jun Wei† Tao Huang
State Key Lab of Computer Science, Institute of Software Chinese Academy of Sciences
University of Chinese Academy of Sciences
{wangdong18,wsdou,gaoyu15,wuchenao20,wj,tao}@otcaix.iscas.ac.cn

Abstract

Distributed systems have become the backbone of cloud computing. Incorrect system designs and implementations can greatly impair the reliability of distributed systems. Although a distributed system design modelled in the formal specification can be verified by formal model checking, it is still challenging to figure out whether its corresponding implementation conforms to the verified specification. An incorrect system implementation can violate its verified specification, and causes intricate bugs.

In this paper, we propose a novel distributed system testing technique, *Model checking guided testing* (*Mocket*), to fill the gap between the specification and its implementation in a distributed system. Specially, we use the state space generated by formal model checking to guide the testing for the system implementation, and unearth bugs in the target distributed system. To evaluate the feasibility and effectiveness of Mocket, we apply Mocket on three popular distributed systems, and find 3 previously unknown bugs in them.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Model checking*.

Keywords: Distributed system, model checking, testing

ACM Reference Format:
Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3587442

*Equal contribution. Wensheng Dou is also affiliated with Chongqing School, University of Chinese Academy of Sciences.
†Wensheng Dou and Jun Wei are the corresponding authors.

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3587442

1 Introduction

Nowadays distributed systems have become pervasive, and play an important role in modern society. Different kinds of distributed systems, e.g., distributed databases [15, 20], distributed coordination systems [4, 60], and distributed computing frameworks [3, 11], are widely used in various areas, e.g., finance, online shopping and transportation.

Distributed systems can provide higher reliability and availability than traditional standalone software. Distributed systems usually adopt complex designs, and face diverse nondeterminism caused by network messages, user requests and external faults. Thus, intricate bugs can exist in distributed system designs and implementations, and challenge their reliability and availability. These bugs can cause severe impacts, e.g., service outage, and result in millions of dollars of damage [17]. However, it is hard to explore all possible testing scenarios to find these bugs.

Recently, researchers have made significant progress in applying formal methods to unearth bugs in distributed system designs [16, 59, 71]. Through building the formal specification for a distributed system design, developers can *clarify the expected behaviors* of the target system. Further, by automatically exploring the state space of the specification, formal methods can verify the correctness of the system design and help developers gain confidence. For example, Zave [71] utilizes Alloy [1] to model and verify Chord [65]. Developers in Amazon [59] and Microsoft [16] use TLA+ [45] to model and verify their distributed system designs.

On the other hand, researchers have proposed many approaches to detect bugs in distributed system implementations. We classify them into three categories.

- Formal verification frameworks [41, 42, 67] can verify the properties of distributed system implementations in a refinement-style way. However, the verification process is complex and time-consuming. Verifying a system implementation usually requires multiple person-years' effort [58]. Therefore, it is challenging to apply them on real-world distributed systems.
- Model-based testing [31, 44, 51] utilizes abstract models to generate test cases to test specific properties or behaviors in distributed system implementations. For example, Modulo [44] models the data consistency property in distributed storage systems and generates test cases to find convergence failure bugs.

- Implementation-level model checkers [47, 57, 64, 68, 69] are specifically designed for finding bugs in distributed system implementations. They intercept and reorder nondeterministic distributed events (e.g., messages and faults) at run time. They cannot know the target system's expected execution results (i.e., test oracles), and rely on developers to manually write general assertions for specific system properties or behaviors to reveal bugs.

We can see that there exists a gap between the formal specification and its corresponding implementation in these approaches. First, although the formal specification of a distributed system can be verified as correct, we still do not know whether the corresponding implementation conforms to the verified specification, and is free of bugs [33]. Second, the formal specification has specified the correct behaviors of a distributed system, but it cannot be used to judge whether the implementation meets these correct behaviors.

In this paper, we propose a novel testing technique, *Model checking guided testing* (*Mocket*), to fill the gap between the formal specification and the implementation in a distributed system. Given the TLA+ [45] specification for a target distributed system, we analyze the verified states generated through verifying the specification, and generate test cases for the system implementation. After mapping the TLA+ specification to the corresponding code in the system implementation, we further deterministically force the system execution to follow the test cases generated from verifying the specification. During system testing, we monitor the system's runtime states and compare them with the corresponding verified states in the TLA+ specification. Any divergence implies an inconsistency between the specification and the implementation, and raises a potential bug.

We apply Mocket on three open-source distributed systems, i.e., Xraft [18], Raft-java [13] and ZooKeeper [4]. Finally, we find 7 bugs, among which 3 bugs are confirmed as previously unknown bugs, and 4 are known bugs. Besides, we find that 2 inconsistencies between the specification and the implementation are caused by specification bugs in the official Raft specification [9]. We have made Mocket publicly available at https://github.com/tcse-iscas/Mocket.

In summary, we make the following main contributions.

- We propose a novel approach, Mocket, which utilizes formal model checking to systematically test distributed system implementations, and checks whether a distributed system implementation violates its specification.
- We implement Mocket, and apply it on three popular distributed systems. Mocket successfully unearths 3 previously unknown bugs in them.

2 Preliminaries

In recent years, distributed system developers are increasingly using formal languages to model their systems, and further verify their system designs' correctness by model checkers [16, 59, 71]. Among all kinds of formal languages, TLA+ [45] is popular for modelling distributed systems, and TLC [5] is the most commonly used TLA+ model checker.

```
1. CONSTANTS Max, NotMax, Data, Nil
2. VARIABLES msg, cache, stage
3. vars ≜ <msg, cache, stage>
4. Init ≜ ∧ msg = Nil
5.       ∧ stage = "request"
6.       ∧ cache = {}
7.
8. getMax(S) ≜ CHOOSE t ∈ S : ∀s ∈ S : t ≥ s
9. Request(data) ≜ ∧ stage = "request"
10.                ∧ stage' = "respond"
11.                ∧ msg' = data
12.                ∧ UNCHANGED ≪ cache ≫
13. Respond ≜ ∧ stage = "respond"
14.           ∧ stage' = "request"
15.           ∧ cache' = cache ∪ {msg}
16.           ∧ msg' = IF msg = getMax(cache')
17.                    THEN Max ELSE NotMax
18. Next ≜ ∨ ∃ d ∈ Data : Request(d)
19.        ∨ Respond
20. Spec ≜ Init ∧ □[Next]_vars
21.
22. Invariant ≜ Cardinality(cache) ≤ Cardinality(Data)
```

Figure 1. A TLA+ specification example.

We introduce TLA+ and TLC by a simple example [46] in Figure 1. The example defines a process, in which, the server uses a set *cache* to store data *msg* from the client, and responds the client with two values: *Max* if *msg* is the largest in *cache* so far, or *NotMax* if it is not.

2.1 TLA+ Specification

Developers use three kinds of elements, i.e., variables, actions and constants, to define a system's TLA+ specification, and utilize properties to express constraints that the system must satisfy.

Variables. Variables express the system's states. Variables are decorated by keyword `VARIABLES`. For example, there are three variables *msg*, *cache* and *stage* in Figure 1 (Line 2). Among them, *msg* stores the request data from the client and the responding content from the server. *cache* is a set on the server that stores all request data. *stage* marks which action, i.e., *Request* or *Repond*, can act on the current state.

Actions. Actions express the system's behaviors. They are functions written in the first-order logic, and are used to define the modification logic on states, i.e., variables. Note that not all functions are actions. Only functions invoked after keyword *Next* and connected by disjunction operators are actions. They can interleave with each other to generate all possible states of the system. For example, Figure 1 shows two actions *Request* (Line 18) and *Respond* (Line 19), but function *getMax* is not an action.

Constants. Constants are used to define specific data values. They are decorated by keyword `CONSTANTS`, and their values are assigned before the model checking process and cannot be changed. Figure 1 defines 4 constants (Line 1). *Nil*

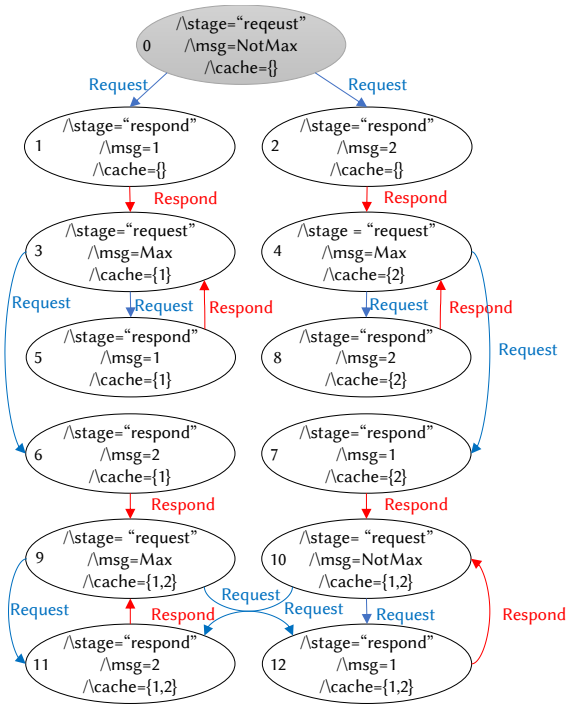

Figure 2. The state space graph generated when verifying the specification in Figure 1 by TLC. Each state is marked by a unique number. State 0 is the initial state.

is the default value of *msg*. *Max* and *NotMax* are two alternative values of *msg*. *Data* restricts what values in *Request* can be written into the server.

