Leveraging TLA$^+$ Specifications to Improve the Reliability of the ZooKeeperCoordination Service

Lingzhi Ouyang, Yu Huang(✉), Binyu Huang, and Xiaoxing Ma

State Key Laboratory for Novel Software Technology, Nanjing 210023, China
{lingzhi.ouyang,binyuhuang}@smail.nju.edu.cn, {yuhuang,xxm}@nju.edu.cn

Abstract. ZooKeeper is a coordination service, widely used as a backbone of various distributed systems. Though its reliability is of critical importance, testing is insufficient for an industrial-strength system of the size and complexity of ZooKeeper, and deep bugs can still be found. To this end, we resort to formal TLA$^+$ specifications to further improve the reliability of ZooKeeper. Our primary objective is usability and automation, rather than full verification. We incrementally develop three levels of specifications for ZooKeeper. We first obtain the *protocol specification*, which unambiguously specifies the Zab protocol behind ZooKeeper. We then proceed to a finer grain and obtain the *system specification*, which serves as the super-doc for system development. In order to further leverage the model-level specification to improve the reliability of the code-level implementation, we develop the *test specification*, which guides the explorative testing of the ZooKeeper implementation. The formal specifications help eliminate the ambiguities in the protocol design and provide comprehensive system documentation. They also help find critical deep bugs in system implementation, which are beyond the reach of state-of-the-art testing techniques. Our specifications have been merged into the official Apache ZooKeeper project.

Keywords: TLA$^+$ · ZooKeeper · Zab · Specification · Model checking

1 Introduction

ZooKeeper is a distributed coordination service for highly reliable synchronization of cloud applications [23]. ZooKeeper essentially offers a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. Its intended

This work is supported by the National Natural Science Foundation of China (62372222), the CCF-Huawei Populus Grove Fund (CCF-HuaweiFM202304), the Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (YBN2019105178SW38), the Fundamental Research Funds for the Central Universities (020214912222) and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

H. Hermanns et al. (Eds.): SETTA 2023, LNCS 14464, pp. 189–205, 2024.
https://doi.org/10.1007/978-981-99-8664-4_11

usage requires Zookeeper to provide strong consistency guarantees, which it does by running a distributed consensus protocol called Zab [25].

Consensus protocols are notoriously difficult to get right. The complex failure recovery logic results in an astronomically large state space, and deep "Heisenbugs" still often escape from intensive testing [31,32]. Toward this challenge, we resort to formal methods to improve the reliability of ZooKeeper. We do not aim to achieve full verification, but instead emphasize a high degree of automation and practical usability. Moreover, our primary goal is to improve the reliability of both the model-level design and the code-level implementation.

We adopt TLA$^+$ as our specification language. TLA$^+$ has been successful in verifying distributed concurrent systems, especially consensus protocols. Many consensus protocols, including Paxos, Raft and their derivatives, have their TLA$^+$ specifications published along with the protocol design and implementation [8,10,18,30]. However, the current usage of TLA$^+$ is mainly restricted to verification of the protocol design. Considering code-level implementation, model checking-driven test case generation is used to ensure the equivalence between two different implementations in MongoDB Realm Sync [16].

Our primary objective is to improve the reliability of ZooKeeper. We incrementally obtain three levels of specifications in TLA$^+$. We first obtain the *protocol specification*, which unambiguously specifies the Zab protocol behind ZooKeeper. We then proceed to a finer grain and obtain the *system specification*, which serves as the super-doc[1] for ZooKeeper development. In order to leverage the model-level specification to improve the reliability of the code-level implementation, we further develop the *test specification*, which guides the explorative testing of the ZooKeeper implementation. The formal specifications help eliminate the ambiguities in the protocol design and provide comprehensive system documentation. They also help find critical deep bugs in system implementation, which are beyond the reach of state-of-the-art testing techniques. Our specifications are available in the online repository [6]. Writing TLA$^+$ specifications for ZooKeeper was raised as an issue [3]. Our specifications have addressed this issue and been accepted by the Apache ZooKeeper project [11].

We start in Sect. 2 by introducing the basics of ZooKeeper and TLA$^+$. We present our three levels of specifications in Sects. 3, 4 and 5. In Sect. 6, we discuss the related work. Finally, Sect. 7 concludes with a discussion on the benefits of and the potential extensions to our formal specification practices.

2 ZooKeeper and TLA$^+$

2.1 ZooKeeper and Zab

ZooKeeper is a fault-tolerant distributed coordination service used by a variety of distributed systems [14,23,35]. These systems often consist of a large number of processes and rely upon ZooKeeper to perform essential coordination tasks, such

[1] Super-doc refers to the precise, concise and testable documentation of the system implementation, which can be explored and experimented on with tools [37].

as maintaining configuration information, storing status of running processes and group membership, providing distributed synchronization and managing failure recovery.

Due to the significant reliance of large applications on ZooKeeper, the service must maintain a high level of availability and possess the ability to mask and recover from failures. ZooKeeper achieves availability and reliability through replication and utilizes a primary-backup scheme to maintain the states of replica processes consistent [13,24,25]. Upon receiving client requests, the primary generates a sequence of non-commutative state changes and propagates them as transactions to the backup replicas using *Zab*, the ZooKeeper atomic broadcast protocol. The protocol consists of three phases: DISCOVERY, SYNC and BROADCAST. Its primary duties include agreeing on a leader in the ensemble, synchronizing the replicas, managing the broadcast of transactions, and recovering from failures.

