UNIVERSITÉ PARIS CITÉ

École doctorale 386
Laboratoire IRIF

&

CHENNAI MATHEMATICAL INSTITUTE

Automated testing of distributed protocol implementations

by SRINIDHI NAGENDRA

Supervised by CONSTANTIN ENEA
and by MANDAYAM SRIVAS

Presented at a public defense on 19/12/2024

Jury composition :

STEPHAN MERZ, DR	Université de Lorraine	Rapporteur
GENNARO PARLATO, PROF	University of Molise	Rapporteur
CEZARA DRAGOI, CR	Amazon Web Services	Examinatrice
PRAVEEN M, ASSOCIATE PROF	Chennai Mathematical Institute	Examinateur
BURCU OZKAN, ASSISTANT PROF	Delft University of Technology	Examinatrice
TAYSSIR TOUILI, DR	CNRS	Examinatrice
CONSTANTIN ENEA, PROF	Ecole Polytechnique	Directeur de thèse
MADAYAM SRIVAS, PROF	Chennai Mathematical Institute	Directeur de thèse

To my beloved grandmother - Vijayalakshmi

Abstract

The growth of modern internet is enabled by large-scale, highly available, and resilient distributed systems. These systems allow data to be replicated globally while ensuring availability under failures. To ensure reliability and availability in the presence of failures, the systems rely on intricate distributed protocols. Yet in practice, bugs in the implementations of these distributed protocols have been the source of many downtimes in popular distributed databases. Ensuring the correctness of the implementations remains a significant challenge due to the large state space.

Over the years, many techniques have been developed to ensure the correctness of the implementations ranging from systematic model checking to pure random exploration. However, a developer testing the implementation with current techniques has no control over the exploration. In effect, the techniques are agnostic to the developer's knowledge of the implementation. Furthermore, very few approaches utilize the formal models of the protocols when testing the implementations. Efforts put into modeling and verifying the correctness of the model are not leveraged to ensure the correctness of the implementation

To address these drawbacks, in this thesis, we propose 3 new approaches to test distributed protocol implementations - Netrix, WaypointRL, and ModelFuzz. The first two techniques - Netrix and WaypointRL are biased exploration approaches that accept developer input. Netrix is a novel unit testing algorithm that allows developers to bias the exploration of an existing testing algorithm. A developer writes low-level filters in a domain-specific language to fix specific events in an execution that are enforced by Netrix. WaypointRL improves on Netrix to accept high-level state predicates as waypoints and uses reinforcement learning to satisfy the predicates. WaypointRL is effective in biasing the exploration while requiring less effort from the developer. Using popular distributed protocol implementations, we show that additional developer input leads to effective biased exploration and improved bug-finding capabilities. The third technique - ModelFuzz - is a new fuzzing algorithm that bridges the gap between the model and the implementation of the protocol. We use model states as coverage to guide input generation that are then executed on the implementation. We show with three industrial benchmarks that existing coverage notions are insufficient for testing distributed systems and that using TLA+ model coverage to test implementations leads to discovering new bugs.

Keywords: Formal methods, Distributed Systems, Testing, Reinforcement Learning, Unit testing, Fuzzing.

Resumé

La croissance de l'internet moderne est rendue possible par des systèmes distribués à grande échelle, hautement disponibles et résilients. Ces systèmes permettent de répliquer les données à l'échelle mondiale tout en garantissant leur disponibilité en cas de défaillance. Pour garantir la fiabilité et la disponibilité en cas de défaillance, les systèmes s'appuient sur des protocoles distribués complexes. Pourtant, dans la pratique, des bogues dans la mise en œuvre de ces protocoles distribués ont été la source de nombreux temps d'arrêt dans les bases de données distribuées les plus populaires. Garantir la correction des implémentations reste un défi de taille en raison du vaste espace d'états.

Au fil des ans, de nombreuses techniques ont été mises au point pour garantir la correction des implémentations, allant de la vérification systématique des modèles à l'exploration aléatoire. Cependant, un développeur qui teste l'implémentation avec les techniques actuelles n'a aucun contrôle sur l'exploration. En effet, les techniques ne tiennent pas compte de la connaissance qu'a le développeur de l'implémentation. En outre, très peu d'approches utilisent les modèles formels des protocoles lors des tests de leurs implémentations. Les efforts consacrés à la modélisation et à la vérification de la correction du modèle ne sont pas mis à profit pour garantir la correction de l'implémentation.

Pour remédier à ces inconvénients, nous proposons dans cette thèse trois nouvelles approches pour tester les implémentations de protocoles distribués - Netrix, WaypointRL, et ModelFuzz. Les deux premières techniques - Netrix et WaypointRL - sont des approches d'exploration biaisées qui prennent en compte l'input du développeur. Netrix est un nouvel algorithme de test unitaire qui permet aux développeurs de biaiser l'exploration d'un algorithme de test existant. Un développeur écrit des filtres de bas niveau dans un langage spécifique au domaine pour fixer des événements spécifiques dans une exécution qui sont appliqués par Netrix. WaypointRL améliore Netrix en acceptant des prédicats d'état de haut niveau comme points de passage et utilise l'apprentissage par renforcement pour satisfaire les prédicats. WaypointRL est efficace pour orienter l'exploration tout en exigeant moins d'efforts de la part du développeur. En utilisant des implémentations de protocoles distribués populaires, nous montrons qu'une contribution supplémentaire du développeur conduit à une exploration biaisée efficace et à des capacités améliorées de recherche de bogues. La troisième technique - ModelFuzz - est un nouvel algorithme de fuzzing qui comble le fossé entre le modèle et la mise en œuvre du protocole. Nous utilisons les états du modèle comme couverture pour guider la génération d'entrées qui sont ensuite exécutées sur l'implémentation. Nous montrons, à l'aide de trois benchmark industriels, que les notions de couverture existantes sont insuffisantes pour tester les systèmes distribués et que l'utilisation de la couverture du modèle TLA+ pour tester les implémentations entraîne la decouverte de nouveaux bogues.

mots-clés: Méthodes formelles, systèmes distribués, tests, apprentissage par renforcement, tests unitaires, Fuzzing.

Acknowledgements

First and foremost, I would like to thank my advisors Constantin Enea and Mandayam Srivas for their guidance and support. I have grown in numerous ways under their mentorship. Beyond technical and research insights, they have helped me develop scientific rigor, improve my time management, technical writing, and numerous other skills necessary to complete this thesis. I will carry these lessons with me always.

I am also very grateful to Cezara Dragoi, who has been a mentor during the PhD and my internship at AWS. Cezara motivated me to embark on this scientific journey and has been a source of encouragement ever since. Cezara has been there to celebrate the highs and help me get out of my lows, always present whenever I needed her.

This thesis would not have been possible without the wonderful collaborators that I had a chance to work with. Thank you Andrea Borgarelli, Ege Berkay Gulcan, Burcu Ozkan, and Rupak Majumdar. Special thanks to Rupak for hosting and mentoring me at MPI, and to Burcu for her guidance and thought-provoking conversations that helped shape my scientific thought process.

I would like to thank the jury - Cezara Dragoi, Praveen M, Burcu Ozkan, Tayssir Touili. My sincere thanks to Gennaro Parlato and Stephan Merz for reviewing this thesis, recognizing the contributions, and giving me detailed feedback.

I would like to thank my mentors and friends at Chennai Mathematical Institute for introducing me to the world of computer science research and for making this joint PhD possible. Thank you Madhavan Mukund, S P Suresh, K Narayan Kumar, and Aishwarya Cyriac. To my friends at CMI - Vishwa and Asif, thank you for sticking with me, and for your support with CMI administrative tasks.

My stay in Paris for the duration of this PhD has been made memorable and easy thanks to the administrative staff at IRIF - Omur Avci, Jemuel Samtchar, and Marie-Laure Susairaj. I am indebted to my colleagues and mentors at IRIF who have helped me in various ways throughout this journey. Thank you Ahmed Bouajjani, Mikaël Rabbi, Filippo, Wael, Shamisa, Lucie, Roman, Weiqiang.

I've been fortunate to meet amazing people and make new friends who have always been there for me, to share a laugh, gossip, and have fun. Thank you Klara, Enrique Roman Calvo, Daniel, Mouna, Patricio and Solange for sharing all the wonderful meals, jokes, and conversations. Thank you Namratha, Viru, Charles, and Prateek for making my time at Cite memorable. I've been lucky to have a group of friends since my childhood. Thank you for always being there for me, celebrating my joys and supporting me in tough times.

I would like to thank my parents for all their sacrifices which allowed me to pursue this endeavor, and the rest of my family for being a constant source of support.

Finally, I thank my wife, Supraja, for all the love. She has been a friend, intellectual partner, mentor, and much more. Thank you for always bringing out the best in me.

Contents

Chapter 1

Introduction

Distributed systems are the backbone of the current Internet infrastructure. Large online services such as banking, e-commerce, and blockchain systems utilize distributed databases which are powered by a conjunction of distributed systems - such as failure detection, recovery and consensus systems. To ensure correct behavior, reliability, and availability under failures, distributed databases rely on functionally correct distributed protocol implementations. However, developing correct implementations remains a challenge due to the complex nature of the underlying distributed algorithms.

Consider an example of a concrete distributed system - a cloud file storage service like Dropbox or Amazon S3. Customers interact with the service by uploading and retrieving files. To enable this behavior, Dropbox needs to store, access, and modify exabytes of data in different nodes across the globe, all the while ensuring that data is not lost, corrupted, or incorrectly overwritten. To ensure these properties, Dropbox relies on a host of distributed protocols. Among them, a key protocol is consensus. A consensus protocol ensures that updates to the data (read and write operations) are ordered the same way across different nodes in the system. Therefore, a customer accessing data from different locations will always receive the same output. In the absence of a consensus protocol, different nodes storing data may end up in different states - such as one deleting a file and the other storing it. In Dropbox like services, the codebase that implements the consensus protocol forms the core of critical codebase. As a consequence, bugs in the implementation of a consensus protocol are expensive and lead to loss of core functionality such as losing customer files.

To ensure correct behavior while tolerating failures, distributed protocols rely on complex rules, use many message types, and synchronization mechanisms leading to a very large state space. Put together, the protocol descriptions (models) require complex proofs of correctness.

However, practical implementations of the distributed protocols are highly prone to bugs [Gao+18]. The primary reason is the complex nature of the protocols themselves. The complexity of protocols that enable services like Dropbox stems from 2 aspects - (1) asynchrony - nodes progress independent of each other due to the lack of a global time [She15], and (2) fault tolerance - nodes can crash and stop communicating. For example, let us consider a bug in the distributed key value store (RedisRaft [1]) that we encounter in one of our

[1] https://github.com/redislabs/redisraft

evaluations. A participating node crashes (fails) during an execution and restarts in a stale state. Upon restarting, the node communicates with other nodes to reconcile the stale state. Until the state is updated, the node remains unavailable to clients. However, due to a bug in handling conflicting state update messages that are received asynchronously from different nodes, the node crashes again. This process repeats making the node permanently unavailable.

Practical implementations also exacerbate the problem of state explosion (large state space). The state space increases to accommodate for numerous additional implementation details which are typically abstracted away in the models. For example, implementations need to keep track of active peers, the messages that have been sent but not acknowledged, timeouts, etc. These additional parameters lead to a much larger state space. Naturally, the implementations are prone to human programming errors where the nodes do not behave as expected. Coupled with the state explosion problem, efficiently finding bugs in an implementation becomes non-trivial. The solution is to develop techniques to traverse the state space more efficiently which results in improved reliability and correctness of the implementation.

Research landscape

Over the years, techniques have been developed to specify and check the correctness of the algorithms. Domain specific languages such as Promela, Dafny, TLA [Hol97; Jab+21; Jab+21; Lam02] allow protocol designers to describe and verify the models with systematic exploration of the model's state space. Similarly, systematic exploration techniques are applied to explore the state space of the implementation. Tools such as SAMC, MODIST, etc [BK08; Lee+14; SBG11; Yan+09; Wan+23] explore the space of all possible message interleavings to certify the absence of bugs. However, systematic exploration techniques applied on the model are disconnected from the efforts on the implementation.

Other tools such as Ivy, Disel, Grove [Pad+16; SWT18; Sha+23] rely on proof techniques to prove the correctness of distributed models. Motivated by the success of these proof techniques on the models, tools such as IronFleet, Verdi, etc [Cha+21; Haw+15; Wil+15] have been developed to formally verify implementations. However, the verification efforts do not scale and the verified implementations take a performance hit.

Both classes of techniques - model-checking and proof-based - are characterized by strong guarantees of finding all bugs or proving their absence. However, despite many advances, the techniques scale poorly. Exploring all possible message inter-leavings and fault scenarios to cover the large state space remains impractical for most distributed protocol implementations. Similarly, verified implementations (using proof-based techniques) fail to satisfy the performance requirements for running in production and therefore, largely remain unused. Moreover, bugs can still lurk in the interfaces between verified and unverified components [Fon+17]. These drawbacks have been partially addressed by new techniques such as symbolic exploration [PDG23] and model-based testing approaches [Wan+23].

Alternatively, randomized testing techniques have been developed to address the drawbacks of systematic exploration techniques. These techniques test implementations with randomly generated inputs. However, randomized techniques provide weak or no guarantees of finding bugs or proving their absence. More widely-used randomized techniques rely on pure randomness to explore the state space. Jepsen [Kin22] being the canonical

example of such an approach. Jepsen randomly injects network partitions and heals them throughout the execution. However, Jepsen is unreliable in reproducing any bugs found.

In general, randomized testing tools have been surprisingly effective in finding bugs in production systems [MN18]. Other techniques control the randomness to provide probabilistic guarantees of visiting a state [Kul+18; CMN16], leverage partial order reduction techniques to efficiently sample random traces [KMO19; Dra+20] or use standard fuzzing loops to mutate inputs and increase coverage [Che+20; Gao+23; MOP23; Win+23].

Operationally, both techniques - systematic exploration and randomized testing - require running the distributed system for many iterations. Each iteration results in an execution characterized by a sequence of partially ordered events occurring at each node. While with randomized testing, we effectively sample a random execution in each iteration, with systematic exploration, the execution is carefully constructed to reach a new state. Due to the large state space, covering it entirely in a reasonable number of iterations remains infeasible. Furthermore, all current techniques are fully automatic i.e. once the initial parameters are set, the developer has no control over the exploration of the state space.

Testing using biased exploration

An alternative testing approach, that we pursue in this thesis, would be to bias the exploration. There are two concrete motivations for biased exploration. First, we know from an existing corpus [Gao+18] of bugs that the root cause of bugs is a small set of problematic behaviors. Biased exploration allows us to explore executions where these problematic behaviors always occur. Second, the developer of a specific implementation knows which executions utilize untested parts of the codebase. Additionally, biased exploration will either find bugs in the untested code or increase the confidence when no bugs are found.

To understand the impact of biased exploration, let us consider a concrete example. Existing corpus of bugs indicates that bugs are more likely when nodes crash during an execution. Therefore we first bias exploration to include crashed nodes. To isolate the exploration further, let us suppose that the developer has not tested for scenarios when at least 2 nodes have crashed. Once the implementation has been tested with an effective biased exploration technique, two outcomes are possible. First, the exploration does find a bug - improving the robustness of the implementation, or second, no bugs are found - the developer has increased confidence in the correctness of the module handling node crashes.

With existing tools, bugs found in implementations are hard to reproduce (bug reproducibility problem). To enable reproducing bugs, all sources of non-determinism (e.g. timeouts, random choices, etc..) in the implementation needs to be controlled. However, certain sources of non-determinism are hard to control. For example, time in each process is a source of non-determinism that is hard to control. Processes maintain independent clocks that need to be synchronized in order to reproduce any bug.

Biased exploration automatically addresses the problem of bug reproducibility. When the exploration is targeted, the size of the state space explored is reduced and therefore bugs are more likely to be reproduced. Consider the earlier example where we only explore traces where 2 nodes have crashed, any bugs found are more likely to be reproduced with biased exploration (due to the smaller state space) when compared to an unbiased automatic exploration technique that is less likely to introduce the same two crashes.

Finally, there is a scarcity of techniques that bridge the gap between a model of the protocol and the implementation. Extensive research has been dedicated to modeling distributed protocols and to verifying the correctness of the models. Very few testing techniques leverage these models to ensure the correctness of the implementation. By bridging the gap, developers can check the correctness of their implementation by ensuring conformance with the model.

To summarize, ensuring the correctness of distributed protocol implementations is critical. However, due to the large state space, systematic exploration techniques do not scale in practice. Therefore, randomized testing have been more successful in exploring the state space to find bugs. Yet, current techniques only allow for fully automatic exploration without any control over the exploration. Our solution - biased and targeted exploration - gives the developer control while addressing the bug-reproducibility problem. Additionally, we identify that there is a lack of techniques to bridge the gap between models and implementation.

Contributions

In this thesis, we introduce three new randomized techniques to test the implementation of a distributed algorithm - Netrix, WaypointRL and ModelFuzz - with the goal of addressing the drawbacks of existing techniques mentioned in the previous section. We show that the techniques are useful in finding new bugs and replicating known bugs in many popular distributed protocol implementations that are used in production. The first two, Netrix and WaypointRL, are biased exploration techniques. They allow the developer to test an implementation in a constrained state space described by specific scenarios. Then, with ModelFuzz, we directly leverage protocol models to test implementations. ModelFuzz is a randomized testing approach where the input generation is guided by coverage over abstract protocol models.

All three techniques rely on an instrumentation to enforce specific executions on the implementation. The instrumentation is similar to other existing testing techniques and consists of two components - (1) control over the network to decide which messages to deliver and (2) control over the nodes to inject failures and restarts. In what follows, we provide a brief overview of the three techniques.

Netrix - Unit tests for Distributed Systems

Netrix [2] is a new testing framework along with a Domain Specific Language (DSL) that allows a developer to bias the exploration of an existing exploration algorithm. The developer, using the DSL, describes constraints that are then enforced by Netrix on the underlying exploration algorithm. Concretely, the developer describes a sequence of filters which fixes the relative order of certain events in every execution. For the remaining events, their order is decided by the underlying exploration algorithm. Given a sequence of filters, the degree of constraints on the execution explored is defined by the number of filters. More

[2] https://github.com/netrixframework/netrix

filters enforce more constraints on the exploration. Effectively, the developer operates on a spectrum. On the one end, the exploration is driven entirely by the underlying exploration algorithm when no filters are specified. On the other end, with a larger number of filters, the developer can constrain the exploration to a single specific execution.

NETRIX's filters define an abstract *monitor* that simulates the network used by the processes to communicate. We formalize the semantics of the filter and characterize the expressivity of NETRIX's filters.

Our concrete implementation of the NETRIX *monitor* consists of a runtime that captures all events of the distributed system and therefore, simulates the network. The events here correspond to message send, message receive or other internal events at each node. For each event, the filters decide the corresponding action to perform on the network. The filters are of the form **If(*condition*).Then(*action*)** where *condition* is a predicate over the state of the system and the current event and *action* corresponds to a network action. For example, consider the rule

```
If(IsMessageOfType(t)
     .And(IsMessageTo(p))
     .And(NumberOfMessagesDeliveredGreaterThan(n/2)))
.Then(DropMessage())
```

The NETRIX runtime, in this case, will enforce that in every execution the process p receives only $n/2$ messages of type t. The state of the *monitor*, in this case, keeps track of how many messages has been delivered so far in the execution. In general, the DSL allows developers to define arbitrary predicates as *conditions* to be used in filters. For a given event, if none of the filters apply, then the underlying exploration algorithm defines the corresponding network action.

In NETRIX, a set of filters is analogous to traditional unit tests - they enforce specific paths of the execution that can be checked for correctness. NETRIX's unit tests are the first regression testing framework for distributed systems. They ensure that the behavior of the protocol implementation remains unchanged across different versions.

To demonstrate the effectiveness of NETRIX's biasing capabilities, we evaluate 3 industrial benchmarks - Tendermint [3], Etcd [4] and BFT-Smart [5]. We show with 34 unit tests that NETRIX filters are effective in biasing exploration. Additionally, we find 4 new bugs and reproduce 3 known bugs with improved bug-reproducibility.

WAYPOINTRL - Biased exploration with Reinforcement Learning

Existing testing techniques sample executions in each iteration independent of prior samples. In other words, they are effectively "blind". Even with NETRIX, while the operational semantics allows a developer to enforce specific events in an execution, each execution is sampled by the underlying exploration algorithm independent of prior executions. The

[3] https://github.com/tendermint/tendermint
[4] https://github.com/etcd-io/raft
[5] https://github.com/bft-smart/library

next two techniques we present in this thesis - WaypointRL and ModelFuzz "learn" from prior iterations and choose executions conditioned on prior samples.

We model state-space exploration as a *reinforcement learning* (RL) problem and propose two new Q-learning based exploration algorithms - BonusMaxRL and WaypointRL[6]. The former, BonusMaxRL, is an unbiased and fully automatic exploration algorithm with a strong incentive to maximize coverage of the state space. We show that it significantly outperforms other random exploration techniques and matches the state-of-the-art learning based approach. The latter, WaypointRL is a biased exploration algorithm. WaypointRL allows developers to augment the exploration with *waypoints* (akin to Netrix filters) - that are then used to provide additional rewards.

An RL agent interacts with the participating nodes in both the algorithms, steering and guiding the exploration based on the rewards provided. At each step in an iteration, the agent observes the current state and decides a network action (or injects a failure) that leads to the next state. In BonusMaxRL, the reward is highest when a new state is observed and diminishes with the number of visits. However, in WaypointRL, this diminishing reward is augmented with an explicit positive reward when a *waypoint* is reached. For example, in a consensus protocol, a waypoint would be a successful leader election or a commit of a client request. Notice that *waypoints* operate at a higher abstraction and unlike Netrix's filters, are not concerned with individual message contents. As a consequence, we reduce the developer's effort to effectively bias the exploration.

To demonstrate the effectiveness of the approach, we evaluate both the algorithms on 3 industrial benchmarks - RedisRaft [7], Etcd [8] and RSL [9]. With 26 different waypoint sequences, we show that WaypointRL is effective in biasing exploration. Furthermore, we find 13 new bugs with WaypointRL and demonstrate that biasing leads to improved bug-reproducibility.

ModelFuzz - Model-Coverage based Fuzzing for Distributed Systems

To describe Netrix's *filters* or WaypointRL's *waypoints*, we provide recommendations that derive predicates from the model. In doing so, we partially address a drawback of current testing techniques - disconnectedness between the model and the implementation. With ModelFuzz, we bridge this gap more explicitly.

ModelFuzz is a novel fuzzing algorithm for testing distributed systems that leverages the formal models of the protocols. ModelFuzz uses the coverage information of the model to guide the exploration of the state space of the implementation. Similar to WaypointRL, ModelFuzz "learns" from prior samples to generate inputs that are more likely to lead to new states. We take as input, an implementation and its corresponding TLA+ model. In each iteration we execute an input first on the implementation, then run the subsequent execution on the model. When run on the model, we collect information of the states visited and use this information to generate new inputs.

[6] https://github.com/zeu5/dist-rl-testing
[7] https://github.com/redislabs/redisraft
[8] https://github.com/etcd-io/raft
[9] https://github.com/Azure/RSL

To implement ModelFuzz, we overcome the challenge of describing deterministic inputs to distributed systems, design ways to simulate models on single executions and develop a framework to instrument and run the whole distributed system.

To demonstrate the effectiveness of ModelFuzz, we instrument and run the algorithm on 2 industrial benchmarks - Etcd [8] and RedisRaft [7], and 1 micro benchmark. We show that using model coverage to guide exploration outperforms traditional notions of coverage - line, branch and trace coverage. We uncover 15 new bugs in the 2 industrial benchmarks and reproduce a synthetic bug.

Summary

Overall, in this thesis, we introduce new randomized techniques to test implementations of distributed protocols. The techniques address the drawbacks of existing state-of-the-art testing tools. Specifically,

- We introduce a unit-testing framework Netrix accompanied by a DSL with the aim of biasing exploration and therefore improving bug-reproducibility. Netrix unit tests reuse the developer's knowledge of the implementation.
- We develop two new learning-based exploration algorithms WaypointRL and BonusMaxRL. WaypointRL automates biased exploration. Additionally, the developer's effort is minimized to providing high-level state predicates as opposed to low level filters.
- We bridge the gap between the model and the implementation by introducing a new fuzzing approach ModelFuzz that leverages the state coverage information of the model to guide input generation to test the implementation.

Publications

The results of Netrix and WaypointRL have led to successful publications listed below.

1. *A Domain Specific Language for Testing Distributed Protocol Implementations*
   Cezara Dragoi, Srinidhi Nagendra, Mandayam Srivas
   NETYS 2024
2. *Reward Augmentation in Reinforcement Learning for Testing Distributed Systems*
   Andrea Borgarelli, Constantin Enea, Rupak Majumdar, Srinidhi Nagendra
   OOPSLA 2024

The results of ModelFuzz are under submission.