Properties. TLA+ developers use properties to define behavior constraints for the target system. In Figure 1, we define an invariant to restrict that the size of *cache* must be smaller than or equal to the size of *Data* (Line 22). The model checker can verify whether any state violates the property. Note that properties have no effect on the construction of the state space. Thus, they are not considered in our work.

2.2 Verifying TLA+ Specifications by TLC

TLC [5] is an explicit model checker for TLA+ specifications. When performing model checking, developers must assign values for all constants, and optionally specify the property to check. Then, the checker starts from the *Init* state in *Spec* (Line 20 in Figure 1) and performs actions on the current state to iteratively enumerate all possible states. This process ends in two situations, i.e., a state violates the property or all states are checked. Finally, TLC can produce the whole state space as a graph. For example, we use TLC to check the specification in Figure 1 with setting *Data* as {1, 2}, and obtain the state space graph shown in Figure 2.

By performing model checking on a TLA+ specification, developers can find counterexamples that violate the defined property, or confirm that the system design is correct.

3 Mocket Overview

We propose Mocket to test distributed system implementations under the guidance of TLA+ specifications. Figure 3 shows the overview of Mocket. Given the target distributed system and the TLA+ specification that models its system design, developers first map the variables and actions in the TLA+ specification to the corresponding code by adding annotations in the system implementation (①). Then, based on the state space graph generated by TLC model checking (②), Mocket generates test cases, in which, each test case consists of an action sequence and the expected states after each action in the sequence (③). Figure 3 shows several test cases in Mocket's testbed, e.g., $s_0 \rightarrow a_1 \rightarrow s_1 \rightarrow a_2 \rightarrow s_3 \rightarrow a_4$ in the current test case box. Here s_0 is the initial state. Actions a_1 and a_2 act on the previous states, and generate new states s_1 and s_3, respectively. Finally, guided by the generated test cases, Mocket performs the controlled testing on the instrumented target system (④). During testing, Mocket forces the target system to follow the action sequence in each test case, e.g., $a_1 \rightarrow a_2 \rightarrow a_4$ in Figure 3, and checks whether the system execution and states are consistent with the test case. Any discrepancy on a state s or an action a will be reported as a potential bug.

To make Mocket to work on real-world distributed system implementations, we need to tackle the following three technical challenges.

Challenge 1: How to map a TLA+ specification to its system implementation? The TLA+ specification only models the key state variables and actions for the target system. That means that some code logic in the implementation is ignored in the TLA+ specification. Thus, it is challenging to perform a perfect mapping between TLA+ elements and their corresponding code logic in the implementation. Instead, we perform the *action-level mapping*. More specifically, we map the name and parameters of actions in the TLA+ specification to the corresponding code in the implementation, and do not consider the internal logic of actions. Figure 4 shows an example about our action-level mapping. Figure 4a defines action *BecomeLeader*, and its corresponding implementation is method *becomeLeader* in Figure 4b. We map action *BecomeLeader* by simply annotating the action's name (Line 6) and collecting the action's parameters (Line 8), but ignore the concrete logic within action *BecomeLeader*.

Challenge 2: How to generate executable test cases based on the state space generated by TLC model checking? As shown in Figure 2, the state space generated by TLC model checking is a graph with all possible states of the target system design. To generate executable test cases in real-world systems, Mocket traverses the state space graph, and takes a verified path that starts from the initial state as a test case.

Besides, although the state space graph is an abstract model for the target system, it still has large amounts of

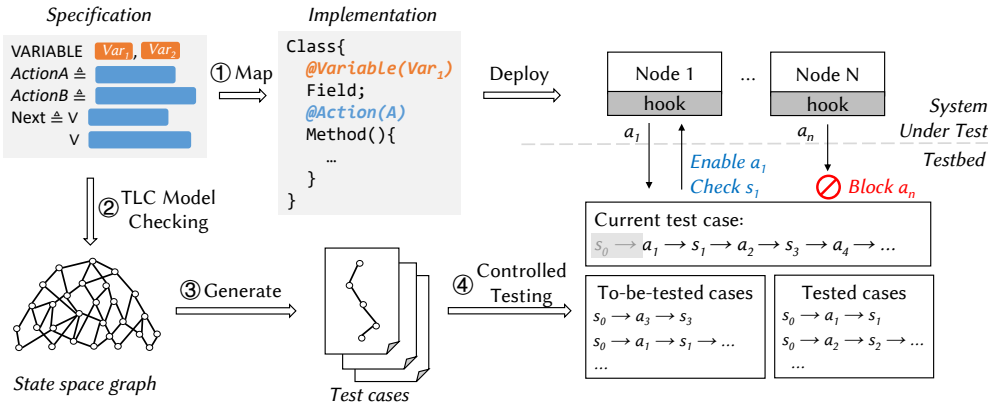

Figure 3. The overview of Mocket. The gray box in current test case denotes that the states and actions are already tested.

```
1. VARIABLE nodeState
2. BecomeLeader(i) ==
3.    /\ nodeState' = [nodeState EXCEPT ![i] = Leader]
```

(a) Variable *nodeState* and action *BecomeLeader* in Raft specification.

```
1.  public class RaftNode {
2. +   @Variable("nodeState")
3.     private NodeState state = NodeState.STATE_FOLLOWER;
4. +   public static NodeState Mocket$state
5. +       = NodeState.STATE_FOLLOWER;
6. +   @Action("BecomeLeader")
7.     private void becomeLeader() {
8. +       Action.collectParams(this.NodeId);
9. +       Mocket.notifyAndBlock();
10.        state = NodeState.STATE_LEADER;
11. +      Mocket$state = NodeState.STATE_LEADER;
12.        …
13. +      Mocket.checkAllStates();
14.    }
15. }
```

(b) The corresponding implementation in Raft-java.

Figure 4. Mocket maps variable *nodeState* and single-node action *BecomeLeader* in the Raft specification to the corresponding implementation in Raft-java. The code in the red color is manually added, and the code in the blue color is automatically generated by Mocket.

states due to the complexity nature of distributed systems. Thus, Mocket further adopts two orthogonal strategies, i.e., edge coverage guided graph traversal and partial order reduction to reduce the number of generated test cases.

Challenge 3: How to perform the controlled testing for the target distributed system? A test case contains a sequence of actions in the specification. In the controlled testing, we force the target system to execute these actions in the same order as that in a test case. However, real-world distributed systems are usually executed with various nondeterminism, e.g., message interleaving and external faults.

To tackle this challenge, Mocket performs the *action-level instrumentation* according to developers' annotations, and then controls the system execution. To be specific, Mocket first adds a hook before the execution of each action (e.g., Line 9 in Figure 4b) through code instrumentation. During system testing, when the distributed system under test (SUT) encounters an annotated action, we will block the action, and send a notification to Mocket's testbed. We further determine whether the action can be scheduled based on the action order in the test case. After the action finishes, the instrumented code collects and reports runtime values of all state variables (e.g., Line 13 in Figure 4b) in SUT. Mocket checks whether the current state of SUT conforms to the corresponding state in the test case. If we find any divergence during this controlled testing, we will report a potential bug.

We explain the above process using Figure 3. In Figure 3, both Node 1 and Node N are blocked, and send two notifications about actions a_1 and a_n to the testbed. According to the action order in the current test case, the testbed enables a_1. Further, after the execution of a_1, Mocket checks whether the SUT's state is the same as its corresponding state s_1 in the test case.

4 Mocket Design

In this section, we introduce Mocket's three stages in detail, i.e., mapping a specification to its corresponding implementation (Section 4.1), test case generation (Section 4.2) and controlled testing (Section 4.3).

4.1 Map a Specification to its Corresponding Implementation

A TLA+ specification consists of variables, actions and constants. In this section, we present how to map each TLA+ element to its corresponding implementation.

4.1.1 Map TLA+ Variables. According to the purposes of variables in the TLA+ specification, we divide TLA+ variables into four categories, i.e., state-related variables, message-related variables, action counters and auxiliary variables. State-related variables express the system states, e.g., *state* (Line 1) in Figure 4a represents the role of a Raft node. Message-related variables are message sets that store all on-the-fly messages transmitted among nodes in the TLA+

specification. Action counters are used to restrict the state space size for TLC model checking, e.g., *clientRequests* in Raft specification [9] limits the number of user requests in model checking. Auxiliary variables are used to ease the expression and verification of the TLA+ specification, e.g., *mode* in Figure 1 is used to control the execution order of actions. Variables in different categories have different mapping methods to their corresponding implementations.