To ensure progress, ZooKeeper requires that a majority (or more generally a quorum) of processes have not crashed. Any minority of processes may crash at any moment. Crashed processes are able to recover and rejoin the ensemble. For a process to perform the primary role, it must have the support of a quorum of processes.

2.2 TLA$^+$ Basics

TLA$^+$ (Temporal Logic of Actions) is a lightweight formal specification language that is particularly suitable for designing distributed and concurrent systems [7]. In contrast to programming languages, TLA$^+$ employs simple mathematics to express concepts in a more elegant and precise manner. In TLA$^+$, a system is specified as a state machine by defining the possible initial *states* and the allowed *actions*, i.e., state transitions. Each state represents a global state of the system. Whenever all *enabling conditions* of a state transition are satisfied in a given *current* state, the system can transfer to the *next* state by applying the action.

One of the notable advantages of TLA$^+$ is its ability to handle different levels of abstraction. Correctness properties and system designs can be regarded as steps on a ladder of abstraction, with correctness properties occupying higher levels, system designs and algorithms in the middle, and executable code at the lower levels [37]. The flexibility to choose and adjust levels of abstraction makes TLA$^+$ a versatile tool suited to a wide range of needs.

TLA$^+$ provides the TLC model checker, which builds a finite state model from the specifications for checking invariant safety properties (in this work, we mainly focus on safety properties and do not consider liveness properties). TLC first generates a set of initial states that satisfy the specification, and then traverses all possible state transitions. If TLC discovers a state violating an invariant property, it halts and provides the trace leading to the violation. Otherwise, the system is verified to satisfy the invariant property. With TLC, we are able to explore every possible behavior of our specifications without additional human effort. As a result, we can identify subtle bugs and edge cases that may not be exposed through other testing or debugging techniques.

3 Protocol Specification

We first develop the *protocol specification*, i.e., specification of the Zab protocol based on its informal description [23,25]. The protocol specification aims at precise description and automatic model checking of the protocol design. It also serves as the basis for further refinements, as detailed in Sects. 4 and 5.

3.1 Development of Protocol Specification

The protocol specification necessitates a comprehensive description of the design of Zab, along with the corresponding correctness conditions that must be upheld. In the following, we present our practice of developing the TLA$^+$ specifications for both aspects.

Specification of Zab. The "upon-do clauses" in the Zab pseudocode can be readily transformed to the enabling conditions and actions in TLA$^+$. This feature greatly simplifies the task of obtaining the initial skeleton of the protocol specification. The real obstacle lies in handling the ambiguities and omissions in the informal design. We have three challenges to address, as detailed below.

First, we need to cope with the ambiguities concerning the abstract mathematical notion of *quorum*. ZooKeeper is a leader-based replicated service. The leader requires certain forms of acknowledgements from a quorum of followers to proceed. Though the notion of quorum greatly simplifies presentation of the basic rationale of Zab, it introduces subtle ambiguities in the design. Specifically, the set Q, which denotes the quorum of followers in the Zab pseudocode, is severely overloaded in the informal design. It refers to different sets of followers in different cases, as shown in Fig. 1. In the TLA$^+$ specification, we must use separate variables for different quorums in different situations, e.g., variable cepochRecv for the quorum acknowledging the CEPOCH message and ackeRecv for the quorum acknowledging the ACKEPOCH message.

Second, the design of Zab mainly describes the "happy case", in which the follower successfully contacts the leader and the leader proceeds with support from a quorum of followers. In our TLA$^+$ specification, we must precisely describe the unhappy case where the leader has not yet received sufficient acknowledgements from the followers. We must also explicitly model failures in the execution environment, enabling the TLC model checker to exhaustively exercise the fault-tolerance logic in Zab. Moreover, Zab is a highly concurrent protocol. Actions of ZooKeeper processes and environment failures can interleave. The complex interleavings are not covered in the protocol design, and we supplement the detailed handling of the interleavings in our TLA$^+$ specification.

Third, the Zab protocol does not implement leader election, but relies on an assumed leader oracle. In our TLA$^+$ specification, we use a variable called leaderOracle to denote the leader oracle. The leader oracle exposes two actions UpdateLeader and FollowLeader to update the leader and to find who is the leader, respectively.

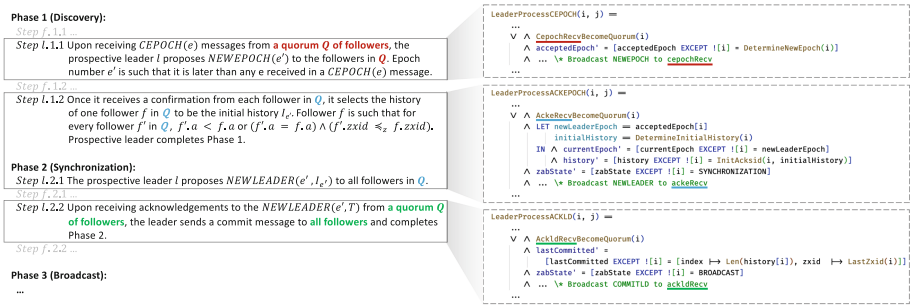

Fig. 1. From informal design to formal specification. In the informal design (Left), the set Q, which denotes the quorum of followers, is ambiguous and overloaded. In the TLA+ specification (Right), the set Q is specified with different variables (cepochRecv, ackeRecv and ackldRecv) for different quorums in different situations.