1. *Model-guided Fuzzing of Distributed Systems*
   Ege Berkay Gulcan, Burcu Kulahcioglu Ozkan, Rupak Majumdar, Srinidhi Nagendra
   Under submission

Tools

All three techniques have been implemented using the Go programming language and are open sourced.

1. Netrix - github.com/netrixframework/netrix
2. WaypointRL - github.com/zeu5/dist-rl-testing
3. ModelFuzz - github.com/zeu5/raft-fuzzing, github.com/zeu5/redisraft-fuzzing

Thesis organization

The rest of the thesis is organized as follows,

- Chapter 2 introduces a formal model of distributed systems. Subsequently, we elaborate on concrete protocols that we test.
- Chapter 3 describes the first technique Netrix where we introduce a new domain specific language to describe unit tests for distributed systems. The unit tests allow a developer to constrain the execution while utilizing existing randomized exploration algorithms to search in the constrained space.
- Chapter 4 first elaborates on modelling distributed systems as reinforcement learning agents before diving into the mechanism of biased exploration. With Netrix, we rely on the developer to define the boundaries of the constrained state space. We overcome this manual effort and show how shaping rewards allows us to *learn* to explore only the constrained state space.
- Chapter 5 introduces a new fuzzing mechanism ModelFuzz, to connect the protocol model specified in TLA+ to an implementation. By doing so, ModelFuzz allows us to utilize the modelling effort to test an implementation.
- Finally, Chapter 6 concludes with a summary of contributions and lists possible future directions of research.

Chapter 2

Modelling Distributed Systems

The specific problem that we are addressing in this thesis is - ensuring the correctness of an implementation with respect to a model of a distributed system. Before presenting our solutions, in this chapter, we describe the abstract model of a distributed system. The model presented does not make any assumptions about the concrete distributed systems, its state, and transition rules. We then elaborate on one particular concrete distributed system - consensus systems - using two examples, Raft and PBFT.

2.1 Abstract Distributed systems

In this thesis we are concerned with testing asynchronous message passing distributed systems. Asynchronous, meaning time at each node progresses independently of other nodes. Message passing, meaning nodes share information by sending and receiving messages. In this subsection, we will describe a formal model to capture executions of asynchronous message passing distributed systems. We will use the terms nodes and processes interchangeably.

In a distributed system, a set of n processes $\mathcal{P}$ communicate by exchanging messages. We define a message as a tuple $m \in (\mathcal{P} \times \mathcal{P} \times \mathbb{V}_M \times \mathbb{T}_M)$ where $\mathbb{V}_M$ is the set of all possible contents of a message and $\mathbb{T}_M$ is the set of all possible message types. $\mathcal{M}$ is the set of all possible messages. A given subset of messages $M \subseteq \mathcal{M}$, is a union of messages intended for a process p - $M[p] = (\mathcal{P} \times \{p\} \times \mathbb{V}_M \times \mathbb{T}_M)$.

The execution of a distributed protocol at a particular process is characterized by a sequence of events such as message send, message receive or other internal steps. We define an event as a tuple $(\mathcal{P} \times \mathcal{T}_E \times \mathcal{V}_E)$, where $\mathcal{T}_E = \{send, receive, internal\}$. $\mathcal{V}_E$ is the set of possible event values and contains $\mathcal{M} \subseteq \mathcal{V}_E$. For message send and receive events, the event value is the message. $\mathcal{E}$ is the set of all possible events.

2.1.1 Protocol transition system

Given the preliminary definitions, we now define a *local state* of each process along with a *local transition system*. We define a protocol π that captures the local state of each process as follows,

Definition 1. *A **protocol** π is defined by the tuple $(\Sigma_\pi, s^0_\pi, \delta_\pi, F_\pi)$ where*

- *Σ_π is the set of process local states as defined by the protocol and $s^0_\pi \in \Sigma_\pi$ is the initial state.*
- *$\delta_\pi : \Sigma_\pi \times (\mathcal{M} \cup \{\perp\}) \rightharpoonup \Sigma_\pi \times \mathcal{E}$ is the (partial) transition function of the process. Each transition emits an event. Internal steps are represented with transitions $\delta_\pi(s, \perp)$ while receiving a message m is represented with a transition $\delta_\pi(s, m)$. We assume that internal steps and message receive steps cannot be enabled in the same state, i.e. from any state $s \in \Sigma_\pi$ if $\delta_\pi(s, \perp)$ is defined, then $\delta_\pi(s, m)$ for any $m \in \mathcal{M}$ is not defined.*
- *$F_\pi \subseteq \Sigma_\pi$. For any state $s \in F_\pi$, δ_π is not defined from s. Final states allows us to restrict the length of an execution in each replica.*

The *global state* of the system consists of two parts. The set of local states of each process and the network. We define the global state of the distributed system (a configuration) as follows,

Definition 2. *A **distributed system configuration** $C = (E, pool, states, messages)$ where,*

- *$E = (e_0, e_1, \cdots)$ is a sequence of events $e_i \in \mathcal{E}$. This serves as a history of events occurred at each process.*
- *$pool \subseteq \mathcal{M}$ is the set of messages in transit between different process and signifies the network.*
- *$states$ maps each process p to a state in Σ_π*
- *$messages$ maps each process p to a sequence of messages $(m_0, m_1, \cdots)$ with $m_i \in \mathcal{M}[p]$. The sequence represents the inbound queue of process p*

Figure 2.1 refers to the transition rules of the global state of the distributed system (between two configurations). The rule INTERNAL allows a process to transition its local state and emit *internal* events. In the SEND rule, a process emits specifically a *send* event for a message m. The message m is added to the network $pool$. The NETWORK rule adds messages from the pool to the inbound message queue of a process $messages[p]$. Rule RECEIVE allows processes to consume a message from their inbound queues by emitting specifically a *receive* event. Additionally, the ADVERSARY rule models adversarial (Byzantine) behavior where the message pool is transformed arbitrarily.

Note that the protocol π is abstract. We do not make any assumptions about its states or the transition relation. The transition relation can be non-deterministic. For example, in a concrete implementation of the protocol, the state contains time of the process, and INTERNAL transitions increment the time.

We define an execution ρ as a sequence of transitions between configurations. The initial configuration is $C_0 = ((), \phi, states_0, messages_0)$ where $\forall r \in \mathcal{P}$, $states_0[r] = s^0_\pi$ and $messages_0[r] = ()$. An execution ρ is *complete* if all replicas have reached final states. $(\forall r, states_k[r] \in F_\pi)$.

The *history* of an execution ρ is the tuple $H_\rho = (E_\rho, <_\rho)$ where E_ρ is the set of events in ρ ordered by the standard (partial) happens-before order $<_\rho$. We will use $e \in E_\rho$ and $e \in H_\rho$ interchangeably to denote an event exists in a history. Similarly, $M_\rho =$

$$\frac{\delta_\pi(states[r], \bot) = (s', e) \qquad e = (r, internal, v)}{(E, pool, states, messages) \xrightarrow{e} (E.e, pool, states[r \to s'], messages)} \textsc{Internal}$$

$$\frac{\delta_\pi(states[r], \bot) = (s', e) \qquad e = (r, send, m)}{(E, pool, states, messages) \xrightarrow{e} (E.e, pool \cup \{m\}, states[r \to s'], messages)} \textsc{Send}$$

$$\frac{messages[r] = \sigma \qquad m \in pool[r]}{(E, pool, states, messages) \xrightarrow{network} (E, pool \setminus \{m\}, states, messages[r \to \sigma.m])} \textsc{Network}$$

$$\frac{messages[r] = m.\sigma \qquad \delta_\pi(states[r], m) = (s', e) \qquad e = (r, receive, m)}{(E, pool, states, messages) \xrightarrow{e} (E.e, pool, states[r \to s'], messages[r \to \sigma])} \textsc{Receive}$$

$$\frac{M \subseteq \mathcal{M}}{(E, pool, states, messages) \xrightarrow{adversary} (E, M, states, messages)} \textsc{Adversary}$$

Figure 2.1: Transition rules of a distributed system.

For a function $f : A \to B$, we use $f[a \to b]$ to denote a function $f' : A \to B$ where $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$.

$\{m \mid (m.to, receive, m) \in E_\rho\}$ is the set of messages delivered to processes in the execution. We define the happens-before relation as the smallest relation between any two events $e_1, e_2 \in E_\rho$ written ($e_1 <_\rho e_2$) that satisfies,

1. e_1, e_2 are emitted by the same replica, and e_1 occurred before e_2 in ρ
2. e_1 is a send event and e_2 is the matching receive event, i.e., $e_1 = (m.from, send, m)$ and $e_2 = (m.to, receive, m)$
3. (transitive closure) there exists e_3 such that $e_1 <_\rho e_3$ and $e_3 <_\rho e_2$

Different executions can result in the same history. The difference between the different executions will be the ordering of concurrent and unrelated events.

Later in Chapter 3, we introduce a generic *monitor* that simulates the network, replacing the Network and Adversary rules. Additionally, the monitor interleaves with the distributed system transition system defined above. By simulating the network, the *monitor* controls the set of possible executions. Therefore, a *monitor* can bias the exploration.

2.2 Consensus protocols

The description above captures behavior of any abstract distributed protocol. While the techniques we develop in this thesis are applicable to any such generic distributed protocol, we restrict our evaluation to implementations of the consensus protocol. The restriction allows us to convincingly demonstrate the effectiveness of our techniques for concrete

implementations that are widely used and deployed.

Consensus protocols define the behavior of nodes with the goal of solving a specific problem - where all correct nodes need to agree on a common value. Initially, all nodes propose a single value and in the resulting final state, all nodes should decide on a common value. When no values are proposed, no value should be decided. The following are the safety properties that should be satisfied by a consensus protocol:

- **(Validity)** If all nodes propose the same value v then the decided value should be v.
- **(Agreement)** The number of decided values among all nodes should be exactly one.
- **(Termination)** All non-faulty nodes should eventually decide.

The network conditions play a crucial role in deciding the solvability of the consensus problem. It is impossible to solve under asynchronous network conditions [FLP85]. However, many solutions exist under partial synchronous conditions [DLS84]. We will discuss two solutions in the next section.

Based on the type of failures the solution tolerates, there are limits to the number of failures. Under crash failures - where nodes abruptly stop - the number of failures is limited to $f < \frac{n}{2}$ [DLS84] where n is the number of nodes and f the number of faulty nodes. Similarly, under Byzantine failures - where nodes behave arbitrarily - the maximum number of failures that can be tolerated is $f < \frac{n}{3}$ [LSP82]. Consequently, the size of the "quorum" depends on the type of failures tolerated. For crash failure, the quorum typically refers to at least $\frac{n}{2} + 1$ nodes and for byzantine failures, at least $\frac{2n}{3} + 1$ nodes.

Failures in a distributed system are consequential only if they are detected or observed through the communication. Specifically, a crash failure is detected only when messages are not received, and Byzantine failures affect the state of the system only when the messages of the violating process is received by other processes. Therefore, in the model of distributed systems we present in Section 2.1 we do not model failures explicitly. However, crash failures are represented by the adversary dropping all the messages of the crashed process thereby simulating a crash. Similarly, the adversary can change the contents of the messages to model Byzantine failures.

We will present one solution that tolerates crash failures and one solution that tolerates Byzantine failures. Accordingly, we will test implementations of these solutions.

The abstract formal model described above captures the executions of a distributed system. In this section, we will describe two concrete distributed protocols - Raft and PBFT. For both examples, we will describe the protocol π, the set of states at each node Σ_π, and the transition function δ_π.

2.2.1 Raft

Raft [OO14] is a distributed protocol that solves the consensus problem under crash failures. The executions of raft begin with a client submitting a request (*ClientRequest*) to a set of processes/nodes. The set of processes will execute the consensus protocol and subsequently respond to the client whether the request was committed or not.

Figure 2.2 illustrates an example execution of the Raft protocol. The protocol proceeds in a sequence of interleaved terms with two phases within each term - leader election phase and a leader replication phase.

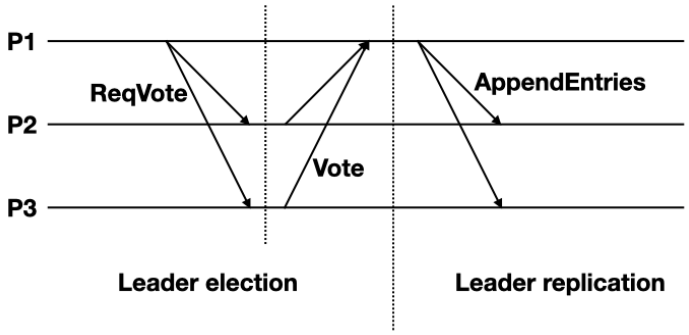


Figure 2.2: Example Raft execution

Initially all processes are *followers* in term 0 and aim to elect a leader. After a randomly chosen timeout duration, a process transitions to *candidate* state and sends a message requesting vote to all other processes. If a process receives a quorum of votes, it becomes a *leader*, concluding the leader-election phase. Subsequently, the leader replicates the requests for that term to other processes and receives acknowledgments. When a quorum of processes acknowledge a replicated request within the term, the leader commits the request and relays the commit information to all processes. Additionally, the leader sends periodic heartbeat messages to all processes. If a process does not hear from the leader within preconfigured duration, it increments the term number and transitions to the *candidate* phase thereby initiating the next leader election phase.

Raft is a concrete instantiation of π from Section 2.1. The local state of a raft process $s \in \Sigma_\pi$ consists of the role (leader, follower or candidate), the term number, the current leader, the log of requests and so on. The initial state s^0_π is term 0 and with the follower role. The informal description of interaction between processes defines the protocol transition function δ_π.

For our testing purpose, we define the final state as those where the process has committed all *ClientRequests*. Since Raft only tolerates crash failures, we accept transitions of the Adversary (from Figure 2.1) rule where the contents of the messages remains unchanged.

In our evaluation of NETRIX, WAYPOINTRL and MODELFUZZ, we test 2 industrial implementations of the Raft protocol - Etcd Raft and RedisRaft. Etcd Raft [1] powers a popular key value store used most commonly in cloud services. Etcd is also one of the core components of the Kubernetes [2] cloud platform and powers many important services such as DNS and service discovery. RedisRaft [3] is an extension of another popular distributed key-value store - Redis.

[1] https://github.com/etcd-io/raft
[2] https://kubernetes.io
[3] https://github.com/redislabs/redisraft

2.2.2 PBFT

Practical Byzantine fault tolerance (PBFT) [CL99] is a consensus protocol that tolerates Byzantine failures. In addition to crash and network failures, Byzantine failures includes arbitrary behaviors of a process such as lying about votes. Byzantine fault-tolerant protocols have seen a rise in popularity due to their applicability in Blockchain systems.

Proof-of-stake blockchain systems rely on a set of participating nodes to agree on the transactions to execute in each block. Byzantine consensus protocols are a natural fit for the problem. Tendermint [BKM18] is one such blockchain protocol which is inspired by PBFT. In our evaluation of Netrix, we test the implementation of Tendermint. The rest of this section describes the PBFT protocol which we use as a motivating example with Netrix.

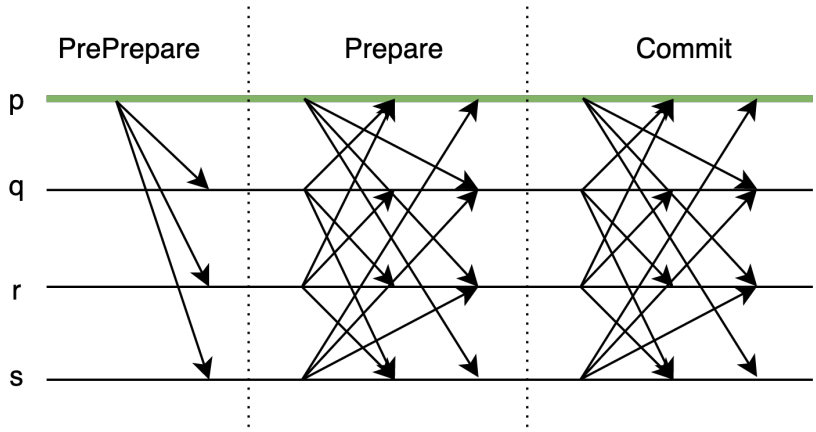


Figure 2.3: One execution round of PBFT with 4 processes

The normal execution of PBFT is denoted in Figure 2.3. The execution begins with a client submitting a request to a designated proposer. The proposer replies to the client once the request is committed.

Execution proceeds in rounds where in each round the designated proposer proposes a request. Each round comprises of 3 phases - *PrePrepare*, *Prepare* and *Commit* that the processes keep track of. Consider the system has n processes and tolerates up-to f failures where $n \geq 3f + 1$. At the end of the three phases, all correct processes decide either to commit the request or not. In *PrePrepare* round, the proposer broadcasts the requests. Processes respond to the request in the *Prepare* phase by broadcasting their vote. If a quorum ($2f + 1$) votes for the request, then they transition to the *Commit* phase where they broadcast votes once again before deciding to commit or reject. Processes keep track of a timer for each request and initiate a round change when no decision is achieved. A process transitions to a new round only if $f + 1$ other processes initiate a round change.

Similar to Raft, the local state of a process in PBFT consists of - the current proposer, the current phase and round number, the log of committed requests, current time etc. Unlike with Raft however, the set of all messages now includes arbitrary message corruptions to model the Byzantine failures. Therefore, a network adversary (represented by the Adversary rule in Figure 2.1) can change the contents of the messages to allow for Byzantine failures.

In our evaluation of Netrix, we test industrial implementations of two Byzantine protocols motivated by PBFT - Tendermint[4] and BFT-Smart[5].

[4]https://github.com/tendermint/tendermint
[5]https://github.com/bft-smart/library

Chapter 3

Netrix: unit tests for distributed systems

In this chapter we present the tool Netrix and a domain specific language to test distributed protocol implementations. Netrix allows developers to balance tradeoffs between effort and precision when testing. On one end, we have fully automatic randomized exploration techniques that require minimal effort from the developer. However, they are unreliable in reproducing bugs due to the large state space. On the other end, a skilled developer with high effort can test implementations by crafting intricate executions where bugs are more likely. Since executions are crafted, bug reproducibility is high.

With Netrix a developer can balance the tradeoffs by crafting scenarios while relying on the exploration capabilities of existing techniques. The scenarios, crafted using the domain specific language (DSL) that accompanies Netrix, are informally a set of constraints on the execution combined with an expected outcome. Concretely, a scenario is defined by a sequence of filters where each rule is of the form 'if-then' - similar to match action filters in a network switch. Similar to network switches, the filters define which messages are enabled. Only the enabled messages are fed to the underlying exploration algorithm. A developer either relies entirely on the exploration algorithm by not coding any filters or retains complete control on the exploration by coding elaborate filters.

The filters are expressive enough to allow for crafting intricate scenarios. A developer can delay (reorder) message delivery, block the delivery or changes the contents of the message before delivery. Effectively, the filters can introduce network partitions, simulate crash and byzantine failures along with exploring different message orderings. The filters along with a property state machine make up a unit test. The state machine encodes a safety property that is checked on every execution. The DSL, embedded in go, provides specific primitives to construct the 'if' and 'then' parts of the filter.

Figure 3.1 illustrates the general architecture of Netrix which involves (1) capturing the messages in transit, (2) recording the events occurring in the distributed system, (3) using the events to drive unit tests and (4) decide which messages to deliver based on the unit test actions. To enable (1) and (2) functions, the developer should instrument the implementation's network interface such that the processes in the distributed systems send and receive messages from Netrix.

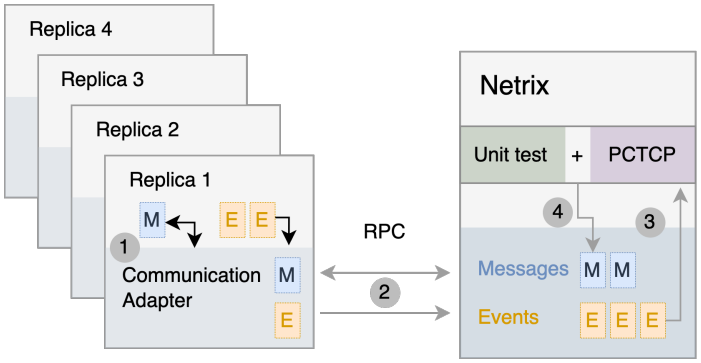


Figure 3.1: **Netrix architecture**
(1) The processes submit events, send/receive messages to a communication adapter. (2) The adapter talks to Netrix via RPC. (3) The events and messages drive unit tests. (4) Unit tests decide which messages to deliver

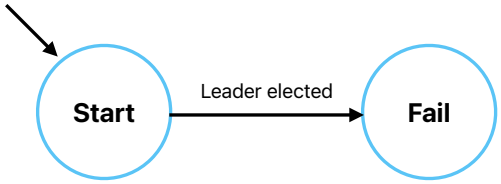


Figure 3.2: No leader election property in Raft.

Example 1. *Drop all votes*

```
If(IsMessageType(Vote))
    .Then(DropMessage())
```

Let us consider an example from Raft protocol to motivate and illustrate the working of a unit test. To recall the protocol, processes in Raft proceed in terms with two phases in each term - a leader election and a leader replication phase. Suppose the developer is interested to test the following scenario - "when no votes are delivered, no leader is elected". Figure 3.2 illustrates the state machine that encodes the property and Example 1 lists the filter that correspond to the scenario where we drop all vote messages. The above scenario tests for a safety property where nothing bad happens - no leader is elected. By enforcing the scenario, the developer is interested in testing the correctness of the relevant parts of the code - which in this case are those components that handle leader election messages to trigger a successful leader election.

As shown in the example, the filters are of the generic form **If(condition).Then(action)**. Netrix explores an execution by taking steps when new events are observed. At each step, Netrix first applies the filters sequentially - similar to a switch case. If a condition matches, the corresponding action is executed. In the default case (when no condition matches), Netrix feeds the messages (enclosed in the events) to the underlying exploration algorithm and finally delivers the messages output by the underlying algorithm. In parallel to the filters, Netrix feeds the events to the state machine encoding the safety property.

Given the informal semantics, the single filter from the earlier example ensures that the underlying exploration algorithm never observes any vote messages (since they are always dropped). Therefore, the exploration is biased to only include executions where no vote

Example 2. *Drop votes of a particular process p*

```
If(IsMessageType(Vote)
    .And(IsMessageFrom(p))
).Then(DropMessage())
```

Example 3. *Duplicate votes of a particular process p*

```
If(IsMessageType(Vote)
    .And(IsMessageFrom(p))
).Then(DuplicateVote())
```

messages are delivered. Current techniques fail to allow for such biased exploration. For example with pure random testing tool such as Jepsen, the likelyhood of dropping all vote messages or introducing sustained partitions to not elect a leader decreases exponentially with the number of vote messages.

In general, consider the *conditions* as predicates over the state of the system and *actions* as functions that decide the messages to deliver, the developer can craft high level scenarios to introduce different types of failures. Consider Examples 2 where we drop vote messages and 3 where we duplicate vote messages of a particular process. Example 2 describes a scenario with message drop failures and Example 3 allows a developer to duplicate messages. Similarly, we will show that Netrix filters are capable of introducing message re-orderings and byzantine failures.

To demonstrate the capabilities of Netrix unit-tests in constraining the search space and finding bugs, we write a total of 34 unit tests for 3 different benchmarks - Tendermint, BFTSmart and Etcd and show that (1) Netrix can constrain the search space by measuring the number of executions where the constraints are satisfied and, (2) with the constrained search, Netrix uncovers new bugs and reproduces known bugs.

Once we demonstrate that Netrix filters are effective in constraining the search space, the effort of the developer shifts to (1) describing scenarios and (2) encoding the scenarios into filters. To aid the developer, we develop two concrete methodologies. First, a methodology to derive scenarios from the specification and proofs of the protocol and second, show that the developer can leverage the strength of existing exploration techniques to write fewer filters. To demonstrate the latter, we introduce a new syntactic measure of filters called *filter distance* that measures the importance of a filter. We show that with a specific underlying exploration algorithm (PCTCP), the developer can forego writing filters with shorter distances and yet efficiently explore the constrained search space.

In what follows, we first provide a background of existing testing methods that are relevant. Then, in Section 3.2 we present the formal model of a *monitor* that captures the behavior of a Netrix unit test. Section 3.3 describes the core DSL syntax, semantics, and possible ways to extend the DSL. Section 3.4 contains details of the implementation. Subsequently, we evaluate Netrix unit tests in Section 3.5. Finally in Section 3.6, we will describe general guidelines to motivate scenarios and introduce the filter distance measure before discussing the related work of unit testing for distributed systems.

3.1 Background

Netrix allows developers to test an implementation with a given underlying exploration algorithm. In this section, we describe in detail two related exploration algorithms - Jepsen and PCTCP that we refer to in this chapter.

3.1.1 Jepsen

Jepsen [Kin22] is a popular randomized testing tool that has been successful in finding bugs in popular distributed systems. The tool runs as a control process that starts and stops all the nodes of a distributed system. Additionally, Jepsen controls the network between the processes using `iptables` rules to enable and disable certain networks. Then, Jepsen runs tests for many iterations. In each iteration, Jepsen generates random faults (network partitions between the nodes) to be injected during the execution and subsequently heals the partition. The resulting executions of running a Jepsen test is checked for safety properties such as linearizability.