State-related variables. State-related variables in the TLA+ specification express the target system's states. They are mapped to two different kinds of implementations, i.e., class variables (aka class fields) and method variables. For a TLA+ variable that is mapped to a class variable, we use Mocket's annotation *@Variable* to build its mapping relationship. For example, in Figure 4a, TLA+ variable *nodeState* (Line 1) defines the role of a Raft node. In Raft-java [13] (Figure 4b), it is mapped to class variable *state* (Line 3). We annotate class variable *state* with *@Variable*("*nodeState*") (Line 2), in which "nodeState" is the name of TLA+ variable *nodeState*, to denote that class variable *state* is the corresponding implementation of TLA+ variable *nodeState*. For a TLA+ variable that is mapped to a method variable, we cannot use annotations to build its mapping relationship. Thus, Mocket uses a tuple <*SpecName*, *ImplName*, *Location*> to store its mapping relationship, in which *SpecName* is the TLA+ variable's name, *ImplName* is the name of its corresponding method variable, and *Location* is the line number in the source code file that declares the method variable. These mapping tuples are stored in a configuration file.

Mocket utilizes the above mapping relationships to instrument the system implementation, and obtains the runtime values of TLA+ variables. More details about Mocket's instrumentation are introduced in Section 4.3.1.

Message-related variables. Message-related variables in the TLA+ specification are used to simulate the message communication processes in distributed systems. We take variable *messages* in Raft specification [9] as an example. Variable *messages* is represented as an unordered set, which is used to temporarily store all on-the-fly messages. When an action sends a message *m*, message *m* is first put into *messages*. When another action wants to process message *m*, it will retrieve message *m* from *messages*. Note that a TLA+ specification can have multiple message-related variables to define different kinds of message communication processes in the target system. For example, in ZooKeeper, we use *le_msgs* and *bc_msgs* to model the message communication in the leader election stage and broadcast stage, respectively.

In a real-world system, we cannot find the corresponding implementation for message-related variables. Thus, we create a message set in Mocket's testbed for each message-related variable. When the system under test encounters an action that sends a message, it sends a notification to Mocket's testbed about the action and the message content. Then, Mocket puts the message in the message set. When the system encounters an action that receives a message, it retrieves the message and sends a notification to Mocket's testbed about the action and the message content, and then Mocket's testbed removes the notified message from the message set. More details about message-related actions are introduced in Section 4.1.2.

Action counters. Action counters are used to impose restrictions on the state space exploration in TLC model checking. They must be used with constants. For example, variable *clientRequests* in Raft specification [9] counts the execution times of action *ClientRequest*, and constant *ClientRequestLimit* sets the maximum execution times for action *ClientRequest*. Mocket does not need to restrict the execution times of actions during testing, since the execution times of each action are fixed in a test case. Therefore, we do not build mapping relationships for action counters.

Auxiliary variables. Auxiliary variables are used to assist TLA+ specifications' expression and verification. For example, variable *mode* in Figure 1 is used to control the execution order of action *Input* and *Respond*. Since auxiliary variables are only used for TLC model checking, we do not build mapping relationships for them.

4.1.2 Map TLA+ Actions.

According to how we map TLA+ actions into system implementations, we divide TLA+ actions into four categories, i.e., single-node actions, message-related actions, external faults, and user requests.

Among these categories, single-node actions and message-related actions can spontaneously occur during system running. For example, action *BecomeLeader* in Figure 4 can occur when a node is elected as the leader. These actions can be mapped to corresponding code in the target system. The remaining two categories, i.e., external faults and user requests, cannot spontaneously occur during system running. They should be triggered by external behaviors. For example, action *UserRequest* in Raft specification [9] requires that a user writes data into Raft system. Thus, to map these actions, we need to design new methods to simulate them.

Single-node actions. Single-node actions are executed within a single node. They can be mapped to methods or code snippets in the target system.

For an action that can be mapped to a method, we use Mocket's annotation *@Action* to build its mapping relationship. For example, action *BecomeLeader* (Line 2 − 3 in Figure 4a) in Raft specification [9] changes the node's role as *Leader*, and is mapped to method *becomeLeader* (Line 7 − 14 in Figure 4b). We annotate method *becomeLeader* with *@Action*("*BecomeLeader*") (Line 6), in which *"BecomeLeader"* is the name of action *BecomeLeader*, to mark that the method is the corresponding implementation of action *BecomeLeader*.

For an action that cannot be mapped to a method, but has a corresponding implementation in a code snippet, we use Mocket's *Action.begin* and *Action.end* APIs to surround the

```
1.  public Vote lookForLeader() {
2. +   Action.begin("StartElection");
3. +   Action.getParams(self.getMyid());
4. +   Mocket.notifyAndBlock();
5. +   // Check the initial state
6. +   Mocket.checkAllStates();
7.     …
8.     sendNotifications();
9.     …
10. +  Mocket.checkAllStates();
11. +  Action.end("StartElection");
12.    while ((self.getPeerState() ==
                ServerState.LOOKING) && (!stop)) {
13. +      Action.begin("HandleVote");
14. +      Action.getParams(self.getMyid());
15. +      Mocket.notifyAndBlock();
16.        …
17.        Notification n = recvqueue.poll();
18.        …
19. +      Mocket.checkAllStates();
20. +      Action.end("HandleVote");
21.    }
22. }
```

Figure 5. Mocket maps single-node actions *StartElection* and *HandleVote* to code snippets in ZooKeeper. The code in the red color are manually added, and the code in the blue color are automatically generated by Mocket.

mapped code snippet. For example, Figure 5 shows two code snippets in ZooKeeper (Line 2−11 and Line 13−20), which are the corresponding implementations for action *StartElection* and *HandleVote*, respectively. Action *StartElection*'s corresponding implementation launches a new round of leader election on a node. Action *HandleVote*'s corresponding implementation is located in a thread loop, which can continuously retrieve voting messages from the local message queue and handle them. We map these two actions by surrounding the corresponding code snippets with *Action.begin* (Line 2 and 13) and *Action.end* (Line 11 and 20) APIs, which use the corresponding actions' names as API parameters. Besides, we collect the runtime values of corresponding actions' parameters by using API *Action.getParams* (Line 3 and 14).

Message-related actions. Message-related actions define the processes in which a node sends / receives messages to / from other nodes. Similar to single-node actions, they can be mapped to methods or code snippets in the target system. We first use Mocket's annotation *@Action*, or *Action.begin* and *Action.end* APIs to map message-related actions to their corresponding methods or code snippets. As discussed in Section 4.1.1, we cannot map message-related variables to the corresponding code in the target system. However, we need to check the values of message-related variables during system testing. To achieve this, we manually collect the runtime values of messages in message-related actions by using Mocket's API *Action.getMsg*.

Figure 6a shows a message-sending action *RequestVote(i, j)*, in which node i sends a vote request message to node j. In action *RequestVote*, TLA+ function *Send* puts the message content (Line 3 − 8), e.g., *mtype* and *mterm*, into a message-related variable *messages*. In the corresponding implementation in Figure 6b, we use Mocket's API *Action.getMsg* (Line

```
1. RequestVote(i, j) ==
2.   /\ state[i] = Candidate
3.   /\ Send([mtype         |-> RequestVoteRequest,
4.            mterm         |-> currentTerm[i],
5.            mlastLogTerm  |-> LastTerm(log[i]),
6.            mlastLogIndex |-> Len(log[i]),
7.            msource       |-> i,
8.            mdest         |-> j])
```

(a) Action *RequestVote* in Raft specification.

```
1. +@Action("RequestVote")
2.  private void requestVote(Peer peer) {
3. +   Action.getParams(this.NodeId, peer.NodeId);
4. +   Mocket.notifyAndBlock();
5.     …
6.     requestBuilder.setServerId(localServer.getServerId())
7.                 .setTerm(currentTerm)
8.                 .setLastLogIndex(getLastLogIndex())
9.                 .setLastLogTerm(getLastLogTerm());
10. +  Action.getMsg("RequestVoteRequest",
11. +              currentTerm,
12. +              getLastLogTerm(),
13. +              raftLog.getLastLogIndex(),
14. +              this.NodeId,
15. +              peer.NodeId);
16.    …
17.    peer.getRaftConsensusServiceAsync()
18.                .requestVote(requestBuilder.build());
19.    …
20. +  Mocket.checkAllStates();
21. }
```

(b) Instrumenting method *requestVote* in raft-java's class *RaftNode*.

Figure 6. Mocket performs instrumentation for mapping message-related action *RequestVote* in Raft. The code in the red color are manually added, and the code in the blue color are automatically generated by Mocket.

10) to get the runtime values of the message content. Note that *Action.getMsg* should be added after the program point, at which we can access all values of the message content (Line 10). To build the mapping relationship of the message content between a message-related TLA+ variable and the corresponding message set in Mocket, the values of the message content in *Action.getMsg* (Line 10 − 15 in Figure 6b) must be placed in the same order as that in the TLA+ specification (Line 3 − 8 in Figure 6a).

Similarly, we also need to use API *Action.getMsg* to get the runtime values of the received message in message-receiving actions. But, API *Action.getMsg* is usually placed at the beginning of a message-receiving action's corresponding implementation. For example, action *HandleRequestVoteRequest* in Raft receives a vote request message. We map action *HandleRequestVoteRequest* to method *requestVote*(*request*) in Raft-java, in which parameter *request* contains all values of a received message's content, so we can add *Action.getMsg* at the first line in method *requestVote*(*request*).

External faults. External faults cannot spontaneously occur during system running. In Mocket, we simulate them by invoking specific scripts or overriding related actions. Mocket supports four kinds of external faults, i.e., node crash, node restart, message drop and message duplicate.