The details of protocol design we supplement in the TLA+ specification are verified by model checking. It is also confirmed by our development of the system specification (see Sect. 4).

Specification of Correctness Conditions. We specify two types of correctness conditions, the *core correctness conditions* and the *invariant properties* (with a focus on safety properties in this work). The Zab protocol [25] prescribes six core correctness conditions, namely *integrity*, *total order*, *agreement*, *local primary order*, *global primary order* and *primary integrity*. These properties are extracted from the requirement analysis of the ZooKeeper coordination service. They constrain the externally observable behavior from the user client's perspective.

Designers usually do not directly guarantee the core correctness conditions when designing a complex protocol like Zab. Rather, they decompose the core correctness conditions into a collection of invariant properties. In principle, the invariants maintained in different parts of the protocol can collectively ensure the core correctness conditions. Model checking against the invariants can also accelerate the detection of bugs in protocol design. We extract the invariant properties based on our understanding of the Zab protocol. We are also inspired by the invariants specified for Paxos, Raft and their derivatives [8,10,30,38].

3.2 Ensuring Quality of Protocol Specification

Upon completing the development of the protocol specification, we perform model checking to ensure its quality. The protocol specification is amenable to automatic exploration by the TLC model checker. We utilize TLC in two modes: the standard *model-checking mode*, in which TLC traverses the state space in a BFS manner, and the *simulation mode*, where TLC keeps sampling execution paths of a predefined length until it uses up the testing budget.

In this work, we perform model checking on a PC equipped with an Intel I5-9500 quad-core CPU (3.00 GHz) and 16 GB RAM, running Windows 10 Enterprise. The software used is TLA$^+$ Toolbox v1.7.0. We first tame the state explosion problem for the model checking and identify subtle bugs resulting from ambiguities in the informal design. Then we conduct further model checking to verify the correctness of the protocol specification.

Taming State Explosion. Model checking suffers from the state explosion problem, making it impractical to fully check models of arbitrary scales. To mitigate this issue, we prune the state space by tuning the enabling conditions of key actions in the protocol specification. The basic rationale behind this is to constrain the scheduling of meaningless events. For example, too many failure events are meaningless when the leader does not even contact the followers, and such scheduling should be ruled out during the checking process.

Furthermore, we directly limit the state space based on the small-scope hypothesis, which suggests that analyzing small system instances suffices in practice [34]. Specifically, we control the scale of the model by restricting the following configuration parameters: the number of servers, the maximum number of transactions, the maximum number of timeouts, and the maximum number of restarts. The server count is confined to a maximum of 5, as it is sufficient to trigger most invariant violations in most cases according to ZooKeeper's bug tracking system [1]. Similarly, the number of transactions is limited to a small value, as it already suffices to cause log differences among servers.

It is also worth noting that in the protocol specification, we model failures as Timeout and Restart mainly for state space reduction. These two actions can effectively describe the effects of multiple failures in the execution environment.

Finding Ambiguities in the Informal Design. Following the above techniques to mitigate the state explosion, we perform model checking and find two invariant violations in the preliminary version of our protocol specification. One is due to the misuse of the quorum of followers. The other concerns how the leader responds to a recovering follower in between logging a transaction and broadcasting it to the followers. The paths to the violations help us find the root cause and fix the bugs in the protocol specification. Our fixes are also in accordance with the system implementation (see Sect. 4), though the implementation contains more details.

Verifying Correctness. After resolving the aforementioned bugs in the protocol specification, we proceed with model checking to ensure its correctness. We adjust the scale of the model using the parameters specified in the model configuration mentioned earlier. For each configuration, we record the checking mode, the number of explored states, and the checking time cost. For the model checking mode, we also record the diameter of the state space graph generated

by the TLC model checker. For the simulation mode, we set the maximum length of the trace to 100. We restrict the model checking time to 24 h and the disk space used to 100 GB.

Table 1. Model checking results of the protocol specification.

Config*	Checking mode	Diameter	Num of states	Time cost
(2, 2, 2, 0)	Model-checking	38	19,980	00 : 00 : 03
(2, 2, 0, 2)	Model-checking	38	25,959	00 : 00 : 04
(2, 2, 1, 1)	Model-checking	38	26,865	00 : 00 : 04
(2, 3, 2, 2)	Model-checking	60	10,370,967	00 : 06 : 58
(3, 2, 1, 0)	Model-checking	43	610,035	00 : 00 : 28
(3, 2, 0, 1)	Model-checking	50	1,902,139	00 : 02 : 36
(3, 2, 2, 0)	Model-checking	54	26,126,204	00 : 17 : 07
(3, 2, 0, 2)	Model-checking	68	245,606,642	03 : 41 : 23
(3, 2, 1, 1)	Model-checking	61	84,543,312	01 : 00 : 18
(3, 2, 2, 1)	Model-checking	50	1,721,643,089	> 24 : 00 : 00
(3, 2, 1, 2)	Model-checking	46	1,825,094,679	> 24 : 00 : 00
(3, 3, 3, 3)	Simulation	–	1,194,558,650	> 24 : 00 : 00
(4, 2, 1, 0)	Model-checking	64	21,393,294	00 : 23 : 29
(4, 2, 0, 1)	Model-checking	71	79,475,010	01 : 37 : 31
(4, 2, 2, 0)	Model-checking	57	1,599,588,210	> 24 : 00 : 00
(5, 2, 3, 3)	Simulation	–	1,044,870,264	> 24 : 00 : 00
(5, 3, 2, 2)	Simulation	–	817,181,422	> 24 : 00 : 00

* In the protocol specification, the **Config** parameters represent the number of servers, the maximum number of transactions, the maximum number of timeouts, and the maximum number of restarts.