While the faults generated are configurable to a certain degree, Jepsen tests in general do not allow a developer to bias exploration towards specific executions. Furthermore, bugs found by Jepsen are hard to reproduce. A typical workflow of testing with Jepsen is as follows. For every new version of the implementation, Jepsen tests are run for many hours on a cluster of machines. When bugs are found, the tests need to be rerun for the same duration with no guarantees of reproducing the bugs.

3.1.2 PCTCP

PCTCP [Kul+18] extends a popular concurrency testing algorithm PCT [Bur+10] to distributed systems. Both algorithms rely on a probabilistic guarantee of visiting a state in a given test run. Similar to Jepsen, the tools run for many iterations, exploring a specific execution in each iteration. The execution unfolds in a sequence of steps where the algorithm accepts a set of new messages (events) and returns a specific message to deliver.

As mentioned in Section 2.1.1, the events of different nodes in the execution of a distributed system are related by a happens-before partial-order relation. In PCTCP, a sequence of related events are chained together in a data structure called chain-partition (CP) with priorities assigned to chains (Illustrated in Figure 3.3). At each step of the execution, the next pending event from the highest priority (lowest value) chain is scheduled. In the figure, there are two chains with priorities 10 and 3 respectively. Therefore, a pending event from chain 2 is scheduled before a pending event from chain 1.

Furthermore, at random points in the execution, the priorities of two chains are swapped. Figure 3.4 shows the transition of the state of PCTCP chains. Chain 1 and 2 swap priorities with chain 1 receiving the higher priority value of 3. Therefore an event from chain 1 is scheduled and the execution continues to other chains when all events from the current chain have been exhausted.

The probabilistic guarantee is parameterized by a depth-bound d. Essentially, assume that a bug occurs when d different messages have a specific order in the execution. With PCTCP,

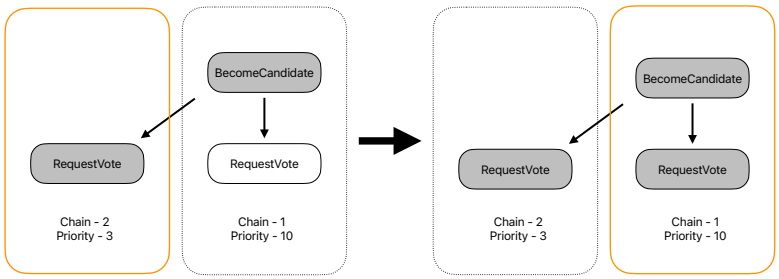


Figure 3.3: Snapshot of an ongoing execution with PCTCP. A step is taken choosing the event from the highest priority chain. Pictorially, executed events are colored in grey.

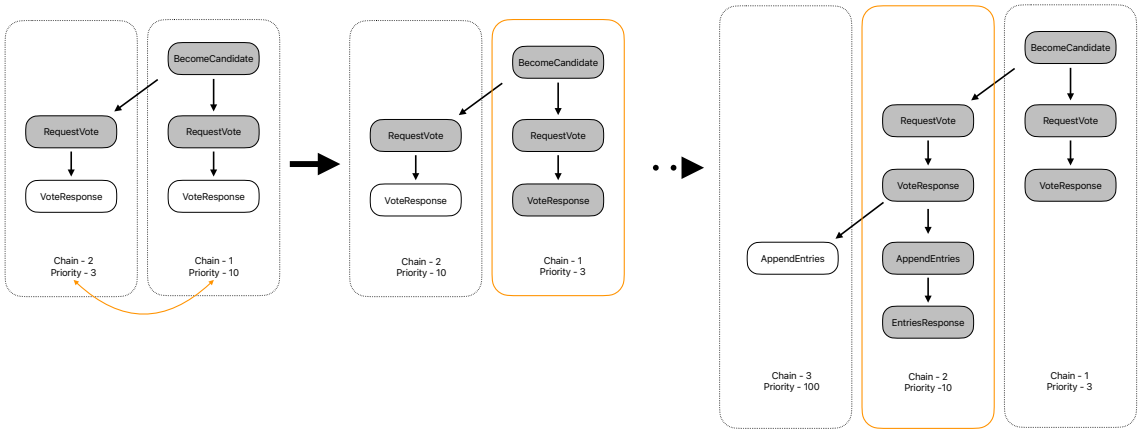


Figure 3.4: A priority change point of PCTCP and subsequent execution steps. Pictorially, executed events are colored in grey.

the probability of exploring an execution with d message ordering is $\frac{1}{w^2 h^{d-1}}$ where h is the length of the execution and w is the maximum width of the partial order of events. Since the expression is inverse exponential in d, the probability of finding bugs diminishes very quickly if the depth d increases.

PCTCP partially addresses the bug-reproducibility problem. Given a fixed random seed, the order of messages delivered are pre-determined therefore if a bug was found during a run of PCTCP, it is more likely to be found again. However, since PCTCP does not control other sources of non-determinism (such as time), in practice it is less likely to reproduce bugs. Furthermore, just as with Jepsen, the set of iterations need to be rerun from scratch to reproduce a bug.

When evaluating NETRIX with the benchmarks, we rely on PCTCP as the underlying exploration algorithm. Furthermore, when the developer is using PCTCP to test implementations, we develop a syntactic measure *filter distance* to help decide which filters to include in the test. The *filter distance* measure we will later present in Section 3.7 is specific to PCTCP.

3.2 Monitor

Earlier in Section 2.1, we describe a transition system (Figure 2.1) to capture executions of a distributed system. To test an implementation and force specific executions, we need control over the transitions of the global state. On one hand, to gain control over all 5 transition rules we require a heavy instrumentation that involves extensive effort and understanding of the codebase. On the other hand, to gain control of just the network and adversary we need a light instrumentation where we only capture the messages in transit. In this thesis, we test implementations only with a light instrumentation. In this section, we formally define a *monitor* which captures the semantics of the light instrumentation. A NETRIX unit test defines a *monitor* that forces executions as per the filters defined in the unit test.

Intuitively, the monitor, driven by the events occurring in the system, delivers messages to the message map of each process. The monitor contains state to encapsulate the logic while moving messages and based on this state can alter the order in which messages are delivered. For example, the monitor can drop messages, duplicate messages, corrupt messages or introduce new messages (to simulate byzantine-failures). Concretely, we define the monitor as follows,

Definition 3. *A **monitor** μ is defined as the tuple $(\Sigma_\mu, s^0_\mu, \delta_\mu, F_\mu)$ where,*

- *Σ_μ is the set of possible monitor states with s^0_μ as the initial state*
- *$\delta_\mu : \Sigma_\mu \times \mathcal{E} \rightharpoonup \Sigma_\mu \times 2^{\mathcal{M}}$ is the transition function which accepts the current state and event, and transitions to a new state along with a set of messages to be delivered.*
- *$F^M \subseteq \Sigma_\mu$ is the set of accepting states (used to signal some property being satisfied)*

Figure 3.5 describes the MONITOR transition rule. The rule invokes the transition function δ_μ with the head of the event queue E and the current monitor state as input and returns the new state, a set of messages to deliver. The delivered messages are added to the respective node's inbound message queues and the monitor state is updated.

$$\frac{\begin{array}{c} E = e.E' \\ \delta_\mu(s, e) = (s', M) \\ \forall r.\ messages'(r) = messages(r).M[r] \end{array}}{(E, pool, messages, s) \xrightarrow{monitor} (E', pool \setminus M, messages', s')}$$

Figure 3.5: Monitor transition rule

Note that in the transition rule, we reuse the terms E, $pool$, and $messages$ from the transition system of a distributed system. To recall, E signifies the sequence of events occurred in the distributed system, $pool$ is the set of in-transit messages, and $messages$ is a map of inbound message queues per process.

In what follows, we describe a product transition that combines the monitor with the distributed system. The product transition system, synchronizes the monitor and the distributed system on these shared components.

3.2.1 Product Transition System

The asynchronous product of the two transition systems, that of the system parameterized by the protocol π and the monitor μ defines the set of executions explored with a monitor. The distributed system takes steps which publish events that are consumed by steps of the monitor. In turn, the monitor decides the order and contents of the messages that are consumed by the processes. We define the configuration of the product transition system as follows,

Definition 4. *A **configuration of the product transition system** C is defined as the tuple $(E, pool, states, messages, s)$ where,*

- *E, $pool$, $states$ and $messages$ are defined as in the distributed system.*
- *s denotes the current state of the monitor.*

Figure 3.6 defines the transition rules for the product transition system. Steps are either Internal, Send, Receive from the distributed system (transitions labeled by events $e \in \mathcal{E}$) or a step in the monitor transition system Monitor. Note that a monitor step delivers messages to the message queues ($messages$ component of the configuration) of the process. Therefore, a monitor step simulates a sequence of Network and Adversary steps from the distributed system.

A run of the product system $\rho = C_0 \xrightarrow{l_0} C_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{k-1}} C_k$ is a sequence of transitions as above. A run is accepting if the monitor's state in the last configuration is a final state, i.e., $C_k^M.s \in F^M$.

3.2.2 On the Expressivity of the Monitor

We give a characterization of the monitor's capability to restrict the distributed system behavior. We show that as an extreme case, the monitor can restrict the distributed system

$$\frac{(E, pool, states, messages) \xrightarrow{e \in \mathcal{E}} (E', pool', states', messages')}{(E, pool, states, messages, s) \longrightarrow (E', pool', states', messages', s)} \textsc{System}$$

$$\frac{(E, pool, messages, s) \xrightarrow{monitor} (E', pool', messages', s')}{(E, pool, states, messages, s) \longrightarrow (E', pool', states, messages', s')} \textsc{Monitor}$$

Figure 3.6: Transition rules of product system

to produce a *single* history for a complete execution, i.e., all the complete executions in the product transition system have the same history. We characterize the capabilities of the monitor in terms of histories because the monitor cannot control the order between concurrent events, which are incomparable w.r.t. happens-before. The order in which such events are pushed to the event queue by the nodes (with the System transition rule) is not under the control of the monitor. While in theory, the number of executions that have the same history is exponential, in practice however, the number is smaller. The reason being that two executions where concurrent events are reordered is indistinguishable from the perspective of the processes. Therefore proving equivalence using histories that captures the relation between related events is sufficient to demonstrate the expressiveness of the monitor.

We state our result as a relation between the histories produced in a product transition system and the history of a given complete distributed system execution ρ. Histories of possibly incomplete executions of the product transition system are not necessarily equal to the history of ρ but only a *prefix*. We define the standard prefix relation as follows,

Definition 5. *The **prefix relation** $\preceq$ between two histories H_1, H_2 where $H_1 = (E_1, <_1)$ and $H_2 = (E_2, <_2)$ is defined as usual, i.e., $H_1 \preceq H_2$ if (1) Downward closure: $E_1 \subseteq E_2$ and for every event $e \in E_1$ and $e' \in E_2$, $e' <_2 e \Rightarrow e' \in E_1 \wedge e' <_1 e$, and (2) Preserving happens before: For two events $e, e' \in E_1$, $e <_1 e' \Leftrightarrow e <_2 e'$.*

Given the prefix relation, we now prove the main theorem capturing the expressiveness of the monitor.

Theorem 1. *For any complete run ρ in the distributed system of protocol π, there exists a monitor μ such that, for all executions ρ' in the product transition system of π and μ, $H_{\rho'} \preceq H_\rho$*

Proof. For the given execution ρ and history $H = H_\rho = (E_\rho, <_\rho)$, we define $O_H : M_\rho \to 2^{E_\rho}$, a function that maps a message m to events e' that *happen before* the receive event $e = (m.to, receive, m)$. (M_ρ is the set of all messages received by processes in the execution ρ)

$$O_H(m) = \{e \mid e <_\rho (m.to, receive, m)\}$$

To capture messages introduced using the Adversary rule, we define Adv_H

$$Adv_H = \{m \mid (m.to, receive, m) \in E_\rho \wedge (m.from, send, m) \notin E_\rho\}$$

We now define the monitor μ. The states of the monitor are $s \in \Sigma_\mu = (pool, E)$ where $pool \subseteq \mathcal{M}$ is the set of pending messages and $E \subseteq \mathcal{E}$ is the set of events that have already occurred. The initial state is $s^0_\mu = (\phi, \phi)$. We define the transition function of the monitor as $\delta_\mu(s, e) = (s', M)$ where

$$
\begin{aligned}
s'.E &= s.E \cup \{e\} \\
M &= \{m \mid m \in (p' \cup Adv_H) \wedge m \in M_\rho \wedge O_H(m) \subseteq s'.E\} \\
s'.pool &= \begin{cases} (s.pool \cup \{m\}) \setminus M & \text{if } e = (m.from, send, m) \\ s.pool \setminus M & \text{otherwise} \end{cases}
\end{aligned}
$$

and $p' = \begin{cases} (s.pool \cup \{m\}) & \text{if } e = (m.from, send, m) \\ s.pool & \text{otherwise} \end{cases}$

Consider any execution in the product transition system $\rho' = C_0 \xrightarrow{l_0} C_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{g-1}} C_g$. We first prove that $H_{\rho'} \preceq H_\rho$.

First we prove the downward closure property. We need to prove that $E_{\rho'} \subseteq E_\rho$ and for any event $e \in E_\rho$,

$$e \in E_{\rho'} \Rightarrow (\forall e', e' <_\rho e \Rightarrow e' \in E_{\rho'} \wedge e' <_{\rho'} e)$$

Let us consider the case when $e = (m.to, receive, m)$. We know that in ρ' every RECEIVE step at i is preceded by a MONITOR step at $j < i$. That is, $\delta_\mu(s_j, e_j) = (s_{j+1}, M)$ where $m \in M$. By the definition of M,

1. $O_H(m) \subseteq s_{j+1}.E \subseteq E_{\rho'}$. In other words all events e' where $e' <_\rho e \Rightarrow e' \in E_{\rho'}$. Furthermore, we can say that e' has occurred in ρ' at step $j < i$. When e' is in the same process, by the definition of *happens before*, this is sufficient to say that $e' <_{\rho'} e$.
2. $m \in M_\rho$. This implies that $M_{\rho'} \subseteq M_\rho$

Let $(e_1, e_2, \cdots, e_k)$ be the sequence of receive events at a process $r \in \mathcal{P}$ in the execution ρ. Consider the largest i such that $e_i \in E_{\rho'}$, then as a consequence of (1) $(e_1, e_2, \cdots, e_i)$ is the sequence of receive events for process r in the execution ρ'. In other words, the sequence of messages delivered to a process in ρ' is a prefix of the same sequence in ρ.

Consider the projection of ρ, ρ' for a given process $r \in \mathcal{P}$. That is, the steps that correspond to state changes for the process r. In this projection for ρ, let $(s_0, s_1, \cdots, s_g)$ and $(e_1, e_2, \cdots, e_g)$ be the sequence of states and events respectively, where g is the length of ρ' (as ρ' can be smaller than ρ). By induction on i, we prove that s_i, e_i is the state of the process and the event emitted by r in the execution ρ'. When $i = 0$, this is trivially true since the initial state is the same for both the executions. Assuming it is true at step i, we need to prove that s_{i+1} is the state in ρ'. From a given state and by the definition of δ_π, one of either $\delta_\pi(s_i, \perp)$ or $\delta_\pi(s_i, m)$ (for some $m \in \mathcal{M}$) is defined.

1. $\delta_\pi(s_i, \perp) = (s_{i+1}, e_i)$. Then, it is the same in ρ' (as δ_π is a function)
2. $\delta_\pi(s_i, m) = (s_{i+1}, e_i)$. This is true because (1) the sequence of messages delivered in ρ' is the same and hence m will be the same and (2) δ_π is a function

Since the sequence of events in each process observed in ρ' is a prefix of that observed in ρ, we infer that $E_{\rho'} \subseteq E_{\rho}$. Futhermore, combined with the property that all events before a receive in ρ also occur in ρ', we infer that $H_{\rho'} \preceq H_{\rho}$ and that $\forall e, e' \in E_{\rho'}, e <_{\rho'} e' \Leftrightarrow e <_{\rho} e'$ □

To prove Theorem 1, we define a very constrained monitor that reproduces exactly one history in every accepting run. However, in practice, such a restrictive monitor is not very useful. When testing an implementation in a constrained state space, the algorithm is more likely to find bugs when it explores more states. Therefore, in practice, we fix only specific events of the history and allow the *monitor* to explore other histories. Theorem 1 shows that even under the relaxed scenario, the *monitor* will be effective in constraining executions.

In the next section, we define the syntax and semantics of NETRIX unit tests. The filters that make up a unit test allow the developer to fix specific events and therefore construct a monitor that constrains executions.

3.3 NETRIX unit tests and DSL

The core of the DSL contains primitives to define *condition* and *action* parts of the filter. Recall that a filter is of the form **If(condition).Then(actions)**. However, in general, a condition and action are functions. A *condition* is a function that accepts the monitor state and event to return a boolean value. Similarly an *action* is a function that accepts the monitor state and event to return a sequence of messages and may also change the state. In our DSL, the monitor state is represented by a `Context` object.

3.3.1 Syntax

Table 3.1 and 3.2 lists the generic conditions and actions provided with the DSL. Using these primitives one can define filters to drop messages, reorder them and introduce network partitions. For example, to isolate a single process p from the rest the filter would be

```
If(IsMessageFrom(p).Or(IsMessageTo(p)))
  .Then(DropMessage()))
```

Similarly to reorder the message, we need two filters - one to capture the message and one to release the message. For example with Raft, to reorder all term 1 messages to after term 2 the filters would be as follows

```
If(IsMessageTerm(1))
  .Then(MessageSet("term1").Store()))
If(IsNewTermEvent(2))
  .Then(MessageSet("term1").DeliverAll()))
```

Note that MessageSet().Store() and MessageSet().DeliverAll() uses the context object to record the messages. Having access to the state allows us to write complex scenarios where we can count and precisely time the delivery of messages.

Table 3.1: Semantics of conditions for event e and context ctx

Condition	Return value
IsEventOf(r)	true if e.replica = r
IsEventType(t)	true if the e.type = t
IsMessageType(t)	true if e.type=send(m)/receive(m) and m.type=t
IsMessageSend	true if e.type=send(*)
IsMessageReceive	true if e.type=receive(*)
IsMessageFrom(r)	true if e.type = send(m)/receive(m) and m.from=r
IsMessageTo(r)	true if e.type = send(m)/receive(m) and m.to=r
IsMessageBetween(r_1, r_2)	true if e.type = send(m)/receive(m) and $\{m.to, m.from\} = \{r_1, r_2\}$
Count(c).Lt(v)	true if ctx[c] < v
Count(c).Gt(v)	true if ctx[c] > v
Count(c).Leq(v)	true if ctx[c] <= v
Count(c).Gte(v)	true if ctx[c] >= v
MessageSet(s).Contains	true if e.type = send(m)/receive(m) and m $\in$ ctx[s]
c1.And(c2)	true if c1(e,ctx) $\wedge$ c2(e,ctx)
c1.Or(c2)	true if c1(e,ctx) $\vee$ c2(e,ctx)
c.Not	true if !c(e,ctx)

While the conditions and actions provided are generic to all distributed systems, a user may extend the DSL by defining custom conditions and actions. Extensions are straight forward since conditions and actions are defined semantically as function. For example in the scenario described above, we use two custom conditions IsMessageTerm and IsNewTermEvent. In Section 3.5.3, we list extensions to the DSL we use in testing specific implementations. Note that the DSL is embedded [FP10] in go programming language.

In addition to the filters, a unit test consists of a state machine that encodes a property. Our DSL also provides a generic builder pattern to construct the state machine. The transitions of the state machine are labelled by *conditions*. Therefore, it is possible to reuse the primitives defined earlier.

Table 3.2: Semantics of actions for an event e and context ctx

Action	Return value	Context changes
DeliverMessage	if e.type=send(m) returns m	-
MessageSet(s).Store	empty set	ctx[s] = ctx[s] $\cup$ m where e.type=send(m)/receive(m)
DropMessage	empty set	-
MessageSet(s).DeliverAll	returns ctx[s]	ctx[s] is set to empty
Count(c).Incr	empty set	ctx[c]++
RecordMessageAs(l)	empty set	ctx[l] = m where e.type = send(m)

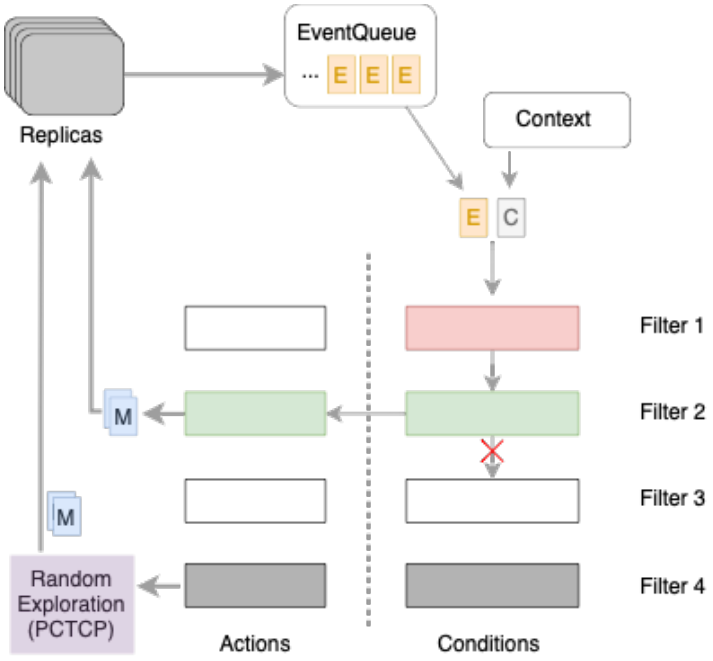


Figure 3.7: Semantics of executing filters

3.3.2 Semantics

The semantics of the DSL essentially define the monitor transition function δ_μ from Section 3.2. The monitor is parametrized by the set of filters, the state machine that encodes a property, and an underlying exploration algorithm.

Recall that the transition function δ_μ is defined as $\delta_\mu : \Sigma_\mu \times \mathcal{E} \rightharpoonup \Sigma_\mu \times 2^{\mathcal{M}}$. The state of the NETRIX monitor consists of - the context, the current state of the state machine, and the state of the underlying exploration algorithm. The transition function encodes how we execute the filters - similar to a switch case.

Figure 3.7 captures the semantics of the transition function. We invoke the filter conditions by passing the event and the context. If a condition returns true, then the corresponding action decides both the new state and the set of messages to deliver (or not). When none of the conditions return true, the event is passed to the underlying exploration algorithm which subsequently decides the set of messages to deliver.

Let us consider two example unit tests to better understand the semantics. First, a unit test with no filters. The execution steps are as follows - (1) the event is fed to the property state machine. Then, (2) the event is passed to the underlying exploration algorithm to update its state. Finally (3) the underlying exploration algorithm decides which messages to deliver.

Second, a unit test with a single filter that drops messages of a particular type t. In that case, the execution is as follows for each event - (1) The event is passed to the filter condition. (2) If the condition is true, then the corresponding action (Drop) is executed. (3) If the condition is false, then the event is passed to the underlying execution.

For unit tests that contains more than one filter, for each event the filters are executed similar to a switch case - sequentially until the first condition is satisfied. Formally, each

filter is a function that accepts a state and an event and returns a new state and a set of messages. The filter inturn is a composition of a condition and action.

Apart from the filters, each event is passed to the property state machine that accompanies the unit test. The accepting states of the *monitor* (F^{μ}) parameterized by the unit test are exactly those states where the property state machine is accepting.

3.3.3 Extending the DSL

As mentioned earlier, a developer may choose to extend the DSL with custom conditions and actions. To introduce Byzantine failures where message contents are changed, extending the DSL is mandatory as the message serialization and de serialization mechanism is specific to each implementation.

Apart from the conditions and actions in Tables 3.1 and 3.2, our DSL provides partitions as first class primitives. The goal of partition primitives is to enforce a logical separation between processes during an execution. During initialization, a new partition can be created using **NewRandomPartition** which accepts a sequence of integers representing the size of each partition. The conditions related to partitions are **IsMessageFromPart(p), IsMessageAcrossPartition** and **IsMessageWithinPartition**.

3.4 Implementing Netrix

In this section we present the implementation of Netrix which consists of two parts - the instrumented implementation and the runtime. The instrumented implementation creates events that are processed by the runtime to guide the exploration. The runtime is a concrete implementation of the *monitor* introduced in Section 3.2.

3.4.1 Instrumentation

The instrumentation consists of a thin shim around the transport interface of the implementation. Implementation provide transport abstractions to send and receive messages from other processes. By instrumenting, we inject a shim to such a transport interface. The shim will send messages to the Netrix runtime instead of the intended processes. Apart from communicating the messages, the shim will also send events related to the messages such as message send and receive events. In addition to these events, a developer can further instrument the implementation and capture other events that are specific to the protocol. For example with Raft, the developer can instrument the implementation to capture events such as Becoming a leader, a term timeout, etc.