Node crash and node restart faults are simulated by invoking specific scripts. For a node crash fault, we use a script

that kills the corresponding node's process to simulate it. For a node restart fault, we use a script that kills the corresponding node's process and relaunches a new node with the same configuration to simulate it. For example, we use action $Crash(i)$ to express that node i crashes. The parameter i is mapped to the corresponding node's process ID that is collected when the target system is deployed and initiated. During system testing, when Mocket observes that the next scheduled action is $Crash(i)$, it will invoke the corresponding script to kill the node immediately.

Message drop and message duplicate faults are simulated by overriding message-receiving actions. For a message drop fault, we skip the statements that handle the received message in the corresponding message-receiving action. For a message duplicate fault, we execute these statements in the corresponding message-receiving action twice. In this way, message drop and message duplicate faults can reuse the collected information in message-related actions, including parameter values and the message content in message-related variables. To make an overridden action to be executed as a normal message-related action when we do not need to inject a fault, we add a switch for each message-related action. When Mocket schedules a normal message-related action, the switch is off. When Mocket schedules a message drop / duplicate action and encounters a message-receiving action whose message should be dropped / duplicated, it turns the switch on and activates the corresponding overriding logic.

User requests. User requests are system-specific actions. Different distributed systems usually provide multiple scripts for users to access different services, e.g., reading and writing data in Raft. Similar to external faults, we need to launch user requests during system testing by invoking specific scripts. For example, action *ClientRequest* in Raft specification [9] expresses a user request that writes data into Raft system, and Raft-java provides a script file *run_client.sh* to write a pair of key-value into Raft-java.

```
./run_client.sh $Cluster $Key $Value
```

Note that TLA+ developers usually do not model the concrete data values in user requests, e.g., $\$Key$ and $\$Value$. Instead, they simply use values of action counters, e.g., variable *clientRequests*, to represent different actions executed in model checking. Thus, in a test case, we can distinguish different user requests by the value of *clientRequests*, e.g., 1 is the first execution of *ClientRequest* and 2 is the second one. However, to launch user requests during system testing, we need concrete values to invoke the above script. To tackle this problem, we simply use different data for different user requests. For example, we write (1, 1) into the target system when scheduling user request *ClientRequest* at the first time in the test case, and write (2, 2) when the second *ClientRequest* is scheduled.

4.1.3 Map TLA+ Constants.

TLA+ constants are used to express the specific values in a TLA+ specification. They can be mapped to specific code in the target system. For example, constant *Follower*, *Candidate* and *Leader* in Raft specification [9] express possible roles that a node can have, and can be assigned to variable *nodeState* (Line 1 in Figure 4a). In Raft-java [13], an enumerated type *NodeState* implements these three constants as three values, i.e., *STATE_FOLLOWER*, *STATE_CANDIDATE* and *STATE_LEADER*.

We use a map to store the mapping relationships between constants and their corresponding implementation. During system testing, Mocket can utilize the map to query related values when necessary. For example, when checking a state in a test case, Mocket finds that the value of variable *nodeState* is *Follower*, and the corresponding variable value collected is *STATE_FOLLOWER* during testing. After querying the map, Mocket can know that the state in the test case and the collected value during system testing are consistent. Note that we do not map constants used with action counters introduced in Section 4.1.1, since the action execution times are fixed in a test case.

4.2 Test Case Generation

TLC model checker [5] can generate a state space graph (as shown in Figure 2) in a GraphViz [21] DOT file after checking a TLA+ specification. In the state space graph, each edge denotes an action, and each node denotes a verified state. We traverse the state space graph to generate executable test cases in a real-world distributed system. A test case is a path in the state space graph, which starts from the initial state (e.g., state 0 in Figure 2), and ends in a certain state (e.g., state 9 in Figure 2). In a test case, an edge represents an action that is scheduled during system testing, and each node represents a program point to check system states. Note that we do not treat a path that does not start from the initial state as a test case, since it is challenging to make a real-world distributed system execute from an arbitrary intermediate state.

The state space graph usually contains large amounts of states and edges, e.g., TLC can generate over 10^5 states and 10^6 edges when checking ZooKeeper's TLA+ specification. If we traverse such a state space graph without any reduction strategies, we can generate numerous test cases. For example, by iteratively traversing the cycle existing in the state space graph in Figure 2, we can theoretically generate infinite test cases for such a simple state space graph. To test a real-world distributed system within in the limited time, we apply two strategies, i.e., edge coverage guided graph traversal and partial order reduction, to generate representative test cases in real-world distributed systems.

4.2.1 Edge Coverage Guided Graph Traversal.

There are many graph traversal strategies for generating executable test cases based on the state space graph, e.g., node coverage guided traversal and edge coverage guided traversal. The former aims to cover more states, and the latter aims to cover more actions. To test as many actions as possible in a

Algorithm 1: Edge coverage guided graph traversal

```
   Input: The graph g and the root node initState
   Output: The set of paths paths
1  paths ← ∅
2  initPath ← new Path()
3  traverse(initState, initPath, g)
4  Function traverse(state, path, graph) do
5      if isEndState(state) ∨ allOutEdgeVisited(state)
        then
6          paths.add(path)
7          return
8      foreach succ ∈ state.successors do
9          edge ← graph.edge(state, succ)
10         if edge.visited = TRUE then
11             continue
12         else
13             edge.visited ← TRUE
14             path.add(succ)
15             traverse(succ, path, graph)
```

real-world distributed system, we utilize a depth first search algorithm with the edge coverage guided traversal strategy to traverse the state space graph.

As Algorithm 1 shows, we start the traversal from the initial state (Line 3), and iteratively traverse its successors (Line 4 – 15). The traversal for a path ends in two conditions, i.e., all edges from the state to its successors are visited before, or the current state is an end state (Line 5). End states are specified by developers. For example, if developers want to test leader election in Raft, they can set the state that is generated by a *BecomeLeader* action as an end state, i.e., a leader has been elected. If either condition is satisfied, we add the *path* in *paths* (Line 6). Otherwise, we continue the traversal. For each edge from the current state to its successor, if it is visited before, we directly skip it (Line 10 – 11). Otherwise, we set it as *visited* (Line 13), add the successor to *path* (Line 14), and continue the traversal on the successor (Line 15).

After the graph traversal, we get a series of paths. Based on these paths, we further generate a group of test cases, which are sequences of actions and states. In a test case, an action contains the information of the action's name and parameter values, and a state contains the values of all the variables defined in the TLA+ specification.

4.2.2 Partial Order Reduction. Although our edge coverage guided graph traversal can generate representative test cases, it can still generate large amounts of test cases. Thus, we further use partial order reduction (POR) [32, 38] to remove some test cases that are less interesting to be tested.

If two actions a_1 and a_2 acting on the same state s_0 can result in the same state s_3 regardless of their schedule order, i.e., $s_0 \rightarrow a_1 \rightarrow s_1 \rightarrow a_2 \rightarrow s_3$ and $s_0 \rightarrow a_2 \rightarrow s_2 \rightarrow a_1 \rightarrow s_3$, a_1 and a_2 are commutative, and there is no need to schedule both $a_1 \rightarrow a_2$ and $a_2 \rightarrow a_1$. In this case, we randomly choose one action schedule (e.g., $a_1 \rightarrow a_2$), but omit the other one (e.g., $a_2 \rightarrow a_1$). We can analyze the state space graph, and identify commutative actions. Then, we do not treat the action schedules (i.e., related edges), which are not chosen, as our coverage target during our graph traversal.

4.3 Controlled Testing

Mocket performs system testing based on the mapped TLA+ elements and the generated test cases. To control the order of actions and check states during system testing, Mocket first performs automatic instrumentation for mapped TLA+ variables and actions. Then, Mocket performs a round of testing for each single test case. During system testing, Mocket reports inconsistencies between the TLA+ specification and its corresponding implementation. We further investigate each reported inconsistency and identify potential bugs.

4.3.1 Automatic Instrumentation. For each TLA+ variable that is mapped to a class variable / method variable in the target system, Mocket automatically adds a shadow field / variable in the implementation. Further, whenever a variable in the implementation is initialized or reassigned, Mocket assigns the same value to its corresponding shadow field / variable. For example, in Figure 4b, Mocket adds shadow field *Mocket$state* (Line 4) for the annotated class field *state* (Line 3) in the implementation, and when *state* is initialized as *STATE_FOLLOWER* (Line 3) and reassigned to *STATE_LEADER* (Line 10), *Mocket$state* is set as the same value (Line 5 and 11)[1]. In this way, when checking states during system testing, Mocket can access the runtime value of every mapped variable in the implementation without affecting the target system's execution.

For each TLA+ action that is mapped to a method / code snippet, Mocket automatically adds a hook at the beginning of the method / code snippet to notify Mocket's testbed about the action information, i.e., action name and parameter values, and block the corresponding thread to wait for scheduling. If the action is a message-receiving action, the hook also sends the message content collected by *Action.getMsg* in the action to Mocket's testbed. Based on the message content, Mocket can judge whether to turn the fault injection switch on. At the end of the method / code snippet, Mocket adds a statement to collect variable values, including the message content collected by *Action.getMsg*, and sends them to Mocket's testbed to check states. For example, in Figure 4b, Mocket adds *Mocket.notifyAndBlock* (Line 9) and

[1]We use ASM to duplicate the value of the target variable in the JVM stack, and assign the duplicated value to the corresponding shadow variable.