Table 1 presents statistics regarding the model checking of the protocol specification. The explorations shown in the table cover a variety of configurations, with the server count ranging from 2 to 5, and the maximum number of transactions, timeouts, and restarts up to 3. Within the time limit of 24 h, the explorations of all configurations do not exceed the space limit. When limiting the model to a relatively small scale, the model-checking mode can exhaustively explore all possible interleavings of actions. In contrast, the simulation mode tends to explore deeper states. All specified correctness conditions are met without violation during the explorations in models of diverse configurations. Based on the results and the small-scope hypothesis [34], we have achieved a high level of confidence in the correctness of our protocol specification.

Moreover, we further tweak the specification a little and see if the model checker can find the planted errors. For instance, in one trial, we modified the definition of the constant Quorums in the specification, which originally denotes

a set of all majority sets of servers, to include server sets that comprise no less than half of all servers. In expectation, the modification will lead to invariant violations only when the number of servers is even. We executed model checking on the modified specification, and as anticipated, no violations occurred when the server number was 3 or 5. However, in the case of 2 or 4 servers, invariant violations emerged, such as two established leaders appearing in the same epoch. Such trials illustrate the effectiveness of the specified correctness conditions and further indicate the correctness of the protocol specification. More details about the verification of the protocol specification can be found in [6].

4 System Specification

Given the protocol specification, we further develop the system specification, which serves as the super-doc supplementing detailed system documentation of Zab implementation for the ZooKeeper developers. In the following, we first discuss the essentials of a system specification written in TLA$^+$. Then, we present the practice of developing the system specification and the approach to ensuring its quality.

4.1 Essentials of a Super-Doc in TLA$^+$

To develop the system specification, we first need to decide which should and should not be included in the specification. We fathom out the right level of abstraction from two complementary perspectives, namely what the system developers need from a super-doc and what the TLA$^+$ specification language can express.

As a super-doc, the system specification should reconcile accuracy with simplicity. When we submitted the preliminary version of the system specification to the ZooKeeper community [22], a basic principle the community suggests is that "whether or not an action should be retained in the specification depends on whether it is critical to reveal important details of the protocol". Further suggestions include "implement the minimum required" and "keep things simpler". These suggestions guide us to calibrate the level of abstraction. In principle, the system specification should cover every aspect of the ZooKeeper system. Meanwhile, low-level details, e.g. the leader-follower heartbeat interactions and internal request queues for blocked threads, can be omitted. We also inherit the modularity of the system implementation in our specification.

The precision of the specifications written in TLA$^+$ is intended to uncover design flaws. TLA$^+$ specifications mostly consist of ordinary non-temporal mathematics, e.g., basic set theory, which is less cumbersome than a purely temporal specification. A major advantage of TLA$^+$ is that it frees us from complex programming considerations like multi-threading and object encapsulation. When enjoying the simplicity of TLA$^+$, we inevitably need to handle the gap in expressiveness between TLA$^+$ and the programming language (Java in the ZooKeeper case). In ZooKeeper, the block-and-wakeup style multi-thread programming is

heavily used, while in TLA$^+$, actions are executed in an asynchronous style. We decompose the block-and-wakeup thread logic into two conditional branches in one TLA$^+$ action. The timing of scheduling the wakeup logic is encoded in the entry conditions of the branches. Moreover, we combine the wakeup logic of multiple threads in one conditional branch in the TLA$^+$ action. This not only improves specification readability, but also helps mitigate the state explosion.

4.2 Development of the Super-Doc

The system specification is in principle a refinement of the protocol specification. Details in the system specification are supplemented based on the source code, as shown in Fig. 2. ZooKeeper inherits the basic architecture of Zab, and we discuss each of its modules in turn.

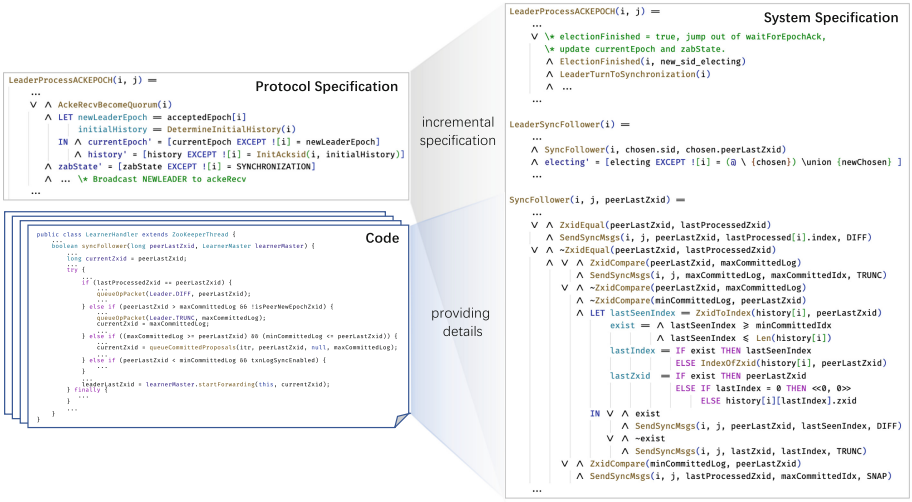

Fig. 2. Incremental development of the system specification. The system specification is in principle a refinement of the protocol specification, with supplementary details derived from the source code.