The shim transmits all events and messages to dedicated interfaces of the runtime. We generalize the interface that the client interacts with and provide language specific client libraries that can be utilized by the shim. The details of the interface is discussed in Section 3.4.2.

Apart from communicating information to the runtime, the shim needs to also receive information from the runtime - such as messages, instructions to start, stop or restart, etc. We

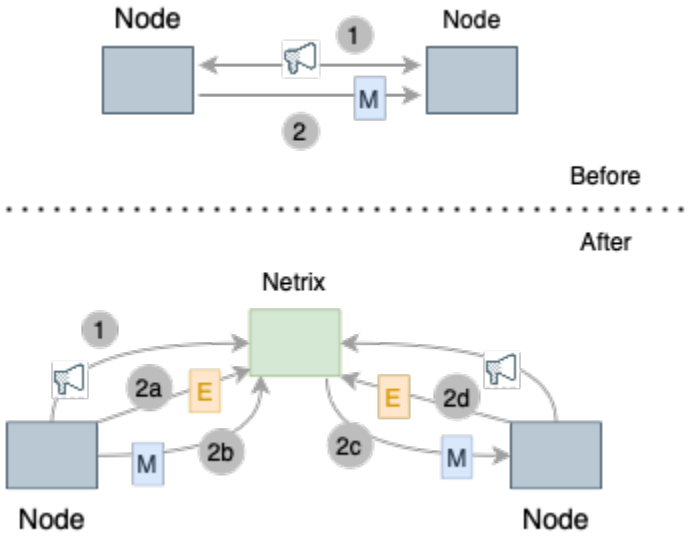


Figure 3.8: Instrumentation workflow

1 Discovery process, nodes establish connection to Netrix (**After**) as opposed to discovering each other (**Before**)

2 **Before**: nodes exchange messages. **After**: (a) Nodes send a message send event to Netrix (b) followed by the message. (c) Node receives the message and (d) submits a receive event

generalize the interface of the shim that is used by the runtime to communicate information to the processes. The interface consists of the following two endpoints.

1. **ClientMessageEndpoint** - To receive messages from the Netrix runtime.
2. **ClientDirectiveEndpoint** - To receive directives such as start, stop and restart from the runtime.

Figure 3.8 describes the difference in workflow before and after instrumentation. The instrumentation allows Netrix to capture all the messages. After instrumentation, nodes communicate to the runtime as opposed to communicating between themselves.

3.4.2 Runtime

The Netrix runtime (written in `go`) drives the exploration of the state space. It is parameterized by an underlying exploration algorithm (such as PCTCP) and a unit test. While the client libraries are language specific, the Netrix runtime is independent of the language of the implementation. Therefore, developers can use Netrix to test implementations of any language.

To drive the exploration, the runtime captures events and messages from the instrumented nodes and feeds them to an exploration `Strategy`. In our evaluation, we use a specific strategy - `PCTStrategyWithTestCase` - that implements the semantics of a Netrix unit test with PCTCP as the underlying exploration algorithm.

Snippet 3.1: TestOne

```
function TestOne() *TestCase {
  ...
  filters.AddFilter(
    If(IsMessageType("Prepare"))
    .Then(DropMessage())
  )
  return  NewTestCase("TestOne",sm,filters)
}
```

A NETRIX unit test is represented by a `TestCase` object that includes the filters and a property state machine. For example, Snippet 3.1 implements a simple unit test with a single filter - one that drops all `Prepare` messages. Note that the function returns a new `TestCase` that consists of the filter and a state machine that encodes a safety assertion.

Once initialized with a `Strategy`, the NETRIX runtime creates the necessary data structures to hold messages, events and to communicate with the nodes. The workflow steps are as follows. (1) The runtime starts an RPC server to communicate with the nodes, (2) Waits till all nodes Register with the runtime, (3) Creates an `EventQueue` and a `MessageStore` to store events and messages respectively, and (4) starts running the test iterations. The inner loop of running the iterations involves (1) initializing a new `Context`, (2) invoking the function `Strategy.Step` with an event and a `Context` (3) delivering the messages to the nodes after the step, and finally, (3) at the end of the iteration, sending a restart directive to all nodes.

The RPC server of the runtime has the following endpoints,

1. **RegisterEndpoint** - Invoked initially to register a process with the runtime. Used to transmit keys, process IDs and other information needed by the runtime. The keys are essential to inject Byzantine faults where the runtime needs to sign the modified message with the process keys.
2. **MessagesEndpoint** - To receive messages from the clients of the processes.
3. **EventsEndpoint** - To receive events from the clients of the processes.

The runtime is not specific to any particular distributed system implementation and contains generic data structures to events and messages. A few more features of the implementations allow NETRIX to be used to test implementations of any language - the generic endpoints, allow for custom handling of start, stop and restart directives, and generic *conditions* and *actions* that make up the filters of a `TestCase`.

3.5 Evaluation

To evaluate NETRIX, we instrument and test three open source consensus protocol implementations. **Tendermint**[1] [BKM18] (version 34.3), **Raft**[2] [OO14] (version v3.5.2) and

[1]https://github.com/tendermint/tendermint
[2]https://github.com/etcd-io/etcd/tree/main/raft

BFTSmart[3]. The Tendermint protocol is a Byzantine consensus algorithm inspired by PBFT and is the backbone of the cosmos network[4]. We test the official implementation of Tendermint written in Go. Similarly, BFTSmart[BSA14; SB12] implements in Java a Byzantine consensus algorithm. It is used to build key-value stores and distributed file systems. Raft is a popular benign consensus protocol that tolerates crash failures. We test the Go implementation that is used in many cloud services such as `etcd` and distributed graph databases such as `dgraph`. For the three benchmarks, the instrumentation effort aided by our language specific libraries needed to test these implementation is very little and is evident in the size of the instrumented code added - 600 LOC for Tendermint codebase of 150kLOC, 120LOC for Raft codebase of 16kLOC and 150LOC for BFTSmart codebase of 16kLOC.

For the three implementations, we write a total of 34 unit tests with the aim of answering the following two research questions,

RQ1 Do Netrix unit tests help bias the exploration?
RQ2 Does biased exploration help uncover bugs and improve bug reproducibility?

We find that the answer to the research questions as Yes, Netrix unit tests constrains the exploration based on the filters provided and with the biased exploration, we uncover 4 instances where the implementation deviates from the protocol specification in Tendermint. We demonstrate additional benefits to testing implementations with Netrix unit tests. Namely, (1) the unit tests are concise and easy to write - unit tests have on average 2.5 filters, (2) the unit tests help with regression testing - We were able to run the tests on multiple versions of the implementation with little or no changes, (3) Netrix can be used to test implementations written in any language - we test implementations written in Java and Go and finally (4) the biased exploration with unit tests allow the developer to gain confidence even when no bugs are found.

Experimental setup The 34 unit tests were run on Intel Xeon(R) 16-core CPU with a clock rate of 2.13GHz and a memory capacity of 128GB. Each test was run for 100 iterations with a benchmark specific timeout for each iteration. The test harness consists of client requests to each of the respective benchmarks.

3.5.1 RQ1: Do Netrix unit tests help bias the exploration?

Yes, Netrix unit tests bias the exploration. We measure the number of biased executions by encoding the constrainment criteria in the property state machine that accompanies the filters. When the state machine reaches a success state, we count the outcome as successful. Table 3.3 lists the outcomes for the 34 unit tests for the 3 benchmarks. In 27 out of the 34 unit tests, we reach an outcome greater than 50%. The outcomes for the remaining 7 tests are low due to fewer constraints on the execution. Meaning, the number of filters allow for executions that does not always satisfy the property that corresponds to the test. While adding more filters improves the outcomes, these tests demonstrate that even with missing filters, the exploration sufficiently satisfies the expected outcomes.

[3]https://github.com/bft-smart/library
[4]https://cosmos.network

Name	#F	#S	LOC	Outcomes	Name	#F	#S	LOC	Outcomes
Tendermint					**Raft**				
ExpectUnlock*	3	5	90	41/100	Livenessˆ	5	3	64	15/100
Relocked*	4	5	115	53/100	LivenessNoCQˆ	5	3	64	100/100
LockedCommit*	3	5	85	100/100	NoLivenessˆ	5	3	33	100/100
LaggingReplica*	3	4	71	100/100	ConfChangeBugˆ	5	2	94	55/100
ForeverLaggingReplica*	5	5	89	100/100	DropHeartbeat	2	3	69	100/100
RoundSkip	3	4	74	90/100	DropVotes	1	3	44	80/100
BlockVotes	2	3	55	33/100	DropFVotes	1	2	57	100/100
PrecommitInvariant	1	3	68	100/100	DropAppend	1	3	81	100/100
CommitAfterRoundSkip	3	3	82	36/100	ReVote	2	3	54	74/100
DifferentDecisions	8	3	180	20/100	ManyReVote	2	4	64	92/100
NilPrevotes	2	3	61	99/100	MultiReVote	2	4	60	81/100
ProposalNilPrevote	1	3	56	56/100	**BFTSmart**				
NotNilDecide	2	2	49	100/100	DPropForP	2	3	60	81/100
GarbledMessage	1	2	68	30/100	DPropSame	2	2	40	100/100
HigherRound	1	3	91	37/100	DropWrite	1	2	30	100/100
					DropWriteForP	1	2	33	89/100
					ExpectNewEpoch	1	2	28	94/100
					ExpectStop	1	2	38	100/100
					ByzLeaderChange	3	2	46	89/100
					PrevEpochProposal	2	3	53	99/100

Table 3.3: **List of unit tests**
The table lists unit tests grouped by the protocol. The columns are number of filters, state machine states, LOC and outcomes in number of iterations that successfully caught the interesting scenario/bug. * indicates new bugs found and ˆ indicates tests for replicating known bugs

The unit tests are quick to run with each test (100 iteration) running for a few minutes since each test iteration takes $\approx$ 2s on average. Since running the test is not compute intensive, the performance of the tests relies heavily on the latency of network communications.

3.5.2 RQ2: Does biased exploration help uncover bugs and improve bug reproducibility?

In the case of Tendermint, we uncover 4 new bugs. These bugs are protocol deviations - instances where the implementation deviates from the protocol specification. In Table 3.3, the respective unit tests are marked. One of them is a performance bug. A process is isolated while the rest progress to higher rounds. When the process reconnects, it fails to immediately synchronize and the duration required to synchronize increases as the gap in the number of rounds increases. Essentially, the remaining participating processes operate with fewer correct processes.

The bugs with Tendermint are reproduced with high outcomes - indicating a significant improvement in bug reproducibility. We reported the bugs to the developer team and 3 of the 4 bugs were duly fixed. Furthermore, we ran the unit tests on the fixed version of the implementation and failed to find any bugs indicating higher confidence in the correctness of the fix.

For Etcd, we did not uncover any new bugs with the unit tests. However, we reverted to an older version of etcd which included 4 previously known bugs. The respective tests are marked in Table 3.3. We see that except for one bug, the remaining 3 bugs are reproduced with high outcomes. The one bug that was not reproducible was due to high level of non-determinism. Specifically, the number of timeouts that need to be synchronized to reproduce the bug is high and therefore very few executions reproduce the bugs.

While, we did not uncover any bugs with BFTSmart, we were able to debug and gain confidence in the correctness of the BFTSmart implementation.

3.5.3 DSL extensions

In the process of testing, we develop custom *conditions* and *actions* for each of the benchmarks presented. Table 3.4 lists the customizations we develop. To recall, *conditions* are functions that accept an event and context to return a boolean, and an *action* is a function that accepts an event and context to return a set of messages to deliver while mutating the context. Some of the custom conditions we list are only valid for message send and receive events (e.g. IsAcceptingVote, CountVotes).

The custom conditions listed are of two types *pure* - does not rely on the context and only depends on the current event (e.g. IsAcceptingVote) - or *stateful* - relies on the information stored in context (e.g. RoundReached). Similarly, actions are pure (e.g. ChangeVoteToNil) or stateful (e.g. CountLeaderChanges) depending on whether they mutate the context or not.

Similar to tradition unit-tests where the primitives (mocks) are reused, the corpus of custom conditions and actions are reusable across different filters in different unit tests.

3.6 Motivating scenarios

As demonstrated, Netrix unit tests are effective in constraining exploration, finding and reproducing bugs more frequently. However, the developer is now responsible for writing the unit tests. To aid the developer, we provide a general methodology to generate unit tests from the protocol specification and proofs.

We rely on three sources to generate unit tests - the protocol specification, the proofs of the protocol and interactions with the developers. The guidelines we describe to generate unit tests are general to distributed systems and not tied to the specific protocols we test. We will use unit tests and scenarios interchangeably.

Protocol Specification Given a protocol specification, one can derive unit tests where no failures occur. The corresponding *happy path* can be encoded in the state machine of the unit test. Alternatively, trying to cover all lines of the protocol specification also leads to test scenarios. For example, consider the following lines from the Tendermint protocol specification

Table 3.4: Table of custom conditions and action for the benchmarks

Name	Benchmark	Type	Description
IsStateLeader	Etcd	Condition	Checks if the event corresponds to a state change where the new state is a leader
IsAcceptingVote	Etcd	Condition	Checks if the message is a vote and that the vote is accepting
IsNewTerm	Etcd	Condition	Checks if the event corresponds to a term change
RecordTerm	Etcd	Action	Records the term of the event in context
CountLeaderChanges	Etcd	Action	Keeps track of how many leader changes have occurred using context
CountVotes	Etcd	Action	Counts the number of messages of type vote received so far
IsNilCommit	Tendermint	Condition	Checks if the message is a commit for the nil block
IsCommitForProposal(p)	Tendermint	Condition	Checks if the messages is a commit for a specific proposal p
IsMessageFromRound(r)	Tendermint	Condition	Checks if the round number contained in the message is r
RoundReached(r)	Tendermint	Condition	Using the context, checks if all the nodes have reached round r
IsEventNewRound	Tendermint	Condition	Checks if the current event is a new round event
ChangeVoteToNil	Tendermint	Action	If the event corresponds to a vote message, then the action changes the vote to nil
RecordProposal	Tendermint	Action	Records the proposal contained in the event (if any)
IsView(v)	BFTSmart	Condition	Checks if the message is from view v
IsNewEpoch	BFTSmart	Condition	Checks if the event corresponds to a epoch change
GarbleValue	BFTSmart	Action	Corrupt the value of the corresponding message to a randomly chosen value

```
Upon (f+1) messages from a higher round r
  Transition to round r
```

To simulate this scenario, we isolate one process p and do not deliver any messages from round 0. We then force the remaining processes to move to round 1. According to the protocol specification, after receiving $f + 1$ messages from the round 1, the isolated process transitions to round 1. To assert the scenario, we define a custom condition ProcessNewRound(p,r) which is true when we observe a round change event from the process p for the round r. Specifically, we found that the Tendermint implementation fails this unit test and the isolated process fails to catch up the other processes immediately. The time required for the catchup increases as the gap in rounds between the lagging process and the remaining process increases. The Tendermint team has acknowledged the bug.

We introduce a notion of protocol coverage to justify the expressiveness of the DSL. As mentioned above, a protocol is typically specified as a sequence of 'Upon' clauses followed by a set of execution rules. We say a unit test covers a protocol clause if, in the executions explored by the unit test, the clause is satisfied at least in one process. With our unit tests, we covered all the clauses as defined in the protocol specification for both Tendermint and Raft.

Protocol proofs Apart from the protocol specification, the developer can derive test scenarios from the proofs of the protocols. For example, consider the following inductive invariants used in the proof of the Tendermint protocol[5]:

$$v \neq nil \wedge \exists p.precommitted(p, r, v) \rightarrow \\ \exists quorum. \forall p.p \in quorum \rightarrow prevoted(p, r, v)$$

This formula states that if a process p `Precommits` a non nil value v in round r, then a quorum of validators should have sent `Prevote` messages for v in round r. This is an implication of the form $A \rightarrow B$, and the corresponding unit test contains filters to ensure $\neg B$, i.e., a quorum of processes do not `Prevote` on the `Proposed` value, and a state machine that reaches the fail state if it observes A, i.e., it observes a `Precommit` from any validator.

3.7 Filter distance

To further aid the developer when writing tests, we define a new syntactic measure *filter distance* to identify the necessary filters when using PCTCP as the underlying exploration algorithm. The developer can forego writing filters with shorter distance and still obtain high outcomes. We demonstrate the same by evaluating a few of the unit tests.

We group the filters that make up the 34 unit tests into 3 categories. For each category, we define the *filter distance* metric that signifies the importance of the filter in a unit test. The *filter distance* is specific to the underlying exploration algorithm and signifies the relative

[5]https://github.com/tendermint/spec/tree/master/ivy-proofs

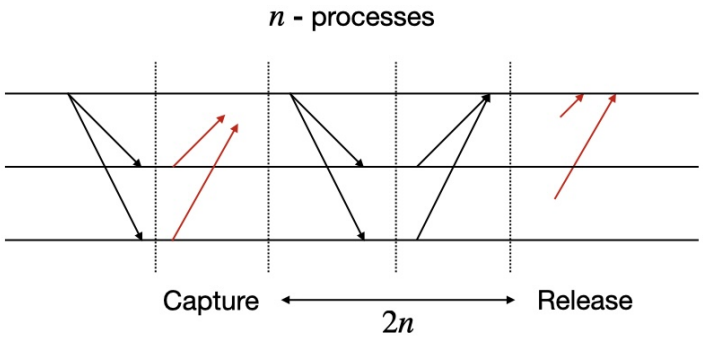


Figure 3.9: Filter distance of a pair of capture and release filters. The release occurs after 2 rounds of communication.

importance based on the underlying algorithm. Filters with low distance measure are less important and the filters with higher distance are necessary.

The three categories of filters are as follows,

1. **Byzantine** - Filters that introduce Byzantine behavior
2. **Drop** - Filters that drop messages
3. **Reorder** - Filters that reorder messages

The Reorder filters can be grouped into pairs. Ones that capture the message and store it in a set and ones that release the messages stored in a set. Similarly, drop filters also consists of pairs where the release filter is triggered after the end of the execution. The distance metric is defined for a pair of capture-release filters and can be determined syntactically.

We define the distance as the number of messages between the capture and release filter in a normal execution of the protocol (without any faults). Let us consider an example from an abstract round based protocol. The example illustrated in Figure 3.9 is as follows, the capture filter captures messages of round r and the release filter releases the messages after round $r + 2$. And in each round $O(n)$ messages are exchanged. Then, the distance of the pair of capture-release filters is $2n$.

Let us consider a second example, a NETRIX filter, to understand the *filter distance measure* in relation to PCTCP,

```
If(IsMessageOfType(t))
  .Then(DropMessage())
```

The filter describes scenarios where all messages of type t are dropped. To observe a similar execution with PCTCP, all the dropped messages have to be scheduled at the end of the execution. Therefore, d will be set to the length of the execution which is very high. PCTCP is very unlikely to explore scenarios where this occurs due to the large d. Additionally, since

Test Name	Protocol	Filter distances	PCTCP outcomes
Liveness	Raft	∞	0
		n^2	0
		n	19/1000
LivnessNoCQ	Raft	∞	0
		n^2	0
		n	19/1000
DropAppend	Raft	n	$21/1000$
ReVote	Raft	n	$409/1000$
ManyReVote	Raft	$2n$	$14/1000$
MultiReVote	Raft	$3n$	$1/1000$
RoundSkip	Tendermint	$r \times n$	$6/1000$

Table 3.5: **List of Filter distances**
Unit tests where filter distances are small and PCTCP is able to explore expected re-orderings. Distance - measures for each filter pair. Outcomes - number of successful iterations when we remove that filter and run PCTCP. We denote distance as an asymptotic measure where n is number of processes and r is the number of rounds.

PCTCP does not have the ability to introduce Byzantine failures, we define the distance as $\inf$ for those filters that introduce Byzantine failures.

To recall from Section 3.1.2, PCTCP algorithm observes the set of messages sent during an execution, stores them into a set of (causally-ordered) chains, and delivers messages from these chains according to a random strategy. The probability of exploring executions with d-message re-orderings is $\frac{1}{w^2 h^{d-1}}$. Therefore, the probability decreases as depth d increases.

We observe that PCTCP fails to explore executions where the re-orderings are beyond a certain distance threshold - $O(n^2)$. However, the filters with short distance measures (1-2 communication rounds or $O(n)$) are not crucial and PCTCP is able to explore the respective re-ordering. Table 3.5 lists the distance of all the filters for specific unit tests. The PCTCP outcomes column signifies the outcomes when we run the test without including the filter. If we remove a filter with high distance, the outcomes are low and therefore the filter is important.

3.8 Related Work

Our work is inspired by Concurrit [Elm+13], which enables a similar scenario-based testing approach for *multi-threaded* concurrent programs. It introduces a DSL that enables developers to define tests where they can control the scheduling between threads with a minimal instrumentation effort. This DSL is specific to multi-threading and very different compared to Netrix DSL which is specific to testing implementations of distributed protocols. GFuzz [Des+18a] applies the idea of exploring different message orderings between concurrent go channels and has demonstrated success in finding concurrency bugs in actual implementations. P-sharp [Del+15a] is an actor based programming language that

allows developers to write asynchronous systems. P-sharp is embedded in the C-sharp programming language and is accompanied by a systematic concurrency testing framework. Similar to GFuzz, P-sharp explores arbitrary event orderings between the actors to find concurrency bugs. However, both GFuzz andP-sharp do not allow describing specific scenarios to test.

Our DSL primitives are motivated by specification languages for protocols such as DISTAL [Del12] where programs are a sequence of *Upon* clauses. Each *Upon* is followed by a predicate on the state of the protocol and current message. Similar to our DSL, DISTAL predicates contain counting, sets of messages and comparing message types. ModP [Des+18b] language allows protocol designers to describe and test a model of the protocol. Similar to DISTAL, ModP machines contains a sequence of *on* event handlers that modify the state of the machine. The *on* handlers are followed by predicates similar to DISTAL. ModP also generates code for testing the programs. While these are effective in finding bugs in a *model* of a protocol, the results however do not help in testing production implementations. The main reason they do not help in production environments is because model checkers do not scale when applied directly on implementation of large systems. Moreover most model-checkers that work at the programming language level, are not even applicable to this application domain, as they focus on primitives for shared memory and not message passing [Kov+24; Bor+24]. Therefore our DSL provides the only guided way to do exploration of the execution space on implementations.

Chapter 4

WaypointRL: biased exploration using reinforcement learning

In the previous chapter we discussed a novel unit testing approach to distributed systems - Netrix. The unit tests allow a developer to encode a specific scenario and enforce the exploration to only explore executions that satisfy that scenario. In this chapter we explore an alternative method to bias exploration using Reinforcement Learning where an agent *learns* the scenario.

In reinforcement learning (RL), an agent interacts with an environment by observing the state and picking actions that lead to new states. The goal of the agent is to collect maximum rewards from the environment. The standard example is that of a robot learning to navigate a maze with the goal of finding a way out. The robot is equipped with an RL agent and the maze is the environment. Over many iterations (episodes) with a fixed number of steps (horizon), the robot learns a path out. The maze rewards the agent every time it finds a way out. While in the maze example the transitions are deterministic, the algorithms driving the RL agent are equipped to explore non-deterministic transition systems as well. The reward collected in an episode are propagated back with time. If the reward is sparse - too infrequent, the agent will revert to random exploration.

RL techniques have achieved significant success in many domains over the years. Naturally, in this chapter we explore the applicability of RL to testing and specifically to testing distributed systems. However unlike a standard environment with rewards, the goal of an RL agent in testing is to explore as many states as possible thereby increasing the probability of finding bugs. Naturally, the question now translates to finding the best reward mechanism to ensure maximal exploration (or pure-coverage). Existing works [Muk+20; Men+23b; Red+20] propose a punishment based reward scheme where the agent is punished proportional to the number of visits to a state. As a consequence, the agent now learns to pick actions that leads to fewer-visited states.

Our first contribution in this chapter is a new reward mechanism BonusMaxRL for pure-coverage that is motivated by recent theoretical results in reward-free reinforcement learning [Jin+20; ZMS20]. In BonusMaxRL, we introduce two new mechanisms to propagate rewards. First, the reward for a new state decays based on the number of visits ($1/k$ where k is the number of visits). With the decaying reward, the RL agent is incentivized to pick

actions that lead to new states in the short run. Second, the standard reward back propagation in BonusMaxRL prioritizes observing new states immediately instead of optimizing for pure-exploration. We elaborate on the differences later on in Section 4.1. Overall, BonusMaxRL achieves surprisingly high coverage and is comparable to existing exploration algorithms. Furthermore, the positive decaying reward allows us to build a biased exploration algorithm WaypointRL

By leveraging the insights gained from testing implementations using Netrix, we design a new RL based algorithm WaypointRL where the developer specifies waypoints to characterize a scenarios (as opposed to filters). However, similar to Netrix, the goal of WaypointRL is to bias exploration towards the specified scenarios. The waypoints serve as intermediate rewards for the agent and therefore the agent automatically learns to explore only those executions where the waypoints are satisfied. The waypoints are generic predicates over the state space that a developer can design just with a high level knowledge of the protocol. For example with Raft, a high level scenario would be observe a leader to be elected. In general, the waypoints serve as intermediate goals that lead to 'interesting' states of the protocol where bugs are more likely.