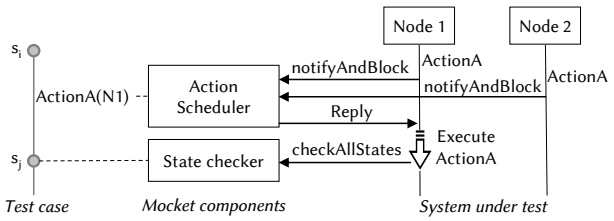

Figure 7. Mocket's testing process.

Mocket.checkAllStates (Line 13) for the annotated method *becomeLeader*.

Note that for the first action scheduled during system testing, we need to check the initial state before executing its corresponding code. For example, in Figure 5, action *StartElection* is the first scheduled action in ZooKeeper's leader election. When *StartElection*'s corresponding code is executed at the first time, Mocket adds *checkAllStates* (Line 6) after *notifyAndBlock* (Line 4) to check the initial state.

4.3.2 System Testing. Figure 7 illustrates Mocket's system testing process for a single test case. Mocket deploys a new cluster for each test case. When a cluster node encounters an action, the instrumented *notifyAndBlock* statement blocks the corresponding thread, and notifies Mocket's action scheduler about the action's name and parameter values. In Figure 7, both *Node* 1 and *Node* 2 notify the action scheduler that they encounter action *ActionA*. When the action scheduler receives a notification, it compares the action information in the notification with the scheduled action in the test case, and replies the node whose action matches the scheduled action. Meanwhile, Mocket moves forward and waits to check the next state following the action. In Figure 7, the scheduled action is *ActionA*(*N*1), which means that action *ActionA* on *Node* 1 should be executed now. Thus, the action scheduler replies *Node* 1 and resumes its blocked thread. For other notified actions that have not been scheduled, e.g., action *ActionA* from *Node* 2, the action scheduler puts them in a set until they match their corresponding scheduled actions in the test case.

When a node, e.g., *Node* 1, finishes executing an action, the instrumented *checkAllStates* statement sends the runtime values of all variables to Mocket's state checker. The state checker compares the received runtime values with the state in the test case. If they are consistent, Mocket moves forward and schedules the next action. Otherwise, we find an inconsistency.

4.3.3 Bug Detection. Mocket reports an inconsistency between a TLA+ specification and its corresponding implementation when any situation below occurs.

- **Inconsistent state.** The state checker finds that the collected runtime values are different from the corresponding state in the test case.
- **Missing action.** The action scheduler waits until timeout, and does not receive any action notification that matches the scheduled action in the test case.
- **Unexpected action.** When a test case finishes, there still exist some action notifications in the action scheduler's waiting set.

When Mocket finds an inconsistency, it generates a bug report, which contains the test case and the inconsistency caused by related states or actions.

Note that Mocket cannot distinguish whether a reported inconsistency is caused by an incorrect implementation or an incorrect specification. For each bug report, we further investigate it and figure out what causes the inconsistency. If the inconsistency is caused by the incorrect system implementation, we treat it as an *implementation bug*. Otherwise, if the distributed system implementation is correct, but its corresponding TLA+ specification is incorrectly written, we treat the inconsistency as a *specification bug*. Specification bugs can make TLC generate unexpected states and actions which do not happen during system testing. For example, we use Raft's official TLA+ specification [9] to test Raft-java [13], and find that Raft's official specification contains bugs that have been correctly fixed in Raft-java's implementation.

5 Applying Mocket on Real-World Systems

In this section, we first present the implementation of Mocket. Then we introduce two practices of applying Mocket, i.e., testing two Raft [61] implementations and ZooKeeper [4]. Last, we discuss the lessons learned in our practices.

5.1 Mocket Implementation

Mocket is implemented in Java and Python, and consists of around 5K lines of code (LOC), including annotations, runtime instrumentation, test case generation, action scheduler and state checker. The annotations for mapping variables and actions are implemented based on *java.lang.annotation* package. The runtime instrumentation is performed based on ASM [2] library. The test case generation is implemented based on Python's networkx [8] package.

5.2 Applying Mocket on Raft

Raft [60] is a classical distributed consensus protocol. Researchers have already modelled [9] and verified [12] its design. Many popular distributed systems, e.g., CockroachDB [20] and TiDB [15], adopt Raft as their underlying consensus protocol to maintain data consistency.

We select two open-source Java-based implementations, i.e., Xraft [18] and Raft-java [13] as our target systems. Raft-java is a popular Raft implementation with more than 1K GitHub stars. Xraft is less popular (around 200 GitHub stars), but it is used as the code example in a published book [29] that introduces Raft. These two Raft implementations are claimed to be compliant to the official Raft specification.

Table 1. Development Effort on Real-World Systems

System	Impl. (LOC)	Spec.			Mapping (LOC)
		LOC	# Var.	# Act.	
Xraft	16,530	841	15	17	151
Raft-java	15,017	809	15	15	152
ZooKeeper	15,895	1,053	25	20	134

However, they are independently developed, and have different implementation choices. For example, Xraft uses asynchronous communication, but Raft-java uses synchronous communication. To make the official Raft specification totally compliant to a Raft implementation, we have to make some minor modifications on the official Raft specification [9] to support their implementation choices.

One author took about one week to modify the official TLA+ specification according to their corresponding implementations, and map the modified specification to its corresponding Raft implementation. We further took about two weeks to perform multi-round testing on these two systems. Table 1 shows the effort we devoted to apply Mocket on Xraft and Raft-java[2]. The modified specifications for Xraft and Raft-java contain 841 and 809 LOC, respectively. Both systems' specifications contain 15 variables. Xraft's specification contains 17 actions, among which, 12 actions come from the official specification [9], and 5 new actions were added by us. Raft-java's specification contains 15 actions. Since Raft-java uses synchronous communication, we remove two external faults *DropMessage* and *DuplicateMessage*, compared with Xraft's specification.

Xraft and Raft-java contain 16,530 and 15,017 LOC, respectively. We use 151 and 152 LOC to map actions and variables in the specifications to Xraft and Raft-java, respectively. Most actions are mapped in less than 5 LOC. Mapping message-related actions requires more effort (10 LOC for each action) than other types of actions, because we need additional code to collect the runtime values of message content.

5.3 Applying Mocket on ZooKeeper

ZooKeeper [4] is a popular and mature distributed coordination system. It has been developed for years, but not ever checked by formal methods. To apply Mocket on ZooKeeper, we developed a TLA+ specification for its core protocol ZAB [43] after reading its code and related design documents.

One author took about three weeks to develop ZAB's TLA+ specification, and map the specification to its corresponding ZooKeeper implementation. We further took about two weeks to perform multi-round testing on ZooKeeper. Table 1 also shows the effort we devoted to apply Mocket on ZooKeeper. The ZooKeeper ZAB's specification contains 1,053 LOC, including 25 variables and 20 actions. Among 25 variables, 20 variables are state-related variables, 2 variables are message-related variables, and the remaining 3 variables are action counters. We design 2 different message-related variables because ZAB has two different message communication mechanisms in its leader election stage and synchronization stage. Among the 20 actions, 6 are single-node actions, 12 are message-related actions, and 2 are external faults. Note that we do not model message duplicate and message drop faults, because ZAB's designers never mentioned that ZAB could handle them.

In ZooKeeper, the ZAB-related implementation is mainly located in package *org.apache.zookeeper.server*, and contains 15,895 LOC. We use 134 LOC to annotate TLA+ variables and actions in the ZAB-related implementation. Among them, we use 20 LOC to map TLA+ variables, including 17 LOC for mapping 17 class-type TLA+ variables, and 3 lines of configuration file code for mapping 3 method-type TLA+ variables. We use 114 LOC to map TLA+ actions, and each action is mapped with no more than 10 LOC.

5.4 Lessons Learned

Testing oriented TLA+ specification development process. It is generally recommended that the TLA+ specification should be developed prior to the corresponding implementation [39]. However, to test an existing and mature distributed system, e.g., ZooKeeper, we have to develop the TLA+ specification referring to its implementation. The specification development process for applying Mocket is different from the traditional development workflow. The TLA+ specification only for model checking is property oriented. Developers usually first consider what properties to check, and then define property-related variables and actions. In our testing oriented development process, we first extract actions from the implementation, and then consider variables related to these actions.

Potential errors introduced by developers when applying Mocket. Applying Mocket to test a real-world distributed system requires some manual effort, e.g., developing TLA+ specification for the target system, and mapping TLA+ elements to their corresponding implementations. Developers (e.g., the authors) may introduce errors in this manual process, which can cause Mocket to report false inconsistencies, i.e., false positives. Based on our experience, we summarize two types of errors that developers may introduce and explain how we handle them.

First, when developing the TLA+ specification for an existing implementation, developers can possibly write incorrect action logic that violates the target system's design. For example, if a real-world concurrent system includes certain synchronization in its implementation, but TLA+ developers do not model the synchronization in the TLA+ specification, Mocket will report a missing action during system testing.

Second, when mapping the TLA+ elements to their corresponding implementations, developers can possibly build

[2]Since we are not the developers of these target systems, we may spend more time than these systems' developers.