ZooKeeper implements its own Fast Leader Election (FLE) algorithm [35], which is omitted in Zab. FLE elects a leader which has the most up-to-date history. We also extract the invariant properties FLE should maintain from its design. Moreover, to conduct "unit test" on the FLE design, i.e., to model check the FLE module alone, we simulate the interaction between FLE and its outside world, e.g., actions updating the epoch or the transaction ID (zxid).

Compared with Zab, the DISCOVERY module in ZooKeeper is simplified, since the leader already has the most up-to-date history. We reuse most part of this module in the protocol specification, and make several revisions according to the implementation. Specifically, the follower does not need to send its full

history to the leader, and only needs to send the latest zxid it has. The leader checks the validity of each follower's state rather than updates its history based on all histories received from the followers.

In Zab, the leader sends its full history to the followers in the SYNC phase. This is obviously a simplification for better illustration of the design rationale. In the implementation, the SYNC module is significantly optimized for better performance. We rewrite the system specification for this module based on the implementation. Specifically, NEWLEADER acts as a signaling message without carrying concrete history data. The leader's history will be synchronized to the followers in one of three modes, namely DIFF, TRUNC, and SNAP. The leader will select the right mode based on the differences between its history and the follower's latest zxid. The follower may update its log or snapshot according to the sync mode. These supplemented details in the system specification are confirmed by the system implementation. The BROADCAST module is basically inherited from the protocol specification.

In order to facilitate conformance checking (see Sect. 4.3), we also refine the failure modeling of the protocol specification. Specifically, we model environment failures as node crash/rejoin events and network partition/reconnection events in the system specification. These failure events are more fundamental and can generate the failures modeled in the protocol specification.

Correctness conditions, including core correctness conditions and invariant properties, are mainly inherited from the protocol specification. Table 2 presents the model checking results of the system specification under certain configuration parameters. No violation of correctness conditions is found during the checking.

Table 2. Model checking results of the system specification.

Config *	Checking mode	Diameter	Num of states	Time cost
$(3, 2, 3, 3)$	Model-checking	24	$3,322,996,126$	$> 24:00:00$
$(5, 2, 3, 3)$	Model-checking	16	$693,381,547$	$> 24:00:00$
$(3, 5, 5, 5)$	Simulation	–	$1,139,420,778$	$> 24:00:00$
$(5, 5, 5, 5)$	Simulation	–	$1,358,120,544$	$> 24:00:00$
$(3, 5, 0, 10)$	Simulation	–	$1,463,314,104$	$> 24:00:00$
$(3, 5, 10, 0)$	Simulation	–	$1,211,089,513$	$> 24:00:00$

* In the system specification, the **Config** parameters represent the number of servers, the maximum number of transactions, the maximum number of node crashes, and the maximum number of network partitions.

4.3 Ensuring Quality of the Super-Doc

The quality of the super-doc primarily depends on whether the doc precisely reflects the system implementation, with unimportant details omitted. Note that model checking can only ensure that the specification satisfies the correctness conditions. It cannot tell whether the specification precisely reflects the system implementation.

We conduct conformance checking between the system specification and the system implementation to ensure the quality of the specification, as shown in Fig. 3. We first let the TLC model checker execute the system specification and obtain the model-level execution traces. We extract the event schedule from the model checking trace, and then control the system execution to follow the event schedule.

The controlled system execution is enabled by the Model Checking-driven Explorative Testing (MET) framework [5]. We first instrument the ZooKeeper system, which enables the test execution environment to intercept the communication between the ZooKeeper nodes, as well as local events of interest, e.g., logging transactions. The intercepted events are dispatched according to the event schedule extracted from the model checking trace. In this way, we control the system execution to "replay" the model checking trace, and check the conformance between these two levels of executions.

Once the conformance checking fails, discrepancies between the specification and the implementation are detected. The specification developer checks the discrepancies and revises the specification based on the implementation. After multiple rounds of conformance checking, the system specification obtains sufficient accuracy. This process is analogous to the regression testing [15].

During our practice, we discovered several discrepancies between the specification and the implementation. For example, in the initial version of the system specification, it was assumed that whenever the leader processes a write request, the client session has already been successfully established. However, client session creation is also considered a transaction in ZooKeeper and requires confirmation by a quorum of servers. This discrepancy was identified during conformance checking, and the specification was subsequently revised to address it. Further details about the system specification can be found in [6].

5 Test Specification

Given the protocol and system specifications, we further develop the test specification, in order to guide the explorative testing of ZooKeeper. The basic workflow of explorative testing following the MET framework is shown in Fig. 3. We detail the three core parts of the framework below.

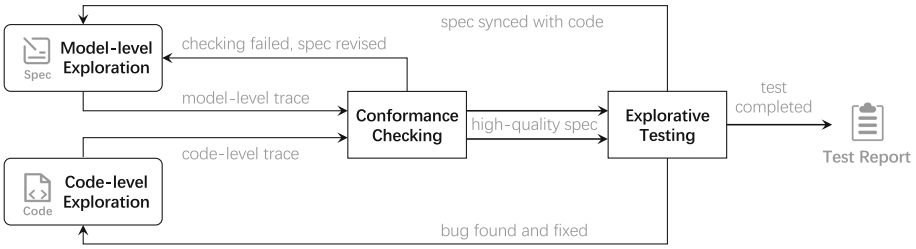

Fig. 3. Model checking-driven explorative testing.