In this chapter, we empirically demonstrate the exploration capabilities of BonusMaxRL and also show that WaypointRL is effective in biasing exploration for 3 benchmarks - RedisRaft, Etcd and RSL. Together, the combination of pure-coverage BonusMaxRL and biased-exploration WaypointRL outperforms existing approaches in finding new bugs and achieving high target coverage.

The rest of the chapter is organized as follows - Section 4.1 introduces the two new RL algorithms BonusMaxRL and WaypointRL after presenting a background in Reinforcement Learning. Then, in section 4.2, we model the distributed systems as a RL environment and compare to the modelling of a Monitor from Chapter 3. Subsequently, we describe a general methodology to describe waypoints for RL in Section 4.3 before presenting our evaluation of the two algorithms in Section 4.5. Finally, we discuss existing work related to Reinforcement learning and testing in Section 4.6.

4.1 Reinforcement Learning with Coverage Bonus and Waypoints

To begin with some background, the standard RL agent [SB18] explores an environment modelled as a Markov Decision Process (MDP). The agent interacts with the environment for a fixed number of episodes. In each episode, the agent starts from a common initial state and picks actions that lead to new states. We are interested in exploring the state space of a distributed system and therefore model the system as a MDP where the state is an abstraction of the states of each process and actions correspond to delivering messages while crashing processes or introducing network partitions. Formally, we define an MDP as follows,

Definition 6. *A **Markov Decision Process (MDP)** is the tuple* $(\mathcal{S}, \mathcal{A}, s_0, \mathcal{T}, \mathcal{R})$ *where*

- $\mathcal{S}$ *defines the state space,*

Algorithm 1: Generic RL loop

Input: K: number of episodes, H: episode horizon, E: environment, A: agent

$A.init()$

for *episode* $k = 1, \cdots, K$ **do**
  $state_1^k \leftarrow E.reset()$
  $A.newEpisode(state_1^k)$
  for *step* $h = 1, \cdots, H$ **do**
    $action_h^k \leftarrow A.pick(state_h^k, E.actions(state_h^k))$
    $state_{h+1}^k, reward_h^k \leftarrow E.step(action_h^k)$
    $A.recordStep(state_h^k, action_h^k, state_{h+1}^k, reward_h^k)$
  $A.processEpisode()$

- *$\mathcal{A}$ is the action space,*
- *$s_0 \in \mathcal{S}$ denotes the initial state,*
- *$\mathcal{T}(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ - a transition probability function; we write $\mathcal{T}(s, a, s')$ to denote the probability of the transition $s \xrightarrow{a} s'$,*
- *a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ that specifies a reward for a specific transition.*

With RL, our goal is to learn a policy $\Pi : \mathcal{S} \to \Delta(\mathcal{A})$ that maps each state to a probability distribution over the actions $\mathcal{A}$ so that the expected discounted sum of rewards is maximized.

Algorithm 1 describes a generic Reinforcement Learning loop. It takes as input the number of episodes K, the horizon H, the environment being explored E, and an RL agent A. The environment behavior is captured by its functions - $reset$ to start a new episode from its initial state, $actions$ to return the available actions in the current state, and $step$ to transition in a new state according to the agent's picked action.

Each episode starts from the environment's initial state s_0. At each step, the RL agent selects the next action from the set of possible actions based on the policy, and observes the resulting state and reward. The agent is characterized by the following functions. $newEpisode$ at the beginning of each episode, $recordStep$ after each step, $pick$ determines the next action and, $processEpisode$ at the end of the episode. Different agents instantiate the functions based on their respective policies. In general an agent implementation uses $newEpisode$ to reset the current episode trace, $recordStep$ to append a transition to the trace, and updates the policy using the full trace in $processEpisode$.

A popular RL algorithm is Q-Learning [WD92]. In Q-Learning, the agent keeps an estimate $Q(s, a)$ of the expected reward for a given state (s) and action (a) pair called Q-value. For a given transition (s, a, s'), and its associated reward $r(s, a, s')$, Q-learning updates the Q-value as follows:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r(s, a, s') + \gamma \max_{a'} Q(s', a'))$$

where $\alpha, \gamma \in [0, 1]$ are hyper-parameters representing the learning rate and discount factor respectively. Intuitively, each time the outcome of picking a state-action pair is observed, its Q-value is updated based on the observed reward and estimated value of the next state.

α determines how quickly the Q-value changes at each update, while γ defines how quickly delayed rewards decrease in value.

We refer to the collection of all the Q-Values (one for each state-action pair) as the Q-Table. The agent's policy at a state s picks an action according to the values in $Q(s, \cdot)$. Specifically, we use an ϵ-greedy strategy: the agent picks the action with the highest Q-value at s with probability $1-\epsilon$ and picks a random action with probability ϵ, where ϵ is a hyper-parameter of the algorithm.

For testing distributed systems, the main technical difficulty lies in defining the reward function. As explained earlier, assigning rewards only to bad states is too sparse. We define rewards in two steps: we provide an *exploration bonus* to the agent if they discover a new state (Algorithm BonusMaxRL) and we use *waypoints* to guide the search (Algorithm WaypointRL). We describe these Q-table based algorithms next.

In both of these algorithms, we rely on backwards updates at the end of an episode to propagate back the updated values faster and achieve higher efficiency in exploration. Assuming states are, in general, not repeated throughout an episode, updating at each step would require several episodes to back propagate an updated value to the initial state. Updating backwards at the end of the episode, instead, allows for an updated reward to be back propagated all the way in a single sweep, thanks to the order of the updates.

4.1.1 BonusMaxRL

Algorithm 2 shows the implementation of the BonusMaxRL exploration policy. The hyper-parameters α, γ and ϵ are given as input. The $pick$ function, which is the way the policy chooses the next action, is the standard ϵ-greedy function. Among the state's available actions, with probability ϵ the policy will return a random action, and with probability $1-\epsilon$ the action with the highest Q-Value. The $processEpisode$ function shows how the policy is updated to maximize the novelty of observed states. Note that no reward is coming from the environment and Q-Values updates are entirely based on the internal exploration bonus of the policy. As explained earlier, BonusMaxRL contains two differences to the updating the Q values against existing approaches.

First, the exploration reward is inversely proportional to the number of visits. Specifically the reward is $\frac{1}{t}$ where t is the number of visits. The visits are recorded by the policy in a table $V(s, a)$ and for every transition (s, a, s'), the visits are updated $V(s, a) = t = V(s, a) + 1$. This reward mechanism rewards new states (with a reward of 1) and diminishes the reward as the number of visits increases.

Second, the update rule of BonusMaxRL is

$$Q(s, a) = (1-\alpha)Q(s, a) + \alpha.max(\frac{1}{t}, \gamma \max_{a'} Q(s', a'))$$

Note the use of max instead of the traditional addition. With this update rule, the Q-Value will now be an estimation based only on the best (less visited) reachable state from that state-action pair. In practice, our algorithm will prioritize a path leading to a new state while ignoring how many times the other states along the path have been visited. When

Algorithm 2: BonusMaxRL: Positive reward based exploration algorithm

Input: α, γ, ϵ

def $init()$: // initialize the Q-Table
  $Q(s,a) \leftarrow 1, V(s,a) \leftarrow 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$

def $newEpisode(\_)$: // reset the trace
  $trace \leftarrow []$

def $pick(s, actions)$: // $\epsilon-greedy$ choice of action
  $x \sim U(0,1)$ // sample a value x uniformly at random (u.a.r.) from (0,1)
  if $x < \epsilon$ **then**
    return $a \sim U\{actions\}$ // return an element from $actions$ chosen u.a.r.
  else
    return $argmax_a Q(s,a)$

def $recordStep(state, action, newState, \_)$:
  $trace \leftarrow append(trace, (state, action, newState))$

def $processEpisode()$: // backward traversal to update Q-Values
  for $i = length(trace) \cdots 1$ **do**
    $(s, a, s') \leftarrow trace[i]$
    $t \leftarrow V(s,a) + 1$
    $V(s,a) \leftarrow t$
    $r \leftarrow \frac{1}{t}$
    if $i < length(trace)$ **then**
      $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot max(r, \gamma \cdot \max_{a'} Q(s', a'))$
    else
      $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot max(r, 0)$

no new states are reachable along a path, the value of its states will converge towards 0 as the number of visits increase.

Our update rule, based on *maximum* instead of sum of immediate and future rewards, would not work to achieve an optimal policy in a traditional RL setting where a reward function is provided. BonusMaxRL would possibly ignore smaller reward signals along trajectories and therefore be unable to learn an optimal policy. For example, consider an environment with a reward function such that two paths lead to the same reward signal. However one of the paths is longer and contains a small additional reward along the way. BonusMaxRL would learn to follow the sub-optimal shorter path despite the other longer path leads to higher total reward. Despite the shortcoming, BonusMaxRL performs better when testing distributed systems. The reason being, the greedy approach of prioritizing new immediate states aligns well with the goal of pure-exploration.

4.1.2 WaypointRL

WaypointRL accepts as input a sequence of predicates $[pred_1, \cdots, pred_n]$ where the last predicate $pred_n$ defines the target space to explore. By allowing the predicates to define the

Algorithm 3: WaypointRL - $init, newEpisode, pick$, and $recordStep$ methods

Input: $predicates = \{pred_1, \cdots, pred_n\}, \alpha, \gamma, \epsilon$

def $init()$**:**
  for $i = 1 \cdots n$ **do** `// init a Q-Table for each predicate`
    $Q_i(s,a) \leftarrow 1, V_i(s,a) \leftarrow 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$

def $newEpisode(initialState)$**:** `// reset values and active predicate`
  $trace \leftarrow [], reached \leftarrow \bot$
  for $i = n \cdots 1$ **do**
    if $predicate_i(initialState) = \top$ **then**
      $activePredicate \leftarrow i$
      $break$

def $pick(s, actions)$**:**
  $x \sim U(0,1)$
  if $x < \epsilon$ **then**
    return *random* $a \sim U(actions)$
  else `// pick greedy w.r.t. current predicate Q-Table`
    $i \leftarrow activePredicate$
    return $argmax_a\, Q_i(s,a)$

def $recordStep(s, a, s', \_)$**:**
  if $reached = \bot$ **then** `// check the new active predicate`
    for $i = length(predicates) \cdots 1$ **do**
      if $predicate_i(s') = \top$ **then**
        $nextActivePredicate \leftarrow i$
        $break$
    if $nextActivePredicate = n$ **then**
      $reached \leftarrow \top$
  else `// target reached, target predicate active for the rest of the episode`
    $nextActivePredicate \leftarrow n$
  $trace \leftarrow append(trace, (s, a, s', activePredicate, nextActivePredicate))$
  $activePredicate \leftarrow nextActivePredicate$

waypoints, WaypointRL enables a semantic guided search of the state space. We set $pred_1$ the starting predicate as true. The core insight behind WaypointRL is to maintain a separate exploration Q-table for each predicate, keep track of the current highest predicate satisfied and use the corresponding Q-table to drive exploration. Updates to the Q-table consists of an additional reward (apart from the exploration bonus) when the exploration satisfies a higher predicate. In other words, driven by the BonusMaxRL exploration algorithm, the agent will learn a policy to reach the target state space and subsequently maximize exploration.

Algorithm 3 and 4 describe the methods that define WaypointRL. At each step we keep track of the highest predicate satisfied in the current states (p - called $activePredicate$ in the Algorithm), use p to pick the action $Q_p(s,a)$, update the highest predicate (to p') based

on the resulting state s' and record the transition (s, a, s', p, p') in the trace. These actions are defined in the $pick$ and $recordStep$ methods.

In the $processEpisode$ method defined in Algorithm 4, we update the Q-tables of the corresponding predicates as follows. First, we check if the trace satisfied the final predicate and record the corresponding step. Then, we iterate backwards in the trace and, for each transition (s, a, s', p, p'), we update the corresponding Q-value $Q_p(s, a)$. The update consists of two rewards, exploration bonus and an additional reward when a higher predicate is satisfied. The update is equivalent to BonusMaxRL when the predicate does not change i.e. $p = p'$ (only exploration bonus). However, when the new predicate p' is higher in the input sequence we define the reward based on whether in that trace we reached the final target predicate or not (recorded by $reachedFinal$ variable). If the trace did not reach ($reachedFinal = \bot$) then the reward is a constant 2. If it reached the final predicate ($reachedFinal = \top$), the reward is $2 + \gamma^d \dot{2}$ where d denotes the proximity to the step where the target was reached. The effect of our reward mechanism is two fold - (1) learning paths to satisfy higher predicates and (2) learning paths to reach the final target space faster. The reward value 2 we use is a hyper parameter that can further be tuned.

Algorithm 4: WaypointRL - $processEpisode$ method

Input: $predicates = \{pred_1, \cdots, pred_n\}, \alpha, \gamma, \epsilon$

def $processEpisode()$**:**

- $reachedFinal \leftarrow \bot, reachedStep \leftarrow 0$
- **for** $i = 1 \cdots length(trace)$ **do** `// check if the episode reached the target predicate`
  - $(s, a, s', p, p') \leftarrow trace[i]$
  - **if** $p = n$ **then** `// if yes, store the step`
    - $reachedFinal \leftarrow \top, reachedStep \leftarrow i$
    - $break$
- **for** $i = length(trace) \cdots 1$ **do** `// backward update for each step`
  - $(s, a, s', p, p') \leftarrow trace[i]$
  - $t \leftarrow V_p(s, a) + 1, V_p(s, a) \leftarrow t$
  - $explR \leftarrow \frac{1}{t}$ `// visits-based bonus`
  - **if** $p = p' \vee p = n$ **then** `// same predicate, update within a single Q-Table`
    - **if** $i < length(trace)$ **then**
      - $Q_p(s, a) \leftarrow (1 - \alpha) \cdot Q_p(s, a) + \alpha \cdot max(explR, \gamma \cdot \max_{a'} Q_p(s', a'))$
    - **else**
      - $Q_p(s, a) \leftarrow (1 - \alpha) \cdot Q_p(s, a) + \alpha \cdot max(explR, 0)$
  - **else** `// sequence transitioned to a different predicate`
    - **if** $p' > p$ **then** $progR \leftarrow 2$ **else** $progR \leftarrow 0$ `// predicate progress bonus`
    - **if** $reachedFinal$ **then** `// final predicate bonus`
      - $d \leftarrow reachedStep - i - 1$
      - $finalR \leftarrow \gamma^d \cdot 2$
    - $Q_p(s, a) \leftarrow (1 - \alpha) \cdot Q_p(s, a) + \alpha \cdot max(explR, \gamma \cdot (progR + finalR))$

A different Q-table for each predicate is initialized in the $init$ method. The $newEpisode$ method resets the trace and the *reached* flag, and checks the active predicate for the initial state. $Pred_1$ is the constant $\top$ predicate. Therefore, when no predicates are satisfied, the algorithm reverts $activePredicate$ to 1. The $pick$ method follows the ϵ-$greedy$ approach on the Q-table of the active predicate. In the $recordStep$ method, the agent updates the active predicate for the next state, eventually setting the *reached* flag to true if it reached the target predicate, and it appends the transition (s, a, s', p, p') to the trace.

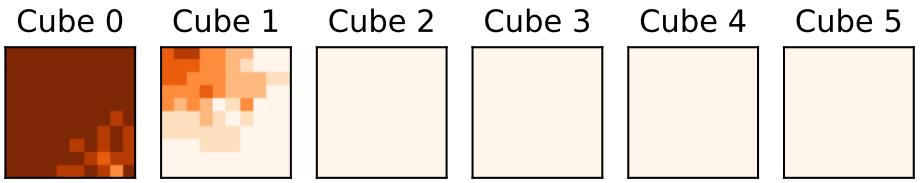
4.1.3 Intuition: Exploring a Cube world

To demonstrate the efficacy of our algorithms, let us consider a "cube world" consisting of a set of 3 dimensional cubes that we want to explore. Each cube in the set is subdivided into a three-dimensional cube of fixed dimension with width (w), breadth (b), and depth (d). The state space is thus a 4-tuple $\mathcal{S} = (g, w, b, d)$ where g defines the cube number in the set. An agent exploring the cubes starts at $(0, 0, 0, 0)$, and at each cell can pick one of the following actions: *up*, *down*, *left*, *right*, *above*, *below*, *into*, and *reset_depth*. The directions will result in moving by one cell, *into* allows to transition through a door, if present, and *reset_depth* brings back the agent to depth zero. The cubes are connected by special cells that act as doors. At doors, the agent can move uni-directionally *into* the next cube by picking the corresponding action. In our example, with 6 $10 \times 10 \times 6$ cubes, we place doors such that $(0, 5, 5, d) \xrightarrow{into} (1, 0, 0, 0)$, $(1, 5, 5, d) \xrightarrow{into} (2, 0, 0, 0)$, and so on, for any depth d of the cubes. The structure and actions of this example aim to reproduce dynamics that can resemble the ones of a system's execution, such as irreversible transitions (doors) or state resets (*reset_depth* action).

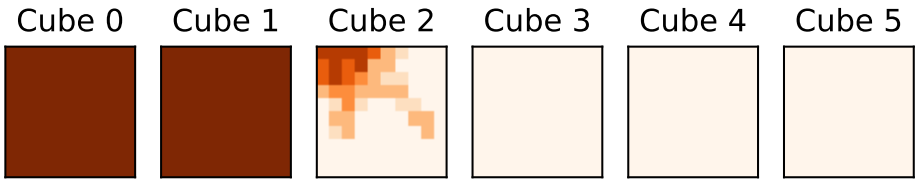
We evaluate different exploration agents to explore the cubes and illustrate the outcome in Figure 4.1. From the starting state, an agent takes a fixed number of steps (horizon) and repeats this process for a given number of episodes. As clear from the Figure, BonusMaxRL covers significantly more cells than the random exploration agent.[1] Additionally, the results show that an RL agent can fail to completely cover cubes that are farther from the initial state.

Let us suppose we are interested only in covering all cells of a specific cube (e.g. cube 3). A trivial solution would be to bias exploration towards cube 3 with an additional bonus reward when reaching the target cube. As shown in Figure 4.1b, the reward would not help, since the agent is never able to reach cube 3 within the given episode budget and hence it would never collect the additional reward. Alternatively, we can improve coverage of cube 3 by introducing an abstraction on the state space. With the abstraction, the state space that RL should cover is smaller and therefore we expect a better coverage of cube 3. An example abstraction would be to ignore the depth co-ordinate, since it is irrelevant to navigate through the cubes (the doors are located along the whole depth of the cubes). Now, RL will explore more cubes but will not systematically explore all depths of each cube. In other words, we lose the granularity of the step. Figure 4.1c shows the cube world coverage of BonusMaxRL using the defined abstraction. It is able to reach and explore part of cube 3, but without covering its entire depth.

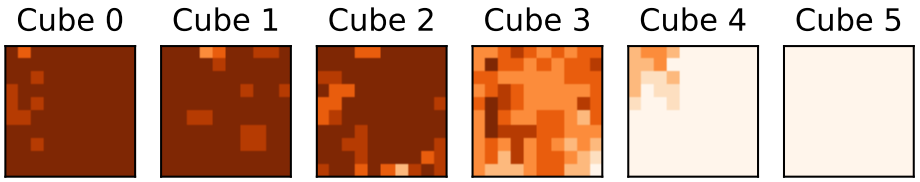
[1] We run the exploration with a horizon of 80 and for 5k episodes. The hyper parameters for BonusMaxRL are $\alpha = 0.3, \gamma = 0.99$



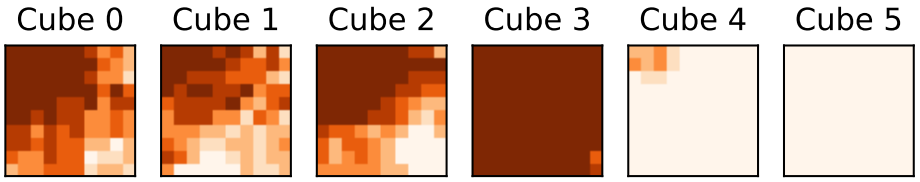
(a) Random exploration



(b) BonusMaxRL exploration

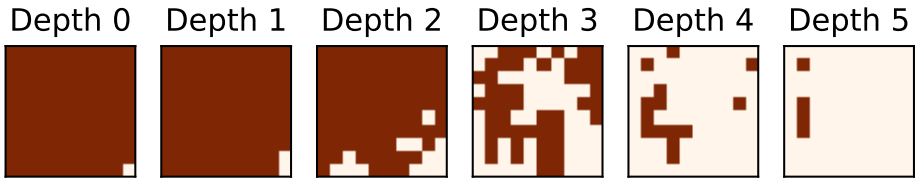


(c) BonusMaxRL exploration with depth abstraction

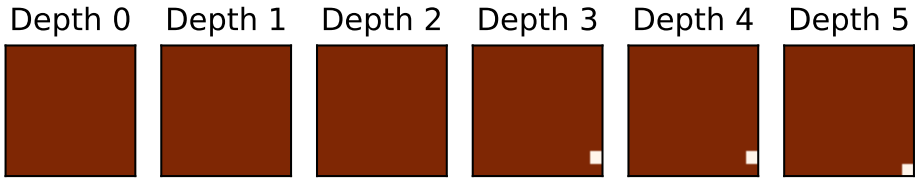


(d) WaypointRL exploration (target cube 3)

Figure 4.1: Exploration of a $6 \times 10 \times 10 \times 6$ cube world, with a given episode budget, using different agents. We plot the heatmap of the top of each cube. The intensity is the sum of the visited cells along the depth of the cube, with the darkest color meaning all the cells have been visited. Here we showcase several points. First, BonusMaxRL (b) achieves better exploration than Random (a), covering more cells. Second, unbiased exploration struggles to reach cubes away from the starting point (b). Third, chosing an appropriate state space abstraction can lead to better coverage, but it can result in reduced capabilities of systematically exploring a target subspace (c), while WaypointRL is able to effectively bias the exploration towards the target cube and almost fully cover it (d).



(a) BonusMaxRL exploration with depth abstraction



(b) WaypointRL cube 3 exploration

Figure 4.2: Detailed Exploration of cube 3 in the $6 \times 10 \times 10 \times 6$ cube world. Each grid represents a depth level of the cube. The colored cells have been explored by the agent. (a) BonusMaxRL using the depth abstraction (b) WaypointRL with reaching cube 1, 2, and 3 as waypoints. WaypointRL is able to explore almost all the cells of the cube.

Within the same budget, our solution WaypointRL achieves the best coverage in cube 3 (Figure 4.1d). In WaypointRL, we split the task into two. First, reaching the target space (cube 3) and second, exploring once the target is reached. We use a different Q table for each task and provide independent rewards. WaypointRL uses the corresponding Q table to pick actions until cube 3 is reached and subsequently performs pure exploration. The first task can be further split into subtasks by providing additional waypoints to guide the agent. WaypointRL generalizes the reward strategy and accepts a set of target state predicates as waypoints. The algorithm associates each waypoint with a specific Q-table that is used to pick actions and update rewards. In our example, we provided reaching cube 1 and 2 as additional waypoints to guide the agent. With WaypointRL, we are able to reach the target space faster, while retaining the granularity of the exploration step. Figure 4.1d shows how WaypointRL avoids exploring the previous cubes and achieves efficient exploration of the target subspace, while Figure 4.2 shows in detail the depths coverage of cube 3 for BonusMaxRL with abstraction and WaypointRL. [2]

As clear from the cube example, RL based agents effectively explore states and also are able to bias exploration. The state space of a distributed program is analogous to the cubes. Distributed protocols have communicating processes that reach different states based on different message inter-leavings. Additionally, when there is a quorum of messages or when a timeout occurs, the processes make irreversible progress (akin to moving through a door). In Section 4.5 we will replicate similar results for 3 different distributed protocol implementations.

[2]We run the exploration with a horizon of 80 and for 5k episodes. The hyper parameters for WaypointRL are $\alpha = 0.3, \gamma = 0.99$

4.2 Distributed program transition system

To use RL for exploring the state space of a distributed program, we need to define an *Environment* that encapsulates the distributed program. Specifically, we should define the *step, reset* and *actions* functions. The functions *step* and *reset* allow navigating through the state space of a distributed system (transition functions). In this section, we first define the concrete state and actions followed by the transition functions. Subsequently, we provide general modelling guidelines for using RL on distributed systems.

We choose our actions to represent the network configurations. The reason for our choice is due to the limitations of testing distributed system implementations — we only control the network between the processes. specifically, we restrict the set of possible network configurations to partitions between processes.

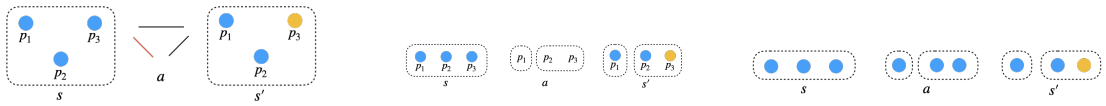
The state of the distributed system contains two components - an abstraction over local states of processes and the partition configuration. Each abstract local state is identified by a color that excludes process identifiers. An example state is $\{\{c_1\}, \{c_1, c_2\}\}$ where c_1, c_2 are abstract local states and one of the proceses with abstract state c_1 is isolated from the rest.