Table 2. Bugs Found by Mocket

ID	Type	Reported Inconsistency	Elapsed Time	# Actions
Xraft Bug #1 (New) [23]	Impl. Bug	Inconsistent state for variable *votesGranted*	1 min	6
Xraft Bug #2 (New) [22]	Impl. Bug	Inconsistent state for variable *votedFor*	7 min	9
Xraft Bug #3 (New) [24]	Impl. Bug	Unexpected action *HandleRequestVoteResponse*	39 min	19
Raft-java Bug #1 [14]	Impl. Bug	Missing action *HandleRequestVoteResponse*	6 min	18
Raft-java Bug #2 [19]	Impl. Bug	Inconsistent state for variable *log*	5 hours	31
ZooKeeper Bug #2 [6]	Impl. Bug	Unexpected action *ReceiveMessage*	13 hours	39
ZooKeeper Bug #2 [7]	Impl. Bug	Missing action *StartElection*	29 hours	51
Raft-spec issue #1 (New)	Spec. Bug	Inconsistent state for variable *messages*	< 1 min	8
Raft-spec issue #2 (New)	Spec. Bug	Missing action *UpdateTerm*	< 1 min	5

wrong mapping relationships. For example, if developers miswrite the action's name when annotating a method, Mocket will report a missing action, since Mocket cannot receive the notification of the required action with the correct name.

These errors introduced by developers (i.e., Mocket's users) can only be discovered during system testing. Therefore, we perform multi-round testing to eliminate these errors. If we find that an inconsistency is caused by these developer-introduced errors, we fix these errors, regenerate test cases and perform system testing again. Fortunately, these errors are usually easy to fix. However, the multi-round testing could waste testing resources. In the future, we can design better strategies to avoid these errors.

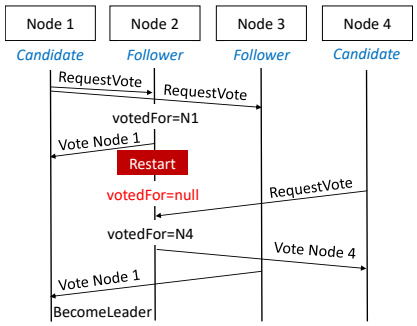

Figure 8. Xraft bug #2 [22]. Node 1 becomes *Leader* when Node 2 changes its vote due to a node restart fault. The reported inconsistency is shown in the red font.

6 Evaluation

We present Mocket's experimental results on three distributed systems, i.e., Xraft [18], Raft-java [13] and ZooKeeper [4]. Table 2 shows the detected bugs. For each bug, we record the elapsed time to reveal it and the number of involved actions in its bug-revealing test case. In this section, we first discuss the detected bugs in detail, and then evaluate the testing effort. Last, we compare Mocket with existing approaches.

6.1 Detected Bugs

As shown in Table 2, Mocket finds 9 bugs in total, including 3 previously unknown implementation bugs, 4 known implementation bugs, and 2 new specification bugs in the official Raft specification [9]. All 3 new implementation bugs have been confirmed by developers. Due to the space limitation, we only introduce new implementation bugs and specification bugs here.

New implementation bugs. In the first Xraft bug [23], Xraft developers oversimplify the implementation of variable *votesGranted* in the Raft specification. In Raft, variable *votesGranted* is used to record the nodes that have granted the vote request from a candidate node. Xraft implements *votesGranted* as an integer value. When the candidate node receives a vote from another node, it increases *votesGranted* by 1. In this bug, Node 1 becomes *Candidate* and votes for itself, and *votesGranted* becomes 1. Then, Node 1 sends a *RequestVote* message to Node 2, and receives a message of granting the vote request from Node 2. Thus, *votesGranted* becomes 2. However, a duplicate message fault makes Node 1 receive two repeated messages from Node 2, and causes that *votesGranted* becomes 3. Finally, Node 1 becomes *Leader* although it does not receive the granted votes from the majority nodes. Mocket finds this bug by observing the inconsistent values of variable *votesGranted*.

Figure 8 shows the second Xraft bug [22]. In this bug, both Node 1 and Node 4 are *Candidate*. Node 1 sends *RequestVote* message to Node 2 and 3. Node 2 grants the vote request, and sets the value of variable *votedFor* as $N1$. Then, a node restart fault occurs on Node 2, which makes *votedFor* become null. Thus, when the *RequestVote* message from Node 4 arrives, Node 2 thinks that it has not voted for any node, and grants the vote request from Node 4. After receiving the vote from Node 3, Node 1 thinks that it has received votes from the majority nodes, i.e., Node 1, 2 and 3, and becomes *Leader*. But in fact, Node 2 has changed its vote and Node 1 is unaware of that. Mocket finds this bug by observing the inconsistent value of variable *votedFor*.

Figure 9 shows a deep bug [24] in Xraft. In this bug, Node 1, 2 and 3 become *Candidate* at first. Node 1 becomes the

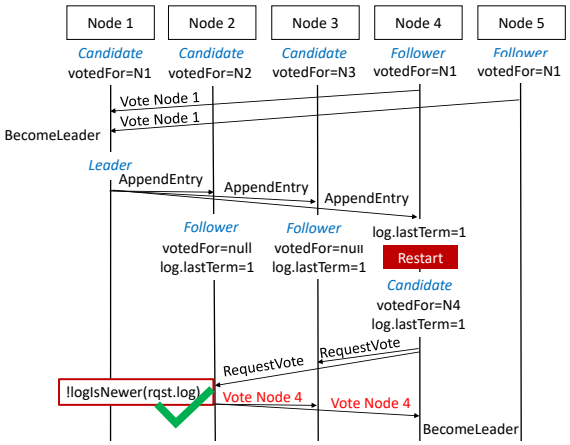

Figure 9. Xraft bug #3 [24]. Both Node 1 and 4 become *Leader*. The reported inconsistency is shown in the red font.

leader after receiving votes from Node 4 and 5, and informs other nodes by *AppendEntry* messages. The nodes receiving message *AppendEntry* will write a log without any data, i.e., NoOp log, which represents that they have voted for others. After writing NoOp logs, Node 2 and 3 become followers and update their states, i.e., changing variable *votedFor* from itself to null and setting *log.lastTerm* as 1. Then, a node restart fault occurs on Node 4. Node 4 becomes *Candidate*, and updates its state, i.e., setting variable *votedFor* as itself, but variable *log.lastTerm* is still 1 since the written NoOp log is persisted. After that, Node 4 sends *RequestVote* messages to Node 2 and 3, and the message contains the information of persisted NoOp log. In Xraft, to make a *Follower* node grant a vote request from a *Candidate* node, the data log on the *Candidate* node must not be staler than that on the *Follower* node. Node 2 and 3 should not grant the vote for Node 4, because the log in *RequestVote* is not the data log. However, the wrong implementation makes Node 2 and 3 wrongly grant the vote requests from Node 4. Finally, Node 4 becomes *Leader* while Node 1 has already been *Leader*. Mocket finds this bug by observing unexpected action *HandleRequestVoteResponse*.

Specification bugs. We reveal two inconsistencies that are caused by the bugs in the official Raft specification. Note that specification bugs have no impact on the specification verification process, but can cause the specification to generate some actions or states that the corresponding implementation cannot turn into during system testing.

Figure 10 shows the first specification bug. Function *UpdateTerm*, *HandleRequestVoteRequest* and *HandleRequestVoteResponse*, are connected by a disjunction operator. This denotes *UpdateTerm* can be performed as an independent action as *HandleRequestVoteRequest* and *HandleRequestVoteResponse*. However, *UpdateTerm* can only be performed within the other two actions in practical system implementations. We need to move *UpdateTerm* into them to fix this bug.

```
1. Receive(m) ==
2.   LET i == m.mdest
3.       j == m.msource
4.   IN  \/ UpdateTerm(i, j, m)
5.       \/ /\ m.mtype = RequestVoteRequest
6.          /\ HandleRequestVoteRequest(i, j, m)
7.       \/ /\ m.mtype = RequestVoteResponse
8.          /\ HandleRequestVoteResponse(i, j, m)
```

Figure 10. Specification bug #1 in Raft. *UpdateTerm* wrongly interleaves with *HandleRequestVoteRequest* and *HandleRequestVoteResponse*.

```
1. HandleAppendEntriesRequest(i, j, m) ==
2.   \/ Reply(…) \* reject request
3.   \/ \* return to follower state
4.      /\ state' = [state EXCEPT ![i] = Follower]
5.   \/ \* accept request when it is a follower
6.      /\ state[i] = Follower
7.      /\ Reply(…)
```

Figure 11. Specification bug #2 in Raft. The second branch does not invoke *Reply*.

Figure 11 shows the second specification bug. Action *HandleAppendEntriesRequest* uses three branches to handle requests under different conditions, i.e., rejecting the request (Line 2), changing the candidate's role into Follower (Line 3 – 4), and accepting the request (Line 5 – 7). However, the second branch does not invoke function *Reply*. When *HandleAppendEntriesRequest* executes its second branch, it does not directly reply the message, but requires the second *HandleAppendEntriesRequest* to execute the third branch to invoke *Reply*. We add a *Reply* function in the second branch to fix this bug.