5.1 Obtaining the Test Specification

In the explorative testing, we mainly focus on the recovery logic of ZooKeeper in its SYNC phase (though MET can be applied to each module of ZooKeeper). This module is heavily optimized in practice. It is under continual revision and deep bugs can still be found, according to ZooKeeper's bug tracking system [1].

The test specification is in principle a refinement of the system specification toward the source code. The test specification first inherits the system specification. The part of the specification to be refined, which corresponds to the part of the system under test, can be obtained by copy-pasting the ZooKeeper source code and manually updating the syntax to be valid TLA^+ specification [16]. Due to the inherent gap in the expressiveness, certain details are inevitably omitted or transformed in the test specification, including low-level programming considerations like multi-threading. The developer can also intentionally omit certain details if they deem such details irrelevant to the potential deep bugs. For example, we do not explicitly model the client and "pack" the workload generation logic inside the leader.

Due to the state explosion, we cannot model check the test specification of the whole ZooKeeper system. In practice, we follow the Interaction-Preserving Abstraction (IPA) framework [19]. We refine one single module to its test specification, while keeping other modules as abstract as possible, with the interactions between modules preserved. As we conduct MET on the SYNC module, we abstract all other modules. For example, we combine ELECTION and DISCOVERY into one action, while their interactions with the SYNC module are preserved.

5.2 Improving the Accuracy of Specification

The quality of the testing specification is also incrementally improved by multiple rounds of conformance checking (see Sect. 4). Typically, we find a number of discrepancies. The developer may need to fine-tune the test specification to better conform to the code. He may also neglect the discrepancy if he confirms that the discrepancy is due to the inherent differences in expressiveness and is irrelevant to the potential deep bug we try to find. The conformance checking keeps going like the regression testing until no discrepancy can be found. This means that the test specification is (sufficiently) accurate to guide the explorative testing.

5.3 Test Specification in Action

With the help of our test specification and the MET framework, we stably trigger and reproduce several critical deep bugs in ZooKeeper [6]. Here, we use ZK-2845 [2] and ZK-3911 [4] as examples to demonstrate the effectiveness and efficiency of this approach. These two bugs will result in the inconsistency of accepted logs or the loss of committed logs, which can be particularly severe for a coordination service like ZooKeeper. However, similar to other deep bugs, they are difficult to

uncover and reproduce. Triggering these bugs typically requires numerous steps, and the timing of failures is subtle. The space of all possible bug-triggering patterns is so vast that it is beyond human reasoning. We can only find the bugs by explorative search guided by model checking.

Table 3 lists the statistics related to the invariant violations of ZK-2845 and ZK-3911. As indicated, we can obtain the traces of these two bugs within a short time by model checking against the invariants. The high efficiency is mainly attributed to the test specification, which abstracts irrelevant details while preserving the necessary information.

Table 3. Invariant violations of ZK-2845 and ZK-3911.

Bug	Invariant violation *	Simulation mode		Model-checking mode	
		Len.	Time cost	Len.	Time cost
ZK-2845	ProcessConsistency	23	00 : 00 : 02	10	00 : 00 : 12
	ProposalConsistency	20	00 : 00 : 03	11	00 : 00 : 18
ZK-3911	LeaderLogCompleteness	25	00 : 01 : 29	14	00 : 00 : 42
	MonotonicRead	39	00 : 01 : 35	18	00 : 13 : 13

* The column **Invariant violation** lists the violated invariants of the bugs. The definitions of these invariants can be found in the test specification. The results in the table are obtained using the configuration of 3 servers, 2 transactions in max, 4 node crashes in max, and 4 network partitions in max.

Trace analysis reveals that a bug may violate multiple invariants that represent different aspects of the system's requirements. For instance, for the two invariants violated by ZK-3911, LeaderLogCompleteness constrains the internal behavior from the developer's perspective, while MonotonicRead constrains the externally observable behavior from the user client's perspective. The quality of the invariants specified in the test specification significantly affects the efficiency of bug detection. Typically, invariants that constrain internal behaviors can expedite the bug-triggering process compared to the invariants that constrain external behaviors.

The two checking modes of TLC exhibit different capabilities in triggering invariant violations. In most cases, the simulation mode is typically faster and more effective in detecting deeper bugs since it tends to explore deeper states. Conversely, the model-checking mode is better suited for searching for the shortest trace that leads to an invariant violation.

It is worth noting that exposing these bugs through model checking on the system specification can be challenging (see Table 2). The human knowledge behind the development of the test specification plays a crucial role in pruning the state space and accelerating the bug-triggering process. With the flexibility to adjust levels of abstraction in TLA$^+$, one can generate the test specification from the system specification at a low cost. Besides, TLC enables efficient explorations without additional human effort, and Met allows us to replay the traces

of invariant violations in the system to validate their authenticity. More bugs exposed by our approach are detailed in [6].