The state and actions we define are similar to the product transitions system of NETRIX (Section 3.2.1). In effect, we redefine an *RLMonitor* that complements the NETRIX *monitor*. However, unlike with NETRIX, *RLMonitor* is not driven by the events of the processes but by explicit actions that the RL agent invokes. Additionally, *RLMonitor* needs access to the state of the processes, which is derived from the events in NETRIX. To overcome the differences, we reimplement the core infrastructure to run RL exploration (elaborated in Section 4.4).

4.2.1 Defining the transition system

Formally the state of the distributed system $s \in \mathcal{S}$ is a multi set of multi set of colors $c \in \mathcal{C}$. For example, consider the state $\{\{c_1\}, \{c_1, c_2\}\}$. Our definition of a state succinctly represents the two components - local state of processes and the network state. The colors are abstractions over local state defined by a coloring function $\mathbb{C} : LS \rightarrow \mathcal{C}$. For the network state, each set defines the partitions. In the example, the first process with color c_1 is isolated from the remaining processes.

The set of actions is the set of all possible partition configurations given the current coloring of local states. For example from the state $\{\{c_1\}, \{c_1, c_2\}\}$, the set of actions are $\{\{c_1, c_1\}, \{c_2\}\}, \{\{c_1, c_1, c_2\}\}, \cdots$. By picking an action, the exploration algorithm determines the new partition configuration in the resulting state. For example, with the action $\{\{c_1, c_1\}, \{c_2\}\}$, the two processes with current abstract state c_1 are grouped in the same partition and the third process c_2 is isolated from them. However in the resulting state, the colors of the processes may be different. Based on the action, we deliver messages allowed by the partition configuration and drop the remaining. The new messages received (or not), determines the new colors of the processes. More formally, we define an action as follows. Given a state s with cardinalities of colors $C_s : \mathcal{C} \rightarrow \mathbb{N}$, the actions enabled in s are all possible multi-sets of multi-sets of colors where the cardinality of each color $c \in \mathcal{C}$



(a) (b) (c)

Figure 4.3: Evolution of the transition system of a distributed program. (a) Fine state space. State is a map of local states and action is the set of messages to deliver and drop. (b) Coarse state space without symmetry reduction. (c) Coarse state space with symmetry reduction

is exactly $C_s(c)$. Then, a transition from s with some action can lead to a state s' where the cardinality of colors $C_{s'}$ is arbitrarily different from C_s.

Let us imagine a few simpler transition system models and use the drawbacks to motivate our final transition system. The first model is illustrated in Figure 4.3a. The global state (3 blue dots with process identities) contains just a map of the complete local state and the network configuration is stored as a set of active network links. The actions pick the next set of active network links (identified by non red arrows between processes). The resulting state (with process p_3 in a different state) is obtained by exchanging messages only along the active links. The granularity of actions is very fine with this model and therefore RL needs more steps and episodes to learn to reach new states. For example, reaching a new state with a quorum of messages requires learning to keep enough links active.

The second model (Figure 4.3b) introduces higher order actions where only partitions between processes are the valid network configurations. Each action corresponds to selecting a partition between processes (depicted using boxes around processes). The resulting state is obtained by exchanging only those messages within the same partition defined by the action. Such a model considers process identities when comparing two different states. Distributed protocols however define transitions only based on the state of the process and not the identities. Therefore, the drawback of the second model is to count redundant states where processes's local states are swapped around. We overcome the drawback by ignoring process identities and only considering an abstract local state (as colors) — leading to our original definition of a state (Figure 4.3c). An action re-defines the partition of colors in the pre-state (isolating the left-most process from the rest), but the processes receiving messages allowed by the new partition may result in colors changing in the post-state (e.g., the right-most process).

4.2.2 General modelling guidelines

RL exploration based algorithms perform best when the transitions in the state space are predictable. Specifically, the non-determinism in the transitions follow an underlying probability distribution that can be learnt. Therefore, we allow RL to perform better by reducing the degree of non-determinism in the transitions. The principle of making systems "more" deterministic however is also common with other testing approaches. With a more deterministic system, any bugs found can be easily replayed.

In this subsection, we will use the Raft protocol as an example to emphasize the guide-

lines. To recall briefly, the protocol dictates the local state components of each process - (1) the term number, (2) the state of the process $\{Candidate, Leader, Follower\}$ and (3) the process which it has voted for. In industrial implementations of Raft however, process states contains lot more information. The color abstraction we use in our model picks only specific components of the local state to be included in the abstraction.

In distributed systems, time is the major contributor for non-determinism. The next local state of a process depends on an internal clock which is usually not part of the state. To remove this non-determinism, we fix the duration of time that passes between two transition steps in our transition system.

Another source of non-determinism is the color abstraction. If the color abstraction excludes crucial state information, the transitions become unpredictable and therefore decrease the performance of RL based exploration algorithms. Consider the Raft protocol. If we do not include the role of each process in the color, RL will not be able to differentiate when a leader is elected or not. Furthermore, it will not able to learn to make progress and commit entries without observing that a leader is necessary to do so. On the contrary, if we include too much information in the color abstraction, we will explode the state space that needs to be covered and also achieve sub-optimal outcomes. With redundant information in the color abstraction, we will reverse the effect of the symmetry reduction. For example, we should not include the process identifier in the color abstraction.

When modelling, a general principle that we rely on is that the state space should be bounded. In other words, the set of possible colors should be bounded. For example, consider the raft protocol. To recall, each process consists of a term number as part of its local state. By including the term number in the color abstraction, we enable an infinite set of colors. Furthermore, we introduce redundancy - consider two global states where the only difference is an offset in the term number of each local states, only one of them is *interesting* as the set of possible next states are the same modulo the offset. Therefore, the color abstraction should bound the term number of each local state. In general, it is important to balance the trade off when defining the color abstraction. Too much information explodes the state space and too little information introduces unpredictability.

Protocol implementations contain more information as part of the local state than what is defined in a model of the protocol. The additional information corresponds to optimizations introduced in the implementation. For example, most distributed protocol implementations introduce snapshots. When defining the color abstraction, the developer has to ensure that relevant parts of the additional state are also captured. As we will describe in Section 4.3, capturing the additional information enables the developer to bias exploration and test the parts of the code related to the optimizations.

4.2.3 Environment parameters

Our final transition system used in the implementation of our approach contains a few more optimizations.

1. We introduce failure actions to crash and start processes. The color abstraction contains an additional parameter to depict if the process is crashed or active.

2. We introduce a finite number of actions where RL can inject new requests to the system. With new requests, we are able to explore more states that wouldn't be possible otherwise. For example with raft, the commit index increases only when new requests are committed.
3. We introduce a parameter *ticks* to control the time duration between two states. The number of ticks is tied to the timeout parameters of the protocols. If too much time passes between two states, then processes always timeout and if too little time passes then processes never timeout. The parameter allows the test developer to control the tradeoff and explore more states.
4. We introduce a bounded counter *SameState* in the global state representation, which increments, up to its bound, if the state (colors configuration) did not change after keeping the same partition. This incentivizes RL to explore the same state up to the counter limit, considering them different states. Setting a short time duration between states enables fine-grained interleavings of different partitions, potentially leading to new states. On the other hand, the protocol might require multiple steps in the same partitions configuration to progress. The counter allows to set a short time duration while enabling RL to explicitly explore scenarios of partitions stability.

In total, to test an implementation, the developer has to specify the following parameters. We will refer to these parameters in Section 4.5 when we list the concrete values used for the different benchmarks.

1. **The color abstraction**. The components of the state that define the color of each process.
2. **Number of processes**. The number of processes in the system.
3. **Ticks**. The *ticks* parameter explained above.
4. **Max Same State**. The maximum value for the *SameState* counter.
5. **Max Crash Actions**. The maximum number of crash actions allowed in an episode.
6. **Max Concurrent Crashes**. The maximum number of processes that can be crashed at the same time.

4.3 Predicate sequences

Unlike with the cube world, the distributed system state space is significantly larger and harder to visualize. Pure exploration is insufficient to cover all possible states. Therefore, the need to bias exploration using WaypointRL is all the more relevant. To bias exploration in distributed systems, we will use state predicates to define both the target state space and the intermediate rewards. In the Raft protocol, an example predicate would be - 'there exists a leader'. A predicate captures a set of executions scenarios. If we strengthen the predicate - 'there exists a leader in term 3' - we will constrain the set of admitted scenarios.

A sequence of state predicates specified to the WaypointRL are similar to the filter *conditions* of Netrix. While with Netrix, the filters are enforced in every execution, the state predicates merely act as guidance posts to be used by WaypointRL. Semantically, the predicate sequences are used to provide rewards and do not directly alter the set of actions

possible from a given state. Syntactically, the state predicates are equivalent to the *conditions* of the NETRIX filters.

While both NETRIX filters and WAYPOINTRL predicate sequences allows the developer to describe scenarios, they differ in terms of effort. WAYPOINTRL predicate sequences are more high level and easier for the developer to describe. In contrast, NETRIX filters are low level and requires the developer to carefully construct the set of events that describe the scenario.

Overall, as mentioned in the Introduction, biasing exploration towards specific scenarios achieves two purposes. The developer gains confidence in the code when there are no bugs found and if there are bugs found, biased exploration will reliably reproduce the bugs. We rely on the developers understanding of the protocol to provide target predicates. In this section, we list example predicates for Raft and provide general guidelines to derive predicates for distributed protocols.

4.3.1 Deriving target predicates

As clear from the example, our main source of target predicates is the abstract protocol specification. The developer's understanding of the protocol specification is sufficient to construct scenarios and use them to bias exploration. However, a developer will have to instantiate the predicates by assessing the data structures used in the implementation.

Hierarchy Name	Description	Intermediate Predicates
AllCommitted(x)	At least x committed entries in the log of all processes	AllCommitted(x - i)
ProcessesInTerm(n, t)	At least n processes are in term t	ProcessesInTerm(n, t-1)
CommittedEntriesInTerm(x, t)	At least x committed entries in the logof a process in term t	CommittedEntriesInTerm(x - i, t) LeaderInTerm(t)
LeaderInTerm(t)	A process is in state 'leader' in term t	ProcessesInTerm(n, t)
LogDiff(x)	A gap of x entries between any two processes logs	LogDiff(x - i)
LogCommitDiff(x)	A gap of x entries between any two processes committed logs	LogDiff(x - i) LogCommitDiff(x - i)
ProcessInRole(r)	Any process in role r	-
ProcessInRoleTerm(r,t)	A process in role r and in term t	ProcessesInTerm(1, t)

Table 4.1: Generic predicates for Raft

In general, a scenario of a distributed protocol contains segments of two kinds. One where processes are in sync and one where processes are out of sync. Using this insight, we derive three classes of predicates - ones which describe processes in sync, ones which describe processes out of sync, a combination of the two. We will refer to the examples of Raft protocol listed in Table 4.1. Progress occurs when processes are in sync - by committing entries (CommittedEntries(2)), having a stable leader (LeaderInTerm(2)). However, progress stalls when processes are out of sync - processes in different terms (ProcessesInTerm(1,2)

and ProcessesInTerm(1,4)), difference in logs (LogDiff(2)). The interesting scenarios are the ones where we combine the two. For example, a leader in a higher term (ProcessInRoleTerm(leader,3) and ProcessesInTerm(1,1)), difference in committed entries (LogCommitDiff(2)). Note that by *sync* or *out of sync*, we are referring to the abstract notion captured in the states of each process and not the concrete network state.

Note that our list of predicates is not exhaustive and the developer is unconstrained while listing target predicates. Apart from the protocol, the developer can derive scenarios based on implementation specific optimizations. As mentioned in Section 4.2.2, the implementation introduces some specific optimizations to the protocol such as snapshots and recovery. An example scenario would be to force a snapshot - 'process with snapshot index 2'. Note that these predicates can be combined with those derived from the protocol.

4.3.2 Specifying intermediate predicates

When biasing exploration using a target predicate, the ability of WaypointRL to effectively bias depends on how many scenarios the target predicate captures. As we will show in our evaluation, when the predicates are easy to satisfy WaypointRL out performs other approaches with just one target predicate. However, a more constrained predicate is harder to bias towards. Therefore, WaypointRL accepts additional intermediate predicates as waypoints ($p_2, \cdots p_{n-1}$ in Algorithm 3) to improve the effectiveness of biasing exploration. The intermediate predicates are used to split the task of reaching the target predicate to provide intermediate rewards to RL. For those target predicates with intermediate waypoints, we will empirically show that providing more intermediate predicates improves the effectiveness of biasing exploration. The question now arises on how to derive the intermediate predicates given a set of target predicates.

In Table 4.1, we also list candidates for intermediate predicates. In general, predicates that are true on every execution path towards the target space are good candidates. For example, consider ProcessesInTerm(1, 3) where we require a process in term 3. To achieve the target, we need ProcessesInTerm(1, 2) to be true first which serves as an intermediate predicate.

4.4 Implementing BonusMaxRL and WaypointRL

We implement the two algorithms in go [3] programming language (open-sourced [4]).

In the process, we reimplement the core infrastructure to capture in-transit messages and read the state of the nodes. While Netrix iterations are driven by the events occurring in the nodes, BonusMaxRL and WaypointRL take steps at a fixed frequency independent of the events in the nodes. Furthermore, BonusMaxRL and WaypointRL require direct access to the states of the nodes which, in the case of Netrix, was inferred from the events observed. The re-implementation derives insights from Netrix. Specifically, we reuse the instrumentation flow that includes the client libraries to capture messages, and the endpoints that receive messages.

[3] https://go.dev

[4] https://github.com/zeu5/dist-rl-testing

Snippet 4.1: Partitioned environment interface definitions

```
type PState interface {
    NodeState(int) NState
    Messages() []Message
    Requests() []Request
    CanDeliverRequest() bool
}

type PartitionedEnvironment interface {
    Reset() (PState, error)
    Tick(*StepContext) (PState, error)
    DeliverMessages([]Message, *StepContext) (PState, error)
    DropMessages([]Message, *StepContext) (PState, error)
    ReceiveRequest(Request, *StepContext) (PState, error)
    StartNode(int, *StepContext) (PState, error)
    StopNode(int, *StepContext) (PState, error)
}
```

Our implementation introduces a generic interfaces for an RL environment (**Environment**), agent and policy (**Policy**). We extend the interfaces into concrete implementations as follows,

- **PartitionedEnvironment** that captures the transition system described in Section 4.2 and extends **Environment**
- **BonusMaxRLPolicy** implements BonusMaxRL and extends the generic **Policy**
- **HierarchyPolicy** implements WaypointRL and also extends the generic **Policy**

In addition to these and for the purpose of comparison, we implement a **RandomPolicy** that chooses actions uniformly at random.

Partitioned Environment

While the policies are agnostic to the concrete protocol implementation we will be testing, the environment is not. The PartitionedEnvironment class is abstract and requires hooks to fill in the details of a concrete implementation. The hooks allows PartitionedEnvironment to construct the state (including colors, etc) and perform actions (deliver messages, start, stop, etc).

Snippet 4.1 lists the functions that represent the hooks needed to test an implementation. The functions described along with the intended functionality are as follows,

- Reset - to reset all the nodes to their initial state
- Tick - to let one unit of time pass. In the process, any new messages intercepted are recorded into the new state. (Can be logical time or system time)
- DeliverMessages - to deliver the messages to the respective messages
- DropMessages - to update the state by removing the respective messages

- ReceiveRequest - to deliver a client request
- StartNode - to start a proceses
- StopNode - to stop a process

Note that all functions return a state PState that contains all the necessary information to construct the state presented to RL. The details of how PState state information is captured is abstracted to allow for customization corresponding to each protocol implementation. For example, the endpoints to read the state of a node is specific to each implementation.

4.5 Evaluation

We evaluate the two algorithms BonusMaxRL and WaypointRL on 3 benchmarks - RSL, Etcd, RedisRaft. In all our experiments, we compare against two baseline approaches - Random and NegRL [Muk+20]. We implement[5] the algorithms in the Go programming language. In addition to the algorithms, our codebase consists of thin shims around the implementations to enable testing. The shim allows RL to read the state and execute the actions chosen at each step.

Our Benchmarks are (1) RSL - a re-implementation of Azure RSL [6]. The algorithm is a variant of Paxos and powers distributed services in the Azure cloud. (2) Etcd [7] - an implementation of the Raft protocol that powers a popular distributed key value store. (3) RedisRaft [8] a distributed version of the popular in-memory key value store that uses the Raft protocol. We re-implement RSL to enable easy instrumentation and testing on a common Linux platform. The original implementation is built to run on Windows systems.

Our baseline approaches are pure Random exploration which picks the next actions uniformly at random and NegRL which relies on a negative reward. We implement the NegRL policy in our system. It performs Q-values updates with negative rewards at each step and uses the *softmax* function to pick actions. The original description of the reward is a constant value of -1. However, the authors of NegRL describe NegRLVisits, an alternative version which proves to be empirically better. We compare against NegRLVisits where the reward for a step that reaches a state s is $-visits$ to state s.

With our evaluation, we aim to answer the following research questions.

RQ1 Are theoretically optimal algorithms efficient in practice?
RQ2 Can we achieve better coverage than existing approaches with BonusMaxRL?
RQ3 Can we bias exploration towards a target state space with WaypointRL?
RQ4 Does RL based exploration approaches help uncover bugs?

For **RQ1**, we show that theoretical optimal algorithm UCBZero fails to achieve good coverage in practice. For **RQ2**, we find that NegRLVisits achieves better coverage than BonusMaxRL in RedisRaft and Etcd benchmarks. However, BonusMaxRL performs better than NegRLVisits in the RSL benchmark. The answer to **RQ3** is Yes WaypointRL explores more

[5] https://github.com/zeu5/dist-rl-testing
[6] https://github.com/Azure/RSL
[7] https://github.com/etcd-io/raft
[8] https://github.com/RedisLabs/redisraft

unique states in the target space than pure exploration approaches for 20 out of 26 different target predicates. Furthermore, when the target states are harder to reach, we show that we can improve the coverage by adding more intermediate predicates. For **RQ4**, we find 13 bugs with WaypointRL as opposed to 11 with Random, 10 with BonusMaxRL and 7 with NegRLVisits. Additionally, for 11 of these bugs, we are able to replicate the bugs with higher average frequency using WaypointRL than other approaches. Furthermore, a developer gains confidence even when testing with WaypointRL does not lead to any new bugs. The rest of the section is as follows - We first describe the coverage metric and the test harness parameters. Then, we present our evaluation for the three research questions.

4.5.1 Test setup

In our experimental results, we present comparison between different approaches using a coverage metric. The coverage metric measures the number of unique abstract states observed in each of the benchmarks. Specifically, the abstract state we measure is a multi-set of colors $s \subseteq (\mathcal{C} \times \mathbb{N})^n$ where colors are abstract local states of each process. While the concrete abstraction of local states differs between the different benchmarks, we follow the same principles when abstracting. Namely, the color abstraction includes

1. A round number (term, ballot, round, etc.)
2. The role (leader, proposer)
3. The log of requests
4. A commit index (commit, number of decided entries)
5. Current vote or leader

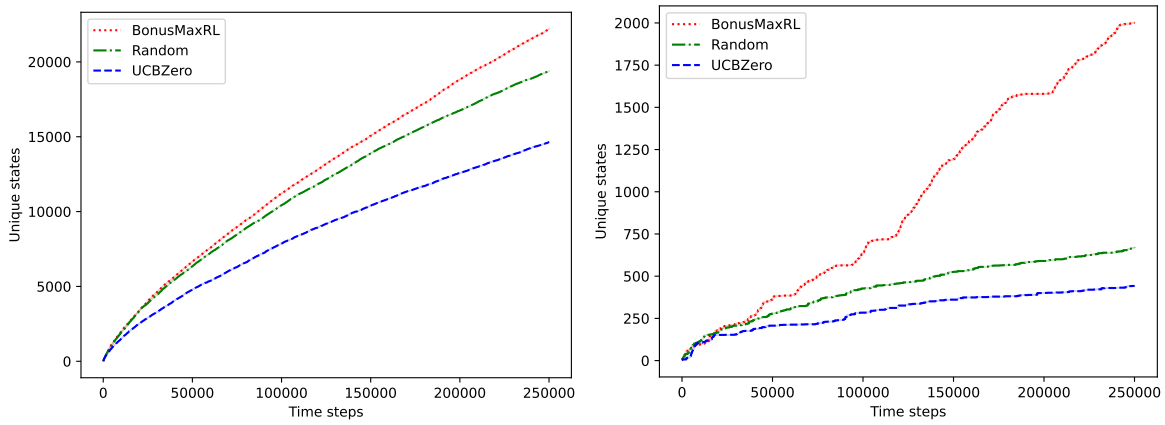
To run experiments, we need to tune two sets of parameters. Ones related to the environment and ones related to the exploration algorithm. Here we list the concrete values used for both sets. The values for the environment parameters described in Section 4.2 are as follows,

1. Number of nodes in the system - 3
2. Ticks between steps, controls the time duration passed when executing an action on the system - 4 units
3. *SameState* counter limit - 5
4. Maximum number of crash actions in an episode - 3
5. Maximum number of nodes to be crashed at the same time - 1

The parameter values for the RL based polices described in Section 4.1 are as follows (the values for NegRLVisits are chosen according to the recommendations of the authors),

1. The learning rate α is 0.2 for both BonusMaxRL and WaypointRL, and 0.3 for NegRLVisits.
2. The discount factor γ is 0.95 for both BonusMaxRL and WaypointRL, and 0.7 for NegRLVisits.
3. The ϵ-greedy values for both BonusMaxRL and WaypointRL is 0.05

Our experiments are run for 10000 episodes or 8h whichever occurs first. Each episode has an horizon of 25 in Etcd and RSL (max 250k total steps) and 50 in RedisRaft (max 500k



(a) Etcd coverage

(b) RSL coverage

Figure 4.4: The pure coverage comparison between UCBZero, BonusMaxRL and Random for the two benchmarks. Each plot contains the coverage vs time steps

total time steps). Horizon is set such that we are able to reach all possible target states and the difference in horizon is due to the different granularity of steps supported by the implementation. However, some episodes might terminate without running for the entire horizon due to a failure in the underlying system. Therefore, when comparing different approaches, we plot the coverage metric against the number of time steps passed instead of number of episodes. We conduct at least 10 trials for each benchmark and report the average numbers. To claim statistical significance in higher coverage, we perform the standard statistical test - Mann-Whitney U test with a threshold of > 0.05.

4.5.2 RQ1: Are theoretically optimal algorithms efficient in practice?

The problem of state space exploration without explicit rewards was first introduced in [Jin+20] and later work provides theoretical optimal algorithms [ZMS20]. In the reward free exploration setting, the goal of the RL agent is to achieve high coverage of the state space. In each iteration/episode, the RL agent takes H steps (horizon) to traverse through a state space of size S with at most A actions possible at each step.

One solution to the problem is UCBZero [ZMS20] which achieves an ϵ-optimal policy to explore after running the RL algorithm for $O(H^5SA/\epsilon^2)$ iterations. Here ϵ denotes the distance from the true optimal policy. Similar to BonusMaxRL, UCBZero relies on a decaying reward function to achieve the optimal policy - an idea we borrow for the design of BonusMaxRL.

In practice, however, the algorithm fails to perform and cover the state space due to the large number of iterations required (5th degree polynomial in the number of steps) to achieve an optimal policy. Even for a small number of steps (horizon) of 25/iteration, the sample complexity of UCBZero is in the order of millions. Figure 4.4 illustrates the coverage of UCBZero in comparison to pure random exploration and BonusMaxRL for two bench-

Benchmark	Random	BonusMaxRL	NegRL
RedisRaft	27081.9 ± 2627.31	32818.4 ± 3017.32	33433.2 ± 2972.34
Etcd	19179.3 ± 106.26	22202.7 ± 137.75	**24898.6 ± 97.40**
RSL	678.9 ± 36.43	**2020.9 ± 190.67**	751.2 ± 37.24

Table 4.2: Final average coverage values for the different benchmarks.

marks (Etcd [9] and RSL [10]). Given the fixed number of iterations (10000), UCBZero performs worse than random.

The reason for low performance in a practical setting is because UCBZero performs exploration similar to a depth-first search. For a short number of iterations, this severely restricts the number of new states that it observes. From an initial random trace, UCBZero repeats the trace several times until the exploration bonus diminishes sufficiently (a state has been visited several times). Only then, UCBZero explores a different trace.

The other drawback of using UCBZero for exploring the state of a distributed systems is the unavailability of the required parameters. It is unclear how we can bound the size of the state space S and the number of actions A for a distributed system.