6.2 Testing Effort

The experiments were performed on a machine with a 3.1 GHz Intel Core i9 CPU, 64 GB memory. All target distributed systems were deployed in the form of pseudo-distributed cluster, i.e., each node runs as a process on the same host machine. In a single test case, a cluster has 5 nodes at most. We inject external faults no more than 5 times, launch user requests no more than 3 times.

Table 3 shows the testing effort for each target distributed system. Column State shows the numbers of states in the state space graph generated by TLC. Column Path_{EC} and $\text{Path}_{\text{EC+POR}}$ show the numbers of paths generated by traversing the state graph when applying only edge coverage (EC) strategy and both edge coverage and partial order reduction (POR) strategies, respectively. We can see that EC can still generate a large amount of paths, while combining EC and POR can generate much fewer paths, e.g., 87% of paths can be reduced in ZooKeeper. This shows that our state space reduction strategies are effective.

Column Time in Table 3 shows the time of executing test cases generated applying both EC and POR. On average, a ZooKeeper test case takes about 10 seconds. Xraft and Raft-java take 7 seconds and 5 seconds to execute a test

Table 3. Testing Effort

System	State	Path_{EC}	$\text{Path}_{\text{EC+POR}}$	Time
Xraft	91,532	296,154	39,047	75 h
Raft-java	23,911	85,976	9,829	13 h
ZooKeeper	105,054	342,770	44,361	123 h

case, respectively. It is reasonable since ZooKeeper is more complex than the other two distributed systems.

6.3 Comparison with Existing Approaches

Mocket closely relates to some existing bug detection approaches for distributed systems, e.g., formal verification frameworks [41, 42, 67], model-based testing [31, 44, 51], and implementation-level model checkers [47, 57, 64, 68, 69]. However, we have encountered difficulties in directly comparing Mocket with these approaches, e.g., lacking available tools. Therefore, we qualitatively discuss the advantages of Mocket over these existing approaches.

Formal verification frameworks. Formal verification frameworks [41, 42, 67] can build a verified distributed system from an abstract specification in a refinement-style way. First, unlike Mocket, formal verification frameworks cannot be used to verify an existing distributed system's implementation. Second, applying formal verification frameworks requires much larger manual effort than Mocket. For example, IronFleet [42] requires a specification with 1,400 lines of code (LOC) and a proof with 39,253 LOC to verify the correctness of an implementation with 5,114 LOC. In comparison, we use 1,187 LOC to apply Mocket on ZooKeeper ZAB protocol's implementation with 15,895 LOC.

Model-based testing. Model-based testing [31, 44, 51] models the specific distributed system properties or behaviors, and generate test cases based on the model.

First, most model-based testing approaches [44, 51] can only define specific properties or behaviors. They cannot perform a systematic testing for distributed systems. For example, Li et al. [51] only models network delay, and Modulo [44] only models the data consistency property in distributed systems. In comparison, Mocket utilizes TLA+ to model all kinds of behaviors and properties in distributed systems.

Second, although MBTCG [31] also uses TLA+ to model the algorithm in MongoDB and generate test inputs, it is designed for a simple program, e.g., operations on a single array, and cannot support common non-deterministic features in distributed systems, e.g., message communication and external faults. Therefore, the bugs in Table 2 cannot be detected by MBTCG theoretically.

Implementation-level model checkers. Implementation-level model checkers [47, 57, 64, 68, 69] intercept and interleave non-deterministic behaviors of distributed systems at runtime, and enforce the target system into different states.

First, implementation-level model checkers test a distributed system under a specified workload, e.g., writing a key-value pair (1, 1) into a Raft system. To perform a systematic testing, developers have to manually design and execute many different workloads. In comparison, Mocket can test all possible workloads modelled in the TLA+ specification, e.g., possible inputs written into a Raft system.

Second, implementation-level model checkers cannot know all expected execution results of distributed systems, e.g., the return result when a crash occurs on a specific execution point, and rely on developers to manually write general assertions related to specific system properties to reveal bugs, e.g., error information in system logs. Other system states beyond the written assertions cannot be checked. In comparison, Mocket can obtain the expected states for each action based on the state space graph, and uses them as test oracles. Furthermore, implementation-level model checkers need more effort to write assertions than Mocket. For example, SAMC [47] uses ZKVerifier.java [10] (59 LOC) to express two properties in ZooKeeper, while we only need to add 2 lines of TLA+ code to define these two properties.

7 Discussion

7.1 Threats to Validity

In our experiment, we evaluate Mocket on two Raft systems and ZooKeeper. Therefore, our experimental results may not reflect the situation in other distributed systems. However, both Raft and ZooKeeper are popular and representative distributed systems. Moreover, our practices in Section 5 on these two kinds of distributed systems also present how to apply Mocket under two different scenarios, i.e., how to apply Mocket on a distributed system with and without its corresponding TLA+ specification, respectively.

Our practices are dependent on the involved developers (e.g., the authors of this paper). This may introduce implicit bias towards the individual developers' expertise. In future, more user studies with other developers (e.g., a distributed system's developer who is familiar with TLA+) could further validate or challenge our approach.

7.2 Limitations

Mocket requires some manual effort from developers. Mocket requires a TLA+ specification, which may be not available for some distributed systems, and involves some effort to develop it. Mocket also requires some manual effort to annotate the target distributed system. Thus, developers should have a certain depth of understanding about the target distributed system. Fortunately, developers only need to understand the protocol design in the target system's implementation rather than all implementation details. For example, we (who are not ZooKeeper developers) can test ZooKeeper with limited understanding of ZooKeeper's implementation, and it took us three weeks to write the TLA+ specification

and annotate ZooKeeper's implementation. Therefore, the required manual effort should be acceptable in practice.

Mocket requires that the TLA+ specification should be close to the corresponding implementation. The abstract level of a TLA+ specification can affect Mocket's bug detection capability. Developers can build a very high-level specification, which may miss some important details, e.g., external faults. Such a high-level specification is easy to build, but can cause Mocket to miss implementation bugs. Developers can also build a very detailed TLA+ specification that is extremely close to its corresponding implementation. Such a detailed specification can help Mocket to unearth more implementation bugs, but is hard to build. Therefore, it is critical to find the balance about how specific the TLA+ specification should be close to the implementation. We cannot provide a clear answer to this balance. In our practice, modelling all user requests, message communication and external faults in the target distributed system should be a proper choice.

Mocket can miss some implementation bugs in the target system under some scenarios. We describe these scenarios as follows.

- Developers might give a TLA+ specification that does not cover all implementation details of a distributed system. If some states and actions in a distributed system are not modelled by its corresponding specification, Mocket will miss implementation bugs in the unmodelled implementations. For example, we only model ZooKeeper's ZAB protocol in our experiment. Thus, we cannot detect implementation bugs in ZooKeeper's authentication module.
- If some concurrency in the system implementation is not properly modelled in the TLA+ specification, we can miss related implementation bugs. Assume that action a_i must happen before a_j in the TLA+ specification, while a_i and a_j can be concurrently scheduled in the system implementation. In such case, Mocket cannot test the schedule $a_j \rightarrow a_i$, and hence misses related implementation bugs hidden in this schedule.
- Although Mocket's edge coverage guided test case generation can cover all actions in a state space graph, it omits some paths that may trigger implementation bugs in the target system. Besides, in our partial order reduction strategy, commutative actions in the state space graph do not always imply that they are commutative in the target system implementation. Thus, our test case generation and reduction strategy can potentially cause Mocket to miss some implementation bugs.

Mocket can only work on Java-based distributed systems. To apply Mocket on a distributed system that is not implemented in Java, we need to reimplement Mocket's annotation and instrumentation mechanisms. However, the remaining parts of Mocket, e.g., test case generation, action scheduler and state checker, can be reused.

8 Related Works

In this section, we discuss related works that are not discussed earlier.

Random testing for distributed systems. Researchers have proposed some approaches for randomly testing distributed systems. PCTCP [62] provides a guarantee of the possibility hitting a potential bug when randomly testing distributed systems. taPCT [63] combines random testing with partial order reduction. Morpheus [70] utilizes conflict analysis among events to make random testing more efficient. Similar to implementation-level model checkers, these approaches also suffer from the test oracle problem. In comparison, Mocket can perform systematic testing for distributed systems, and solves the test oracle problem.

Bug detection in distributed systems. There has been significant progress in understanding and detecting bugs in distributed systems. Researchers have conducted several empirical bug studies to understand different kinds of bugs in distributed systems, e.g., cloud bugs [40], concurrency bugs [48], crash recovery bugs [35], timeout bugs [30], exception-related bugs [27], network partition-related bugs [25, 26] and partial failures [54]. Researchers further propose various approaches to detect bugs in distributed systems. CrashTuner [56], Deminer [37] and CrashFuzz [36] detect crash recovery bugs. CoFi [28] detects network partition-related bugs. DCatch [52], FCatch [53] and PCatch [49, 50] detect concurrency bugs, time-of-fault bugs and performance bugs, respectively. DUPChecker [72] detects software upgrade failures in distributed systems. DIET [27] detects exception-related bugs in distributed systems. MPChecker [55], FlowDist [34] and DisTA [66] detect missing-permission-check bugs and information flow vulnerabilities in distributed systems, respectively. These works use specific patterns to detect different kinds of bugs in distributed systems. Mocket is general, and is orthogonal to these approaches.