6 Related Work

Specification in TLA$^+$. TLA$^+$ is widely used for the specification and verification of distributed protocols in both academia and industry. Lamport *et al.* utilized TLA$^+$ to specify the original Paxos protocol [30], as well as various Paxos variants, including Disk Paxos [18], Fast Paxos [28], and Byzantine Paxos [29]. These protocols were also verified to be correct using TLC. Diego Ongaro provided a TLA$^+$ specification for the Raft consensus algorithm and further verified its correctness through model checking [10]. Yin *et al.* employed TLA$^+$ to specify and verify three properties of the Zab protocol [42]. Moraru *et al.* utilized TLA$^+$ to specify EPaxos when first introducing the protocol [36].

In industry, Amazon Web Services (AWS) extensively employs TLA$^+$ to help solve subtle design problems in critical systems [37]. Microsoft's cloud service Azure leverages TLA$^+$ to detect deeply-hidden bugs in the logical design and reduce risk in their system [12]. PolarFS also uses TLA$^+$ to precisely document the design of its ParallelRaft protocol, effectively ensuring the reliability and maintainability of the protocol design and implementation [20]. WeChat's storage system PaxosStore specifies its consensus algorithm TPaxos in TLA$^+$ and verifies its correctness using TLC to increase confidence in the design [9].

The practices mentioned above utilize TLA$^+$ to specify and verify distributed protocols with the goal of identifying design flaws and increasing confidence in the core protocol design. However, they do not address the code-level implementation and cannot guarantee that the specification accurately reflects the system implementation. Discrepancies between the specification and the implementation can result from transcription errors, and model checking is solely responsible for verifying the specification. Our TLA$^+$ specifications for ZooKeeper focus on both the protocol design and the system implementation. Based on the source code and the protocol specification, we incrementally develop the system specification that serves as the super-doc for the ZooKeeper developers. Additionally, we conduct conformance checking between the system specification and the system implementation to eliminate discrepancies between them and ensure the quality of the specification.

Model Checking-Driven Testing on ZooKeeper. Model checking-driven testing has been extensively employed in distributed systems such as ZooKeeper. The FATE and DESTINI framework systematically exercises multiple combinations of failures in cloud systems and utilizes declarative testing specifications to support the checking of expected behaviors [21]. This framework has been effectively used to reproduce several bugs in ZooKeeper. SAMC incorporates semantic information into state-space reduction policies to trigger deep bugs in ZooKeeper [32]. FlyMC introduces state symmetry and event independence to reduce the state-space explosion [33]. PCTCP employs a randomized scheduling algorithm

for testing distributed message-passing systems [39], while taPCT integrates partial order reduction techniques into random testing [40]. Both approaches have been utilized to detect bugs in ZooKeeper's leader election module. Modulo utilizes divergence resync models to systematically explore divergence failure bugs in ZooKeeper [26]. The aforementioned works explore ZooKeeper based on implementation-level model checkers.

In contrast, inspired by the practice of eXtreme Modelling [16] and other test case generation techniques with TLA$^+$[17,27], we leverage the TLA$^+$ specification and the TLC model checker to guide the explorative testing of ZooKeeper. TLC is highly efficient at exploring long traces and uncovering subtle deep bugs that require multiple steps to trigger, making it a powerful tool for test case generation. Similarly, Mocket uses TLC to guide the testing and reproduces bugs in ZooKeeper [41]. We further reduce the state space by taking advantage of the flexibility of the TLA$^+$ specification, which can be integrated with human knowledge at a low cost. We develop a test specification that efficiently triggers bugs in ZooKeeper, further enhancing the effectiveness of our model checking-driven explorative testing framework.

7 Conclusion and Future Work

In this work, we use TLA$^+$ to present precise design of and provide detailed documentation for ZooKeeper. We also use model checking to guide explorative testing of ZooKeeper. The formal specifications well complement state-of-the-art testing techniques and further improve the reliability of ZooKeeper.

In our future work, we will use TLA$^+$ specifications in more distributed systems, e.g., cloud-native databases and distributed streaming systems. Enabling techniques, such as taming of state explosion and deterministic simulation of system execution, also need to be strengthened.

References

1. Apache ZooKeeper's issue tracking system. https://issues.apache.org/jira/projects/ZOOKEEPER/issues
2. Issue: ZK-2845. https://issues.apache.org/jira/browse/ZOOKEEPER-2845
3. Issue: ZK-3615. https://issues.apache.org/jira/browse/ZOOKEEPER-3615
4. Issue: ZK-3911. https://issues.apache.org/jira/browse/ZOOKEEPER-3911
5. MET. https://github.com/Lingzhi-Ouyang/MET
6. Three levels of TLA$^+$ specifications for ZooKeeper. https://github.com/Disalg-ICS-NJU/zookeeper-tla-spec
7. TLA$^+$ home page. https://lamport.azurewebsites.net/tla/tla.html
8. TLA$^+$ specification for Paxos. https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Paxos.tla
9. TLA$^+$ specification for PaxosStore. https://github.com/Starydark/PaxosStore-tla
10. TLA$^+$ specification for Raft. https://github.com/ongardie/raft.tla
11. TLA$^+$ specifications for the Apache ZooKeeper project. https://github.com/apache/zookeeper/tree/master/zookeeper-specifications