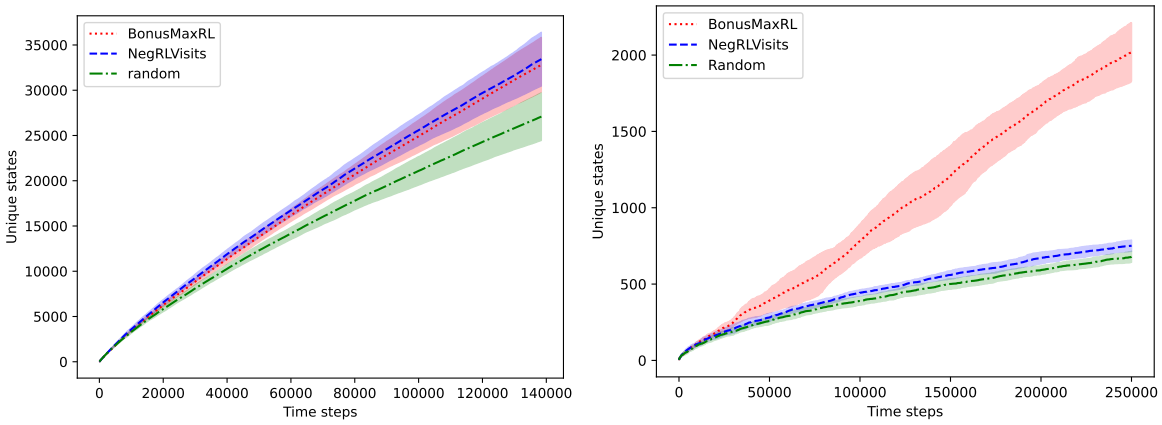
Therefore, we borrow the intuitions behind the reward functions to develop new algorithms BonusMaxRL and WaypointRL to achieve high coverage in practical settings.

4.5.3 RQ2: Can we achieve better coverage with BonusMaxRL?

We observe that RL based approaches achieve significantly better coverage than random exploration. Between the two RL approaches, BonusMaxRL achieves better coverage than NegRLVisits approaches in the RSL benchmark and NegRLVisits has better coverage in the remaining 2. We present in Figure 4.5 the coverage of the different exploration algorithms for RedisRaft and RSL benchmarks. The results of the Etcd benchmark are similar to RedisRaft and we refer the reader to Appendix for details. We believe that the high negative reward in NegRLVisits provides a very strong incentive for pure exploration and hence the better overall coverage. Table 4.2 contains the final average coverage numbers of the three approaches for the different benchmarks. Overall, BonusMaxRL covers 21%, 15.7%, 197% more states than Random in the RedisRaft, Etcd and RSL benchmarks respectively and 169% more than NegRLVisits in the RSL benchmark. NegRLVisits covers 12.1% more states than BonusMaxRL in Etcd benchmark. Both NegRLVisits and BonusMaxRL achieve similar coverage in RedisRaft benchmark.

While NegRLVisits is able to achieve higher coverage than BonusMaxRL (except in the RSL benchmark), it is unclear if the reward mechanism can be adopted to bias exploration. Our reward mechanism to bias relies on providing a constant positive reward. NegRLVisits does not limit the Q-values within a fixed range, they will get more and more negative with the increasing number of visits. In this scenario, it would be difficult to control the tradeoff between reward exploitation and further exploration. A low constant reward value could

[9] https://github.com/etcd-io/raft
[10] https://github.com/Azure/RSL



(a) RedisRaft

(b) RSL

Figure 4.5: The pure coverage comparison between NegRLVisits, BonusMaxRL and Random for the different benchmarks. Each plot contains the average coverage vs time steps

be unable to balance the negative rewards and hence fail to learn, while a high value might immediately drive the policy to converge to that path, giving up further exploration and optimization. Therefore, WaypointRL is based on the principles used in BonusMaxRL.

4.5.4 RQ3: Can we bias exploration towards a target state space with WaypointRL?

Yes, given a sequence of predicates, we are able to bias exploration to cover more states in the target coverage with WaypointRL. As motivated in Section 4.3, we use a total of 26 target predicates for all the benchmarks together in order to bias exploration. Table 4.3 describes the set of target predicates that we use to bias for the two Raft benchmarks and RSL benchmark. As mentioned in Section 4.3, the predicates belong to classes which require processes to be in sync, out of sync or a combination of both.

For each target predicate, we measure the number of unique states observed that appear in the episode after the final target predicate has been satisfied. Table 4.4 compares the target coverage of WaypointRL, BonusMaxRL, NegRLVisits and Random exploration. For 20 out of 26 predicates, we observe that the biased exploration WaypointRL guided by the predicate achieves significantly more states than all the unbiased approaches - BonusMaxRL, NegRLVisits and Random. Furthermore, the difference in the biasing is more stark for those target predicates where the number of permitted scenarios is low. For example, consider the predicate LogCommitDiff3 in RedisRaft which requires that there are two processes whose commit indices differ by 3. WaypointRL on average covers 97x, 94x, 147x more states than BonusMaxRL, NegRL and Random respectively. For the predicates for which we did not achieve improved coverage, we can try to speculate about the reasons. A possible reason is that a predicate might be too easy to reach for the unbiased baselines. In these cases, the impact of learning a policy to reach the predicate is reduced, while, on the other hand, it can reduce the variability of the explored executions. This could be the case for a predicate as OneInTerm3, which can easily happen during any execution. Other pred-

Target Predicate	Description
Raft (RedisRaft, Etcd)	
OneInTerm(3)	At least one process in term 3
AllInTerm(2)	All the processes simultaneously in term 2
TermDiff(2)	A difference of 2 terms between any two processes
CommitEntries(2)	At least 2 committed entries in the log of a process
EntryInTerm(2)	At least 1 committed entries in the log of a process in term 2
LeaderInTerm(2)	A process is in state 'leader' in term 2
LogDiff(1)	A gap of 1 entry between any two processes logs
LogCommitDiff(3)	A gap of 3 entries between any two processes committed logs
OneLeaderOneCandidate	A process in state "leader" and another in state "candidate"
RSL	
AllBallot(3)	All processes reach ballot (round) 3
AnyBallot(3)	Any process in ballot 3
AnyDecided(3)	Any process decides 3 entries
AnyDecree(2)	Any process reaches decree 2
BallotDiff(2)	A difference of 2 ballots between two active processes
EntryBallot(2)	An entry in the log in ballot 2
PrimaryBallot(2)	A process becomes primary (leader) in ballot 2
DecidedDiff(3)	A difference of 3 decided log entries between any two processes

Table 4.3: Target predicate descriptions used to bias exploration in Raft and RSL benchmarks

icates might be simply too hard to reach. In such cases, specific knowledge of the protocol implementation details could help with designing better intermediate predicates.

For each predicate sequence, we list the values for the non parametric Mann Whitney U statistical test. The test determines if two samples are from different statistical distributions. The test outputs a value which if > 0.05 then we reject the hypothesis. We report the test values in Table 4.5 for all the target predicates where we compare WaypointRL vs other pure exploration approaches.

Sensitivity to intermediate rewards In Section 4.3, we describe a methodology to list intermediate predicates to bias the exploration along with example predicates for Raft (Table 4.1). The intermediate predicates help WaypointRL to reach the target space faster and improve the accuracy especially when the target space described by a predicate is hard to reach. We list in Table 4.6 the target coverage with increasing number of intermediate predicates for three target predicates. The result is also visualized in Figure 4.6 for two of the target predicates. Let us consider the example of LogCommitDiff(3), where we use LogDiff(1), LogDiff(2), and LogDiff(3) as intermediate predicates. We observe that the biasing without intermediate predicates fails to improve coverage over the unbiased baselines in the given time budget. On the other hand, when using respectively 1 (LogDiff(1)) and all the 3 intermediate predicates, the final coverage is significantly better, showing that intermediate predicates can play a key role in the biased exploration efficiency and performance. Interestingly, by providing LogDiff(1) as the target predicate, we still achieve improved coverage over the unbiased baselines, showing that even partial biasing can po-

Benchmarks	No.Pred	PredHRL	BonusMaxRL	NegRLVisits	Random
RedisRaft					
OneInTerm(3)	1	15934 ± 3040	16640 ± 2033	17214 ± 2103	14370 ± 1810
AllInTerm(2)	1	**23695 ± 3810**	7660 ± 945	8437 ± 876	6401 ± 794
TermDiff(2)	1	23478 ± 2917	22214 ± 1973	23323 ± 2149	20112 ± 2007
CommitEntries(2)	1	**24656 ± 3995**	4894 ± 552	5325 ± 741	2835 ± 530
EntryInTerm(2)	3	**22758 ± 5457**	8267 ± 272	10114 ± 418	9812 ± 662
LeaderInTerm(2)	1	**22533 ± 5418**	8971 ± 299	10834 ± 410	10361 ± 684
LogDiff(1)	1	**30779 ± 3365**	5755 ± 713	5958 ± 825	3332 ± 606
LogCommitDiff(3)	2	**14960 ± 4100**	154 ± 63	158 ± 52	102 ± 41
OneLeaderOneCandidate	3	1301 ± 1360	482 ± 120	471 ± 158	356 ± 93
Etcd					
LogCommitGap(3)	4	**13336 ± 664**	5692 ± 132	6411 ± 112	4717 ± 92
OneInTerm(4)	3	**33011 ± 826**	29545 ± 258	29739 ± 167	25309 ± 200
MinCommit(2)	3	**29862 ± 885**	25015 ± 193	24031 ± 110	21765 ± 195
TermDiff(2)	1	**15289 ± 885**	4673 ± 162	7792 ± 66	4879 ± 142
LeaderInTerm(4)	3	**15727 ± 1157**	10684 ± 142	11171 ± 215	9571 ± 101
AtLeastOneCommitInTerm(2)	3	**32709 ± 1025**	28169 ± 262	25863 ± 176	23903 ± 314
OneLeaderOneCandidate	3	35403 ± 958	36178 ± 142	32891 ± 208	29021 ± 307
LogGap(2)	2	**37832 ± 3485**	31372 ± 314	33040 ± 115	27445 ± 238
AllInTerm(5)	3	**10346 ± 1121**	8202 ± 65	7888 ± 122	6682 ± 94
RSL					
AnyBallot(3)	1	**1573 ± 174**	837 ± 75	301 ± 34	264 ± 31
AllBallot(3)	1	**1021 ± 57**	493 ± 92	102 ± 14	101 ± 16
EntryBallot(2)	1	1016 ± 460	**1954 ± 131**	698 ± 27	658 ± 48
AnyDecree(2)	1	**1068 ± 119**	663 ± 52	188 ± 19	155 ± 23
BallotDiff(2)	1	19 ± 5	12 ± 6	2 ± 2	3 ± 3
AnyDecided(3)	1	**856 ± 93**	492 ± 46	134 ± 18	110 ± 16
PrimaryInBallot(2)	2	**607 ± 66**	467 ± 54	232 ± 22	196 ± 30
DecidedDiff(3)	3	**113.7 ± 46.8**	22.7 ± 10.1	5.7 ± 3.1	2.9 ± 1.5

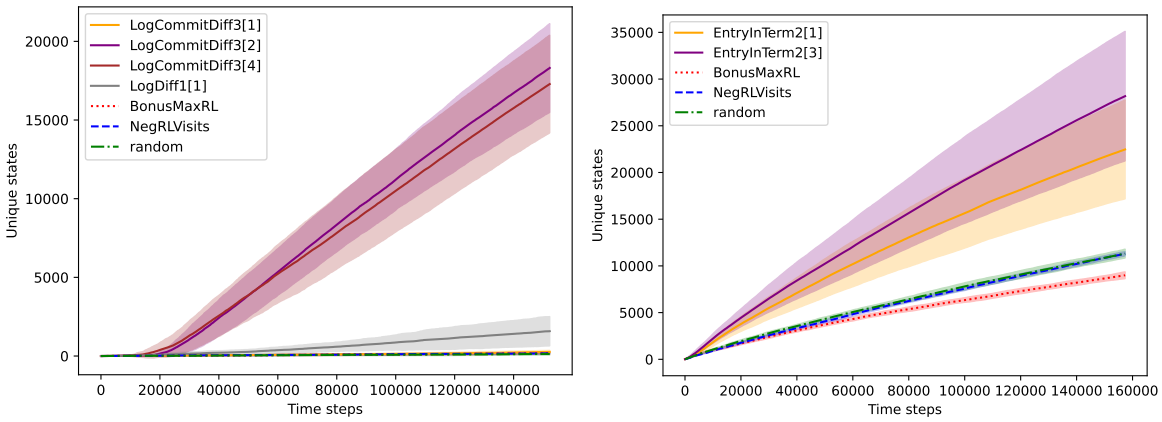
Table 4.4: Coverage results - the table shows the target coverage results in our benchmarks. Each row contains the target predicate, the number of predicates in the sequence used by WaypointRL (excluding the first one), and, for each algorithm, the average number of unique explored states (± Standard Deviation).

Benchmarks	BonusMaxRL	NegRLVisits	Random
RedisRaft			
OneInTerm3	0.70	0.86	0.01
AllInTerm2	8.25e-06	8.25e-06	8.22e-06
TermDiff2	0.14	0.40	4.46e-03
CommitEntries2	8.25e-06	8.25e-06	8.25e-06
EntryInTerm2	8.25e-06	8.25e-06	8.25e-06
LeaderInTerm2	8.25e-06	8.25e-06	8.25e-06
LogDiff1	8.25e-06	8.25e-06	8.25e-06
LogCommitDiff3	1.25e-05	1.25e-05	1.25e-05
OneLeaderOneCandidate	6.91e-03	5.17e-03	9.07e-05
Etcd			
LogCommitGap3	9.13e-05	9.13e-05	9.13e-05
OneInTerm4	9.13e-05	9.13e-05	9.13e-05
MinCommit2	9.13e-05	9.13e-05	9.08e-05
TermDiff2	9.13e-05	9.13e-05	9.13e-05
LeaderInTerm4	9.13e-05	9.13e-05	9.13e-05
AtLeastOneCommitInTerm2	9.13e-05	9.13e-05	9.13e-05
OneLeaderOneCandidate	0.98	9.13e-05	9.13e-05
LogGap2	2.91e-04	0.01	9.13e-05
AllInTerm5	1.40e-03	1.41e-03	9.13e-05
RSL			
AnyBallot3	9.08e-05	9.03e-05	9.08e-05
AllBallot3	9.13e-05	9.13e-05	9.08e-05
EntryBallot2	0.99	0.07	0.05
AnyDecree2	9.08e-05	9.08e-05	9.08e-05
BallotDiff2	0.02	8.19e-05	9.81e-05
AnyDecided3	9.08e-05	9.08e-05	9.08e-05
PrimaryInBallot2	2.91e-04	9.13e-05	9.08e-05

Table 4.5: Statistical tests results for the three benchmarks. For each predicate, we present the values of Mann Whitney U test. If the values are ≤ 0.05 then WaypointRL outperforms the compared approach.

Benchmark	Target predicate	Predicates Sequence	Target Coverage
RedisRaft	LogCommitDiff(3)	LogCommitDiff(3)	264 ± 99
		LogDiff(1), LogCommitDiff(3)	**18309 ± 2820**
		LogDiff(1), LogDiff(2), LogDiff(3), LogCommitDiff(3)	**17287 ± 3102**
		LogDiff(1)	1578 ± 931
RedisRaft	EntryInTerm(2)	EntryInTerm(2)	22468.1 ± 5295.2
		OneInTerm(2), LeaderInTerm(2), EntryInTerm(2)	28173.7 ± 6935.9
RSL	DecidedDiff(3)	DecidedDiff(3)	26.4 ± 13.2
		DecidedDiff(2), DecidedDiff(3)	**173.1 ± 70.1**
		DecidedDiff(1), DecidedDiff(2), DecidedDiff(3)	**113.7 ± 46.8**

Table 4.6: Improvements with intermediate predicates. For each target predicate (one example from each benchmark) and the sequence used to bias, we report the average final target coverage (± Standard Deviation). We use the same classes of predicates described in Table 4.3, eventually instantiated with different values.



(a) LogCommitDiff(3)

(b) EntryInTerm(2)

Figure 4.6: Different versions of predicate sequences for the same target coverage. For the predicate sequences, the legend specifies the target predicate and, between square brackets, the number of predicates in the sequence excluding the first one (true predicate).

Bug	WaypointRL	BonusMaxRL	NegRLVisits	Random
RedisRaft				
RaftRestoreLog	5704.6 (AllInTerm(2))	**6499.8**	5056.2	4897.5
HandleBeforeSleep	**23.4 (LeaderInTerm(2))**	13.8	15.1	7.4
ConnIsConnected	1.3 (LogDiff(1))	0.5	-	**4.5**
RaftAppendEntry	**9.8 (EntryInTerm(2))**	0.7	2.8	5.7
RaftBecomeFollower	**2.2 (AllInTerm(2))**	0.2	0.8	1.5
RaftApplyEntry	**0.8 (LogDiff(1))**	0.1	-	0.4
RaftDeleteEntry	**0.1 (LeaderInTerm(2))**	-	-	-
InconsistentLogs	**2.0 (AllInTerm(2))**	0.4	0.2	0.2
ReducedLogs	**7.1 (EntryInTerm(2))**	2.2	2.1	1.9
ModifiedLog	**0.3 (LogDiff(1))**	0.1	-	0.1
Etcd				
IncorrectLogRestore	**0.1 (OneLeaderOneCandidate)**	-	-	-
NilSnapshotPanic	**0.3 (LogCommitGap(3))**	-	-	0.2
RSL				
InconsistentLogs	**167.2 (AnyBallot(3))**	51.1	8.3	8.0

Table 4.7: Average occurrence of bug comparing biased exploration with different pure-exploration algorithms. The column for WaypointRL reports the highest average occurrence of the bug and the corresponding target predicate used to bias.

tentially improve a different target coverage.

4.5.5 RQ4: Does biased exploration help uncover bugs?

We evaluate the ability of different approaches to find bugs by measuring the number of bugs and the average occurrence of each bug. As shown in Table 4.7, we find that WaypointRL is better at uncovering new bugs when biasing exploration towards developer defined predicates. Furthermore, when other approaches uncover the same bug, WaypointRL reliably reproduces the bug more frequently that the other approaches. The table lists the average occurrence of each bug. Furthermore, biased exploration is able to find all the bugs while pure exploration is only able to find some.

RedisRaft We are able to identify 3 new bugs and reproduce 7 known bugs in RedisRaft. Table 4.8 provides a short description of the bugs. We capture two classes of bugs for RedisRaft when testing with using BonusMaxRL and WaypointRL. First, bugs that violate a safety property of the protocol during an episode. Second, an unexpected failure in the implementation during the episode. For the first class of bugs, we are able to capture the trace and identify any issues. However analyzing an unexpected failure in the implementation code requires a deeper understanding of the implementation codebase.

Using WaypointRL, we are able to identify all bugs. However, pure exploration approaches fail to replicate one bug. Furthermore, for some bugs, the higher average occurrence using the predicate for biased exploration correlates with the bug description. For example, RaftAppendEntry occurs when biasing exploration to add an entry to the log, RaftBecomeFollower bug is more common when biasing exploration towards states where all processes

Bug	Category	Description
RaftRestoreLog	Crash	Occurs when restoring log from file
HandleBeforeSleep	Crash	When flushing log entries to file
ConnIsConnected	Crash	When connecting to a node added to the cluster
RaftAppendEntry	Crash	When adding a new entry to the log
RaftBecomeFollower	Crash	When updating the state to follower upon receiving an append entries message
RaftApplyEntry	Crash	When applying a committed entry onto the state machine
RaftDeleteEntry	Crash	When removing an uncommitted entry from the log
InconsistentLogs	Safety violation	Two committed logs differ in an entry
ReducedLogs	Safety violation	A process loses a committed entry in the log
ModifiedLog	Safety violation	A process changes a committed entry in the log

Table 4.8: Bug descriptions for RedisRaft benchmark along with the category of bug. Crash bugs are unexpected failures in the process and Safety violation bugs are those where the trace violates safety properties

transition to term 2.

Etcd Etcd is a robust and well tested implementation that has been used in production for many years. Despite the robustness, we replicate 1 known bug and find 1 new bug with etcd with our testing efforts. The new bug, that occurs due to incorrect log restoration, is uncovered by biased exploration as well as by random exploration with the same frequency. However, the known bug is uncovered only using biased exploration.

RSL We find one new bug (safety violation) in our RSL implementation where two processes decide on different values. The bug is caught by all approaches but replicated more frequently than other approaches using biased exploration.

4.6 Related Work

Reinforcement Learning for Testing.

Two closely related works apply Q-learning techniques to test distributed or concurrent programs. QL [Muk+20] applies Reinforcement learning techniques to test concurrent message passing and shared memory programs. They introduce a novel reward mechanism that provides strong incentives for exploration. However it is unclear if the reward mechanism can be extended to bias exploration. Mallory [Men+23b] utilizes the reward mechanism of QL to improve the principles of Jepsen. Mallory relies on Q-learning to pick the next set of

failures and maximize coverage of all possible failure scenarios. In our work, we combine the modelling efforts of both these works to define new algorithms and introduce a new mechanism to bias exploration and find new bugs.

We note that RL has been used in testing in orthogonal ways, e.g., in sequential fuzzing [Red+20], in learning appropriate parameters [Wan+21], synthesizing valid inputs [BGS18; VRC06], and inputs that induce failures in control systems [Zha+21]

Reward-Free Exploration in Reinforcement Learning.

This line of work provides theoretically efficient algorithms to explore a given environment in absence of a reward function [Jin+20; ZMS20]. They provide theoretical guarantees over the coverage of the state space, also using decaying reward augmentation. Unfortunately, the number of episodes required for coverage are (5th degree) polynomial functions in the size of the states and actions and thus unsuitable in our setting. When run for a limited amount of time, they do not perform well in practice compared to our algorithms and our baselines.

Splitting Goals into Sub-goals.

Hierarchical RL relies on splitting an RL problem into subtasks, learn policies to solve each subtask, and then combine these policies to solve the original problem [SPS99; Die00; PR97]. These methods build on the idea of defining higher level actions, with multiple steps duration, to solve the subtasks. They then learn some sort of global policy that chooses which of these actions to follow at each step. While we also leverage the idea of splitting the problem into smaller tasks, our subtasks are just waypoints towards the goal with fixed priorities. Our approach is simpler and does not allow to reuse or combine these sub-policies in a structured way.

Reward Machines

Another way to leverage tasks decomposition and knowledge of the reward function structure is given with Reward Machines [Ica+22]. They allow for complex reward structure specification, hierarchical learning approach, and efficient algorithms to speed up policy optimization. Unfortunately, we can't benefit from these advantages in our setting. The efficient algorithms mainly build on the idea of decoupling the reward function from the environment transitions, allowing to simulate the result of an observed transition as if it happened at a different stage of the reward function. In our setting, the rewards strictly depend on the system transitions and hence we can't decouple them.

Temporal Goals in RL

Recent work in combining temporal logic goals and reinforcement learning has also explored the idea of intermediate goals [Jot+21; Jia+21; Alu+22; XT19]. For example, given a goal described by an automaton, a sub-goal is to reach intermediate states of the automaton between the initial and accepting states. Similar to the problem with reward machines, simulating transitions based on the sub-goals is infeasible when testing real world implementations with minimal instrumentation.

Chapter 5

ModelFuzz: model coverage guided fuzzing

In the previous two chapters we saw the benefit of allowing a developer to bias exploration and guide the state space search in order to find bugs more effectively. At the same time, effective biased search allows the developer to gain confidence in the correctness of specific parts of the codebase. We motivate the predicates required for biased exploration using the formal models of the protocols. However, the algorithms do not rely on the models for exploration. In this chapter we introduce a novel fuzzing technique ModelFuzz that addresses the challenge of connecting the model to the implementation in real time. ModelFuzz is a fully automatic testing mechanism that leverages the substantial effort put into the formal modelling of distributed protocols [Lam02; Des+13; Del+15b; Del+21; Bor+21; New+15].

We show that coverage over the formal model states serves as an effective proxy to finding bugs in the implementation. Specifically, our novelty lies in guiding fuzzing on the implementation using the coverage information of formal model. The reason for the effectiveness of the approach is due to the abstract nature of the formal model. The model only captures the *interesting* behaviors of the implementation and therefore coverage over the model translates to coverage of *interesting* states of the implementation. Unlike with fuzzing sequential programs, our use of the formal model for fuzzing is a departure from existing testing techniques. Existing notions of coverage fall on the ends of the abstraction spectrum - too fine grained or too coarse grained. Using line or branch coverage is too coarse grained since they ignore different message interleaving information and is insufficient to explore *interesting* states of distributed programs. Trace coverage, on the other hand, is too fine grained and explore different specific message interleaving which might not lead to new states.

By relying on the model, we can tune the coverage to the right abstraction and achieve better results. Our approach, ModelFuzz, shares common insights with semantic fuzzing approaches [Pad+19] and grammar-based fuzzing approaches [Le+19; GMZ20]. Semantic fuzzing aims to cover interesting program executions processing program inputs rather than spending exploration budget for exercising uninteresting, syntactic input parsing logic. While a naive fuzzer is likely to generate inputs that cannot pass the input validation and parsing stage, semantic fuzzing generates test inputs that can go deep into the execution.