9 Conclusion

We propose a novel testing approach, Mocket, to fill the gap between the formal specification and the corresponding implementation of a distributed system. Given a TLA+ specification, Mocket can systematically check whether its corresponding system implementation conforms to the specification. We apply Mocket on three widely-used distributed systems, and unearth 3 previously unknown bugs.

10 Acknowledgments

We thank the anonymous reviewers for their constructive comments, and Marko Vukolić for shepherding this work. This work was partially supported by National Natural Science Foundation of China (62072444, U20A6003, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

References

[1] 1997. *Alloy*. Retrieved May 6, 2022 from https://alloytools.org/

[2] 2002. *ASM*. Retrieved April 22, 2021 from https://asm.ow2.io/

[3] 2008. *Apache Hadoop MapReduce*. Retrieved March 29, 2021 from https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[4] 2010. *Apache ZooKeeper*. Retrieved June 14, 2020 from https://zookeeper.apache.org/

[5] 2010. *The TLA+ Toolbox*. Retrieved May 6, 2022 from http://lamport.azurewebsites.net/tla/toolbox.html

[6] 2012. *Leader election never settles for a 5-node cluster*. Retrieved Oct 2, 2022 from https://issues.apache.org/jira/browse/ZOOKEEPER-1419

[7] 2013. *Zookeeper fails to start because of inconsistent epoch*. Retrieved Oct 2, 2022 from https://issues.apache.org/jira/browse/ZOOKEEPER-1653

[8] 2014. *Python NetworkX*. Retrieved May 6, 2022 from https://networkx.org/

[9] 2014. *TLA+ specification for the Raft consensus algorithm*. Retrieved May 6, 2022 from https://github.com/ongardie/raft.tla

[10] 2014. *ZKVerifier.java in SAMC*. Retrieved Feb 25, 2023 from https://github.com/wangsnowyin/samc/blob/master/src/edu/uchicago/cs/ucare/samc/zookeeper/ZKVerifier.java

[11] 2015. *Apache Spark*. Retrieved April 12, 2021 from https://spark.apache.org/

[12] 2016. *Revise TLA+ spec*. Retrieved May 6, 2022 from https://github.com/ongardie/raft.tla/pull/4/

[13] 2017. *Raft-java*. Retrieved April 11, 2022 from https://github.com/wenweihu86/raft-java

[14] 2017. *Raft-java issue#3*. Retrieved April 25, 2022 from https://github.com/wenweihu86/raft-java/issues/3

[15] 2017. *TiDB*. Retrieved April 13, 2022 from https://github.com/pingcap/tidb

[16] 2018. *Foundations of Azure Cosmos DB (Multi-Master) with Dr. Leslie Lamport*. Retrieved May 6, 2022 from https://www.youtube.com/watch?v=kYX6UrY_ooA

[17] 2018. *Lloyd's Estimates the Impact of a U.S. Cloud Outage at $19 Billion*. Retrieved May 6, 2022 from https://www.eweek.com/cloud/lloyd-s-estimates-the-impact-of-a-u.s.-cloud-outage-at-19-billion

[18] 2018. *xraft*. Retrieved April 11, 2022 from https://github.com/xnnyygn/xraft

[19] 2019. *Raft-java issue#19*. Retrieved April 25, 2022 from https://github.com/wenweihu86/raft-java/issues/19

[20] 2020. *CockroachDB*. Retrieved April 13, 2022 from https://github.com/cockroachdb/cockroach

[21] 2021. *GraphViz*. Retrieved Feb 17, 2023 from https://graphviz.org/

[22] 2022. *xraft commit: Handle with canceled votes*. Retrieved April 24, 2022 from https://github.com/xnnyygn/xraft/pull/28/commits/a48000080b6590402fbf45dd1a06af001d558830

[23] 2022. *xraft issue: Duplicate vote response can make illegal leader without a quorum*. Retrieved April 24, 2022 from https://github.com/xnnyygn/xraft/issues/27

[24] 2022. *xraft issue: VotedFor is not stored when a node is candidate and receives an AppendEntriesRpc*. Retrieved May 6, 2022 from https://github.com/xnnyygn/xraft/issues/29

[25] Basil Alkhatib, Sreeharsha Udayashankar, Sara Qunaibi, Ahmed Alquraan, Mohammed Alfatafta, Wael Al-Manasrah, Alex Depoutovitch, and Samer Al-Kiswany. 2022. Partial Network Partitioning. *ACM Transactions on Computer Systems* (2022).

[26] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 51–68.

[27] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding Exception-Related Bugs in Large-Scale Cloud Systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Automated Software Engineering (ASE)*. 339–351.

[28] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Automated Software Engineering (ASE)*. 536–547.

[29] Zhao Chen. 2020. *The Practice on Developing the Distributed Consensus Algorithm*. Peking University Press.

[30] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding Real-World Timeout Problems in Cloud Server Systems. In *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*. 1–11.

[31] A Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. eXtreme Modelling in Practice. *Proceedings of International Conference on Very Large Data Bases (VLDB)* 13, 9 (2020), 1346–1358.

[32] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 110–121.

[33] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 328–343.

[34] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 2093–2110.

[35] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of ACM SIGSOFT Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESCE/FSE)*. 539–550.

[36] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*.

[37] Yu Gao, Dong Wang, Qianwang Dai, Wensheng Dou, and Jun Wei. 2022. Common Data Guided Crash Injection for Cloud Systems. In *Proceedings of ACM/IEEE SIGSOFT International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. 36–40.

[38] Patrice Godefroid. 1994. *Partial-Order Methods for the Verification of Concurrent Systems-An Approach to the State-Explosion Problem*. University de Liege, Faculte des Sciences Appliquees.

[39] A Gravell, Yvonne Howard, Juan C Augusto, Carla Ferreira, and Stefan Gruner. 2011. Concurrent Development of Model and Implementation. *arXiv preprint arXiv:1111.2826* (2011).

[40] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*. 1–14.

[41] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023. Compiling Distributed System Models with PGo. In *Proceedings of ACM SIGARCH-SIGPLAN-SIGOPS International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 159–175.

[42] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 1–17.

[43] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. 245–256.

[44] Beom Heyn Kim, Taesoo Kim, and David Lie. 2022. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 383–398.

[45] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

[46] Leslie Lamport and Stephan Merz. 2017. Auxiliary Variables in TLA+. *arXiv preprint arXiv:1703.05121* (2017).

[47] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 399–414.

[48] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A Taxonomy of Non-deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 517–530.

[49] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–14.

[50] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. 2023. Performance Bug Analysis and Detection for Distributed Storage and Computing Systems. *ACM Transactions on Storage (TOS)* (2023), 1–31.

[51] Yishuai Li, Benjamin C Pierce, and Steve Zdancewic. 2021. Model-Based Testing of Networked Applications. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 529–539.

[52] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of ACM SIGARCH-SIGPLAN-SIGOPS International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 677–691.

[53] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-Fault Bugs in Cloud Systems. In *Proceedings of ACM SIGARCH-SIGPLAN-SIGOPS International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 419–431.

[54] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 559–574.

[55] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2145–2158.

[56] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 114–130.

[57] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–16.

[58] Ellis Michael, Doug Woos, Thomas Anderson, Michael D Ernst, and Zachary Tatlock. 2019. Teaching Rigorous Distributed Systems with Efficient Model Checking. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–15.

[59] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (2015), 66–73.

[60] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Stanford University.

[61] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 305–319.

[62] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized Testing of Distributed Systems with Probabilistic Guarantees. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages Applications (OOPSLA)*. 1–28.

[63] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace Aware Random testing for Distributed Systems. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages Applications (OOPSLA)*. 1–28.

[64] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *International Workshop on Systems Software Verification (SSV)*.

[65] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review (CCR)* 31, 4 (2001), 149–160.

[66] Dong Wang, Yu Gao, Wensheng Dou, and Jun Wei. 2022. DisTA: Generic Dynamic Taint Tracking for Java-Based Distributed Systems. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 547–558.

[67] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 357–368.

[68] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 229–244.

[69] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 213–228.

[70] Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *Proceedings of ACM SIGARCH-SIGPLAN-SIGOPS International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1141–1156.

[71] Pamela Zave. 2012. Using Lightweight Modelling to Understand Chord. *ACM SIGCOMM Computer Communication Review (CCR)* 42, 2 (2012), 49–57.

[72] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 116–131.

A Artifact Appendix

This artifact provides the source code of Mocket for EuroSys 2023 paper - "Model Checking Guided Testing for Distributed Systems". Mocket is designed to perform systematic testing for distributed systems guided by model checking TLA+ specifications. Mocket assumes that the TLA+ specification is the correct design for a distributed system, and finds bugs existing in the distributed system implementation that violates the TLA+ specification. Our artifact obtained the "Artifacts Available" badge from the Artifact Evaluation process of EuroSys 2023. The DOI of our artifact is https://doi.org/10.5281/zenodo.7654817.

Artifact repository. All the project source code including the instructions on how to build and run Mocket on distributed systems is available in the following git repository: https://github.com/tcse-iscas/Mocket.