12. The use of TLA$^+$ in industry. https://lamport.azurewebsites.net/tla/industrial-use.html
13. Zab's wiki. https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab1.0
14. ZooKeeper home page. https://zookeeper.apache.org/
15. Bourque, P., Fairley, R.E., Society, I.C.: Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0, 3rd edn. IEEE Computer Society Press, Washington, DC, USA (2014)
16. Davis, A.J.J., Hirschhorn, M., Schvimer, J.: eXtreme modelling in practice. Proc. VLDB Endow. **13**(9), 1346–1358 (2020). https://doi.org/10.14778/3397230.3397233
17. Dorminey, S.: Kayfabe: model-based program testing with TLA$^+$/TLC. Technical report, Microsoft Azure WAN (2020). https://conf.tlapl.us/2020/11-Star_Dorminey-Kayfabe_Model_based_program_testing_with_TLC.pdf
18. Gafni, E., Lamport, L.: Disk Paxos. Distrib. Comput. **16**(1), 1–20 (2003). https://doi.org/10.1007/s00446-002-0070-8
19. Gu, X., Cao, W., Zhu, Y., Song, X., Huang, Y., Ma, X.: Compositional model checking of consensus protocols specified in TLA$^+$ via interaction-preserving abstraction. In: Proceedings of International Symposium on Reliable Distributed Systems (SRDS 2022). IEEE (2022). https://doi.org/10.1109/srds55811.2022.00018
20. Gu, X., Wei, H., Qiao, L., Huang, Y.: Raft with out-of-order executions. Int. J. Softw. Informatics **11**(4), 473–503 (2021). https://doi.org/10.21655/ijsi.1673-7288.00257
21. Gunawi, H.S., et al.: FATE and DESTINI: a framework for cloud recovery testing. In: Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation. p. 239 (2011). https://dl.acm.org/doi/10.5555/1972457.1972482
22. Huang, B., Ouyang, L.: Pull request for ZOOKEEPER-3615: provide formal specification and verification using TLA$^+$ for Zab #1690. https://github.com/apache/zookeeper/pull/1690
23. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of ATC 2010, USENIX Annual Technical Conference, pp. 145–158. USENIX (2010). https://dl.acm.org/doi/10.5555/1855840.1855851
24. Junqueira, F.P., Reed, B.C., Serafini, M.: Dissecting Zab. Technical report, YL-2010-007, Yahoo! Research, Sunnyvale, CA, USA (2010). https://cwiki.apache.org/confluence/download/attachments/24193444/yl-2010-007.pdf
25. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: Proceedings of DSN 2011, IEEE/IFIP Conference on Dependable Systems and Networks, pp. 245–256. IEEE (2011). https://doi.org/10.1109/DSN.2011.5958223
26. Kim, B.H., Kim, T., Lie, D.: Modulo: finding convergence failure bugs in distributed systems with divergence resync models. https://www.usenix.org/conference/atc22/presentation/kim-beom-heyn
27. Kuprianov, A., Konnov, I.: Model-based testing with TLA$^+$ and Apalache. Technical report, Informal Systems (2020). https://conf.tlapl.us/2020/09-Kuprianov_and_Konnov-Model-based_testing_with_TLA_+_and_Apalache.pdf
28. Lamport, L.: Fast Paxos. Distrib. Comput. **19**(2), 79–103 (2006). https://doi.org/10.1007/s00446-006-0005-x
29. Lamport, L.: The PlusCal Code for Byzantizing Paxos by Refinement. TechReport, Microsoft Research (2011)
30. Lamport, L., Merz, S., D, D.: A TLA$^+$ specification of Paxos and its refinement (2019). https://github.com/tlaplus/Examples/tree/master/specifications/Paxos

31. Leesatapornwongsa, T., Gunawi, H.S.: SAMC: a fast model checker for finding Heisenbugs in distributed systems (demo). In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pp. 423–427. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2771783.2784771
32. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, pp. 399–414. USENIX Association, Berkeley (2014). https://dl.acm.org/doi/10.5555/2685048.2685080
33. Lukman, J.F., et al.: Flymc: highly scalable testing of complex interleavings in distributed systems. In: Proceedings of the Fourteenth EuroSys Conference 2019, pp. 1–16 (2019). https://doi.org/10.1145/3302424.3303986
34. Marić, O., Sprenger, C., Basin, D.: Cutoff bounds for consensus algorithms. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 217–237. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_12
35. Medeiros, A.: ZooKeeper's atomic broadcast protocol: theory and practice (2012). https://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf
36. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in egalitarian parliaments. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 358–372 (2013). https://doi.org/10.1145/2517349.2517350
37. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
38. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC 2014, pp. 305–320. USENIX Association, Berkeley (2014). https://dl.acm.org/doi/10.5555/2643634.2643666
39. Ozkan, B.K., Majumdar, R., Niksic, F., Befrouei, M.T., Weissenbacher, G.: Randomized testing of distributed systems with probabilistic guarantees. Proc. ACM Program. Lang. **2**(OOPSLA), 1–28 (2018). https://doi.org/10.1145/3276530
40. Ozkan, B.K., Majumdar, R., Oraee, S.: Trace aware random testing for distributed systems. Proc. ACM Program. Lang. **3**(OOPSLA), 1–29 (2019). https://doi.org/10.1145/3360606
41. Wang, D., Dou, W., Gao, Y., Wu, C., Wei, J., Huang, T.: Model checking guided testing for distributed systems. In: Proceedings of the Eighteenth European Conference on Computer Systems, pp. 127–143 (2023). https://doi.org/10.1145/3552326.3587442
42. Yin, J.Q., Zhu, H.B., Fei, Y.: Specification and verification of the Zab protocol with TLA$^+$. J. Comput. Sci. Technol. **35**, 1312–1323 (2020). https://doi.org/10.1007/s11390-020-0538-7