Similarly, a naive event scheduler for distributed systems is likely to produce tests that spend execution budget in exercising uninteresting, network setup stages. For example, it can explore many different orderings of vote messages during the cluster's leader election phase, barely electing a leader after a prolonged execution. Our approach aims to direct testing toward interesting system behaviors, e.g., processing of user requests once a leader is elected. Similar to grammar-based fuzzing that uses formal specification of the test input to guide test generation, we use an abstract formal model of distributed systems to guide the generation of semantically interesting temporal event schedules.

We implement ModelFuzz to test distributed systems implementations using their corresponding TLA+ models [Lam02]. At the same time, we rely on the conventional fuzzing loop in the implementation. Starting from a random corpus of inputs, ModelFuzz samples an input to run on the implementation in each iteration. The resulting execution is simulated on the model to obtain coverage information. If we observe new states, similar to other fuzzing approaches, we *mutate* the input and add it to the corpus of inputs to sample from. Apart from the algorithm, the technical contribution of ModelFuzz lies in (1) defining the input for distributed systems and (2) simulating the execution on the TLA+ model. To enable (2), we augment the TLC model checker with a server that accepts a trace in a generic format to simulate.

To demonstrate the effectiveness of ModelFuzz, we test 2 Raft implementations Etcd and RedisRaft along with a motivating micro-benchmark. We find 13 new bugs in RedisRaft and 1 new bug in etcd. Furthermore, 4 of 13 bugs were detected only by ModelFuzz. In comparison, WaypointRL was able to uncover 7 of the 13 bugs with the RedisRaft, and reproduce the 1 new bug in Etcd. The two approaches are complementary and help uncover different set of bugs.

In what follows, we provide background on fuzzing (Section 5.1), followed by a description of fuzzing for distributed systems along with the ModelFuzz algorithm (Section 5.2). Then, we describe the implementation details (Section 5.3) before presenting the results of our evaluation (Section 5.4). Finally, we discuss similar existing work and compare the conceptual differences to ModelFuzz (Section 5.5).

5.1 Background

The standard fuzzer loop (e.g. AFL [Zal]) consists of the following components, (1) the system under test, (2) input generation and (3) the coverage information. Starting from a random corpus of inputs, the fuzzer will test the system with those inputs while collecting coverage information. When a new coverage state is observed, the corresponding input is mutated to generate new inputs. In effect, the fuzzing will search the random space of inputs using the guidance from the coverage. In this section, we will first describe a generic fuzzing algorithm, the components along with a motivating example.

Algorithm 5 describes the generic fuzzer loop. Apart from the three components, the algorithm accepts the test budget K as a parameter. For standard programs, the input are a sequence of bits and mutations involve flipping bits. Coverage is measured based on the structure of the program - number of lines of code executed or number of branches in the code coverage. Optimizations to the algorithm include - mutating the trace more than once

Algorithm 5: Generic fuzzer algorithm

Input: System under test S
Input: Coverage guidance G
Input: Input generator I
Input: Test budget K

$inputPool \leftarrow ()$

for *episode* $k = 1, \cdots, K$ **do**
  if $|inputPool| = 0$ **then**
    $inputPool \leftarrow I.reseed()$
  $input \leftarrow inputPool.pop()$
  $trace \leftarrow S.run(input)$
  if $G.haveNewState(trace)$ **then**
    $inputPool.push(I.mutate(input))$
  $G.record(trace)$

given by a parameter $mutationsPerTrace$, reseed at a fixed frequency and reseeding the input when the coverage saturates. In ModelFuzz, we adopt two of the optimizations - mutating more than once and reseeding at a fixed frequency.

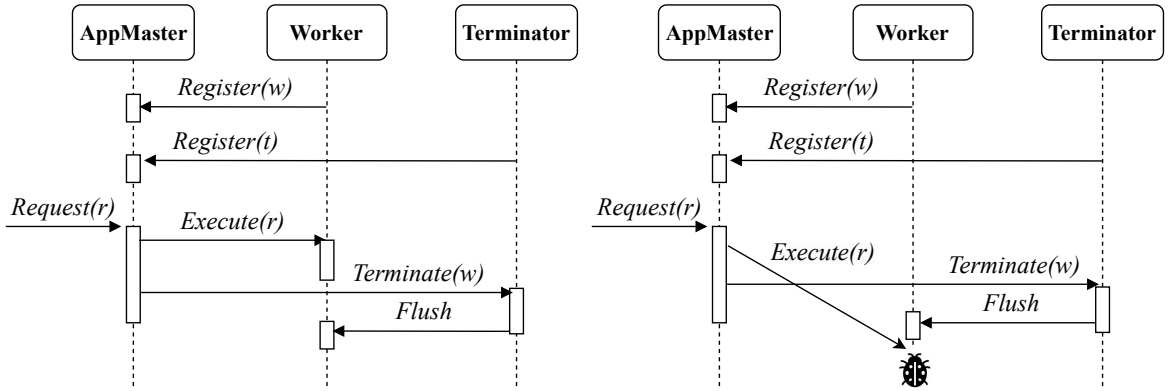
The system S in Algorithm 5 is characterized by the function run. Similarly, coverage guidance G by $haveNewState$ and $record$, and, input generator I by $reseed$ and $mutate$ functions. When describing ModelFuzz for distributed systems. We will concretely define these functions.

Motivating example

Consider a parallel processing environment - A **master** process sends work to **workers** in the form of execute messages. The master also sends commands to a **terminator** which clears the work at a worker by sending flush messages. All three process type - master, worker and terminator - communicate by sending messages and therefore contain inbound messages queues. The correct working of the environment is shown in Figure 5.1a The worker stores messages in a buffer and processes them sequentially. The buffer is cleared when a flush message arrives.

A correct worker will check the buffer before executing a request. A buggy worker will crash if this check is not performed. Figure 5.1b illustrates the buggy execution where the terminator schedules a flush before the worker executes a request. Due to the flush, the buffer is now empty and any attempt by the worker to execute will crash it.

To test an implementation of this system, we first need to capture the messages. Delivering messages based on the input will produce deterministic executions necessary for testing. A naive notion of inputs to the system would be the sequence of messages that are processed throughout the execution. However, since the messages are a consequence of the process executing, an arbitrarily generated input will not necessarily be a valid input. For example, we cannot have an execute message in the execution without first observing a request. Therefore, we fix the input as the sequence of processes scheduled in an execution. For ex-



(a) Correct example (b) Buggy example

Figure 5.1: A parallel processing system with a master, worker and terminator that process work in parallel while communicating over the network

ample, *master, worker1, worker2, terminator, master, worker1* is an input where we schedule the corresponding process and deliver messages if any in the inbox of that process. Note that the result of executing an input is a trace - defined by the sequence of messages along with the states. A mutation on the input would be to swap two processes or insert a new process in between two.

Now let us consider the different notions of coverage. Existing notions are either too coarse grained or too fine grained. Line and branch coverage is too coarse grained.Intuitively, a single input will explore all lines of the implementation - initializing, executing work and flushing. However, we still leave large set of traces uncovered. The same is true for notions of branch coverage. Trace coverage (based on the notion of Mazurkiewicz traces), on the other hand, is too fine grained. For example, although the messages *Register(w)* and *Register(t)* are dependent, their relative ordering does not affect the system state. However, the ordering of *Request(r)* affects the reached system state; the system handles the request only if it is delivered after the two registration messages. The given system has 10 possible message orderings with 8 Mazurkiewicz traces (capturing the commutativity of the *Execute* and *Terminate* messages). However, the set of all possible system states can be covered by running fewer executions, e.g., only 2 executions for this example. Therefore, guiding testing using unique traces as a coverage notion results in exploring redundant behaviors.

Our notion of coverage defined by the different states of the TLA model (state-based) that captures the system is better suited to guide the testing of such distributed systems. Guiding testing using new behaviors in the model results in an efficient exploration of the different possible executions on the system that are truly unique. To compare with trace based coverage notion consider the following traces that are explored.

E1 `Request(r), Register(w), Register(t)`
E2 `Register(w), Request(r), Register(t)`
E3 `Request(r), Register(t), Register(w)`
E4 `Register(t), Request(r), Register(w)`
E5 `Register(w), Register(t), Request(r), Execute(r), Terminate(w), Flush`
E6 `Register(t), Register(w), Request(r), Terminate(w), Execute(r), Flush`

Trace-based coverage would label all these executions as *interesting* since each belongs to a different coverage class (i.e., they deliver `Request(r)`, `Register(w)`, and `Register(t)` to the same process in a different order). Generating new tests around all these executions leads to a high number of redundant executions since many of them already produce the same system behavior. In contrast, state-based coverage identifies the coverage of new states in the executions of E5 and E6, which hit some new system states that are not observed in E1-E4. Therefore, state-based coverage-guided testing generates new test cases only around these executions.

5.2 Fuzzing distributed system with models

We will now describe the formal model of a distributed system, an input to the system, our coverage guidance measure and mutations to the input thereby describing the core components of the ModelFuzz algorithm.

5.2.1 Distributed system model and inputs

In a nutshell, a distributed system S consists of a set of processes that concurrently operate on their own local states and communicate with each other by exchanging asynchronous *messages* ($\mathcal{M}$). Each process contains a FIFO queue of incoming messages. By processing a message from the queue, the process may update its local state and/or send new messages to the processes. We consider FIFO message queues that preserve the order of messages between the same sender-receiver pairs, as in other works such as P [Des+13], Coyote [Del+21] or Akka [Lig11]. Note that this is a deviation from our previous models considered in this thesis. To enable fuzzing we need to randomly generate inputs that are executable by definition. The FIFO queue model allows us to generate high-level inputs that specifies the schedule of processes and avoids reasoning about the order of specific messages. Any input that reasons about the order of specific messages is not always executable. Certain messages sent by the processes are conditioned on the order and contents of prior messages.

Running an input on the distributed system is equivalent to implementing a monitor from Section 3.2. In this case, the monitor enforces the specific input which involves maintaining a mail box, sending messages, crashing, and starting processes.

Given a set of processes $\mathcal{P}$ and a size of the input n, we define a fuzzer input as follows,

Definition 7. *A **fuzzer input** $I = (\langle p_0, a_0\rangle, \langle p_1, a_1\rangle, \cdots, \langle p_n, a_n\rangle)$ where $p_i \in \mathcal{P}$ and $a_i \in \{message, start, stop\}$. In words, the input is a sequence of processes to schedule along with an action at each step. The actions are one of,*

- $message$ *- deliver messages to the process from its inbox*
- $start$ *- start the process (requires it to be currently stopped)*
- $stop$ *- crash the process*

Valid inputs are those where the $start$ actions are well defined i.e., the process that needs to be started has been stopped before. In practice, we add an optimization to the input

- in step i with action $a_i = message$, instead of delivering a single message from the inbox, we parameterize it to at most t_i messages. This is equivalent to having t_i actions of message for the same process in the input. The motivation for the optimization is to deliver more messages to observe meaningful state changes in the system. Delivering only a single message at each action is too fine grained a step in the model.

Algorithm 6: Running an input I on distributed system S

```
def updateBuffers(bs, es):
    for e ∈ es do
        if e.type = send then
            bs(e.p).append(e.val)
    return bs
def S.run(input):
    ∀p ∈ P, bs(p) = ()
    active ← P
    eventTrace ← ()
    for ⟨p_i, a_i⟩ ∈ input do
        if a_i = message(t_i) then
            es ← p_i.deliver(bs(p_i), t_i)
                    // deliver at most t_i messages from the buffer
            eventTrace.append(es)
            bs ← updateBuffers(bs, es)
                // update the buffers based on the resulting events
        if a_i = stop ∧ p_i ∈ active then
            p_i.stop()
            active ← active \ {p_i}
            eventTrace.append((p_i, stop, ϕ))
        if a_i = start ∧ p_i ∉ active then
            p_i.start()
            active ← active ∪ {p_i}
            eventTrace.append((p_i, start, ϕ))
    return eventTrace
```

The result of executing an input on the system is an event trace. To understand the event trace, let us define the semantics of executing the input. The state of the system consists of two components (1) the FIFO input queues of each process (2) the set of active process. More formally, the state s is the tuple $(bs, active)$ where $bs : \mathcal{P} \rightarrow [\mathcal{M}]$ and $active \subseteq \mathcal{P}$. Algorithm 6 outlines the execution of an input on the system. The result is $eventTrace$ that captures the execution on the system. Here, an event $e \in \mathcal{E}$ is the tuple $(p, type, val)$ where $p \in \mathcal{P}, type \in \{send, receive, internal\}$ and $val \in \mathcal{V}_E$.

5.2.2 Coverage guidance

The guidance mechanism connects the obtained traces to the model. We will refer to our coverage mechanism as **model-coverage**. Specifically, the guidance mechanism simulates the trace on the model, stores and tracks all the states observed and, identifies when a trace

leads to a new state in the model. In our experiments, we are concerned with the TLA+ models of the implementation and to simulate traces, we modify the TLC model checker. However, the process is not fully automatic and requires as input a mapper that maps the events in the trace to actions in the model. We will later elaborate on the specific mapping we use for the Raft protocol.

Algorithm 7: Guidance algorithm G

Input: Event Mapper Φ
Input: Model M

$G.states = \{\}$

def $G.haveNewState(eventTrace)$**:**
  $modelActions \leftarrow \Phi(eventTrace)$
  $modelStates \leftarrow M.run(modelActions)$
  if $modelStates \setminus G.states \neq \phi$ **then**
    return *true*
  return *false*

def $G.record(eventTrace)$**:**
  $modelActions \leftarrow \Phi(eventTrace)$
  $modelStates \leftarrow M.run(modelActions)$
  $G.states \leftarrow G.states \cup modelStates$

Formally, the model is a transition system $M = \langle Q, I, A, \delta \rangle$ where Q is a set of states, I is the set of initial states, A is the set of actions, and $\delta \subseteq Q \times A \times Q$ is a set of transitions. An action $a \in A$ is enabled at state $q \in Q$ iff $(q, a, q') \in \delta$ for some $q' \in Q$. A run of M is a sequence $\rho = q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_m} q_m$ where $q_0 \in I$ and $(q_i, a_i, q_{i+1}) \in \delta$ holds for all i. The mapper $\Phi : [\mathcal{E}] \rightarrow [A]$ is a function that maps the given sequence of events to a sequence of actions. Note the actions on the model differ from the actions on the input. Algorithm 7 defines the methods of the model-coverage guidance mechanism and accepts as input the formal model M and the mapper Φ

Trace-coverage is an alternative notion of coverage that we compare against in our evaluation. Trace coverage tracks the set of unique traces covered otherwise known as Mazurkiewicz traces [Maz86]. The set of all possible traces is partitioned into equivalence classes where traces in the same class differ only in the ordering of unrelated events. here, we refer to the traditional *happens-before* ordering of events [Lam78]. However, as with the motivating example, trace coverage explores redundant states since it is too fine grained. It is possible that two distinct Mazurkiewicz traces results in the same state trace in the model. Therefore, we observe guidance using trace coverage does not lead to high coverage of model states.

Line-coverage, on the other hand, is too coarse grained. A few execution traces reaches high coverage of lines executed, beyond which fuzzing proceeds very similar to random exploration. Therefore, guidance using line coverage does not lead to high coverage over the model state. In contrast, we observe that guidance using model-coverage does not degrade the line coverage of the implementation.

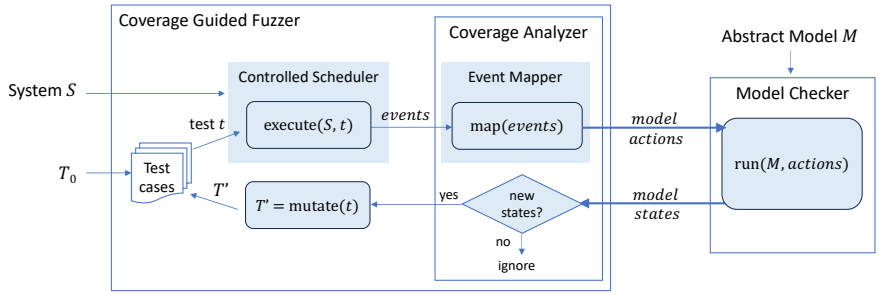


Figure 5.2: The workflow of ModelFuzz

5.2.3 Mutation strategies

When a coverage guidance mechanism designates an input as *interesting* - leads to a new states, the input is mutated to generate new traces. The key to capitalizing on the coverage guidance is to define meaningful mutations. Here, we elaborate on our mutation strategies. To recall, an input I is the sequence $(\langle p_0, a_0\rangle, \langle p_1, a_1\rangle, \cdots, \langle p_n, a_n\rangle)$ where the actions are one of $message(t_i), start$ or $stop$. We define a combination of 3 mutations to apply on the input as follows,

- **SwapProcesses** randomly selects two schedule indices i, j in I, and swaps the processes p_i and p_j.
- **SwapCrashProcesses** randomly selects two schedule indices i, j where the actions are $stop$, and swaps the positions of the processes p_i and p_j (for schedules with a single crashing process, it changes the process and, for schedules without a $stop$, nothing is changed),
- **SwapMaxMessages**, which randomly selects two schedule indices i, j with actions $message(t_i)$ and $message(t_j)$ respectively and swaps t_i and t_j, i.e., the number of messages to deliver at these positions.

5.2.4 The ModelFuzz algorithm

Together, the three components - distributed system, coverage guidance and the input generator describe the ModelFuzz testing algorithm. We illustrate in Figure 5.2, the mechanism by which ModelFuzz combines the different components. Starting from a set of initial inputs T_0, ModelFuzz runs each input on the system S, collects the coverage information using the coverage guidance and mutates *interesting* states to replenish the input corpus. The coverage guidance internally uses the mapper Φ to simulate the trace on the abstract model M. This is denoted as the "Event Mapper" in Figure 5.2.

Note a few caveats. In the figure, we refer to the inputs as test cases in line with the standard terminology used in fuzzing literature. As mentioned earlier, ModelFuzz incorporates two optimizations - (1) mutations are performed more than once when the input leads to an *interesting* trace and (2) ModelFuzz reseeds the input corpus (test cases) periodically.

5.3 Implementing ModelFuzz

As shown in Figure 5.2, we need to implement a *controlled scheduler* to enforce executions on the system, *coverage analyzer* that includes a *mapper* to measure coverage and a *controlled model checker* to simulate the trace on the model. In this section, we outline the technical challenges we overcome to implement ModelFuzz. Subsequently, we will use Raft as an example to describe a concrete event mapper. Also shown in the Figure 5.2 are the two sections of the implementation. First, the main fuzzer loop that drives the testing and second, the controlled model checker.

5.3.1 Controlled scheduler

The controlled scheduler is responsible for executing an input I on the distributed system S and obtain the resulting event trace. To that end, the controlled scheduler requires control of all in-flight messages to enforce delivery order and control of the processes to start and stop when needed. In our evaluation, we instrument implementations and write specific process control mechanisms to enable the two actions.

The event trace is obtained also by instrumenting the implementation. At each step of the input, when messages are delivered, processes are started or stopped, we record events. Additionally, we also record internal events that are necessary for the mapper.

The controlled scheduler is an implementation of the *monitor* we define in Section 3.2. As with WaypointRL, the *monitor* here is not driven by the events. Instead, the *monitor* has an explicit input trace to reproduce with clear scheduling boundaries. The actions of the input correspond to the actions of the RL agent and therefore, we reuse the extended implementation of the *monitor* from WaypointRL.

5.3.2 Event mapper

The event mapper translates the sequence of concrete events executed on the system by the controlled scheduler into a sequence of actions in the abstract model. However, the information contained in the events recorded by the controlled scheduler is specific to the implementation under test and cannot be mapped directly onto actions on the abstract model. Furthermore, different implementations of the same protocol (that share a model) may encode the parameters differently. Therefore, the event mapping has to be specific to the implementation under test.

The translation should ensure that the resulting abstract state captures the concrete implementation state at each step. This means that the translation may ignore some unnecessary events in the implementation. Indeed, in our event mapper for Raft, we observe the need to ignore certain events. For example, the model does not capture `Heartbeat` messages exchanged between processes, which are regularly sent to monitor whether the processes are alive. Hence, when performing the translation, we do not map the message send and receive events of `Heartbeat` messages to any actions on the abstract model.

5.3.3 Controlled model checker

The resulting actions from the event mapper are used by the controlled model checker to enforce the execution on the model. Unlike a standard model checker, which explores the whole state space of the abstract model, ModelFuzz requires exploring only the specific execution defined by the sequence of actions and returning the visited set of model states. For this, we implement a simulation engine that (i) controls the next action to take at each current state of the model execution and (ii) records the visited states to provide them as feedback to the fuzzer.

Our implementation uses TLA+ [Lam02] models of the distributed systems, and we implement the controlled model checker for the TLC explicit state model checker [YML99] in the TLA+ Toolbox [KLR19].

5.3.4 Testing Raft protocol implementations

In our evaluation, we test implementations of the Raft protocol. Here we elaborate on the specific implementation details related to the Raft protocol, including the specific event mapping.

Abstract model

We use the TLA+ model of the Raft protocol made available by the protocol's authors[1] and extend it by modeling (i) crash and restart of the processes and (ii) snapshot operations. Besides the abstract variables for the internal states of each process (e.g., the term number, its log of requests), the extended model uses an additional variable to keep the set of active processes in a cluster and introduces crash and restart actions. The crash and restart actions are enabled for the active and crashed processes, respectively, and they update the set of active processes in the cluster.

Based on empirical evidence that implementation bugs can occur in the snapshot processing logic, we also extend the model to capture snapshot operations so that the model can guide testing toward executions where processes trigger snapshots and restore them upon recovery. Specifically, we introduce a snapshot index for each process and actions to update the processes' snapshot indices.

To ensure that the extended TLA+ model satisfies the original correctness specifications, we run the model-checker on the extended model. The extended model is available open source [2].

Abstractions of the model states

While the abstraction provided by the TLA+ model is useful in guiding the fuzzer, we observe the need for further abstracting the set of observed states to guide the exploration to 'interesting' parts of the state space.

[1] https://github.com/ongardie/raft.tla

[2] https://github.com/burcuku/tlc-controlled-with-benchmarks/blob/main/tla-benchmarks/Raft/model/raft_enhanced.tla

Concretely, the states of the existing TLA+ model are defined by the local states (e.g., operation logs) of each process, along with the current term numbers of each process. Therefore, the model states differentiate between the system states with the same set of local process states if they are reached in different term numbers. Consider the leader election phase of an execution. Although the local states of the processes do not change, unsuccessful leader election rounds result in hitting new system states since the processes increment their term numbers. Such state information guides the fuzzer toward exploring executions with growing term numbers without covering interesting system behavior.

To guide the fuzzer with more precise state information, we abstract the term numbers in the model state within the TLC model checker. Specifically, we merge two consecutive states in an execution that only differ in the term numbers of the non-leader processes. Note that we do not modify the TLA+ model of the system; we implement the state abstraction within the TLC model checker and communicate the abstracted states to the fuzzer.

Event mapper

We implemented event mappers for two different implementations of the Raft protocol (in etcd and Redis) to the abstract TLA+ model. The event mappers map the events to the abstract actions in two steps.

The first step converts the implementation-specific encoding of messages to a standard JSON encoding before passing it to the controlled model checker. The standardization helps with mapping different message encodings for different implementations of the protocol. Note that the encoding of the messages to JSON format resides with the instrumentation of the specific implementation under test.

In the second step, we map the standardized events in JSON format to abstract actions on the model. The second step is embedded in the controlled TLC model checker. Our implementation of the controlled TLC model checker parses the TLA+ model and exposes a remote procedure call (RPC) interface to run simulated executions. The interface accepts a sequence of standardized events and outputs the sequence of abstract states observed in the model.

Our mapper for Raft maps system events into three classes of model actions:

1. The cluster actions related to the set of active and participating processes: *AddProcess, Crash, Restart*, all with a process ID as an argument. The action *AddProcess* is mapped when a process is added into the cluster, *Crash* corresponds to the system events crashing a process, and *Restart* corresponds to system events restarting a process.
2. The protocol actions of a process - *Timeout*, *ElectLeader* and *UpdateSnapshotIndex*. Identified with a process id as an argument in addition to other arguments. (i) *Timeout* action is mapped to when a process initiates a term change, (ii) *ElectLeader* is mapped to when a process is designated to be the leader, and (iii) *UpdateSnapshotIndex* is mapped to when a process creates a snapshot.
3. The protocol actions for processing the protocol message, with the arguments corresponding to the contents of the message. *ClientRequest* is mapped to when a process receives a request. *HandleRequestVoteRequest* mapped to sending a request vote request message and the correspoding *HandleRequestVoteResponse* mapped to request

vote response message. Similarly, *HandleAppendEntriesRequest* is mapped to append entries request message and a special event *HandleNilAppendEntriesRequest* is mapped to append entries request message with no entries. Finally, *HandleAppendEntriesResponse* is mapped to append entries response messages.

5.4 Evaluation

We implement ModelFuzz to test two implementations of the Raft protocol [OO14]: Etcd-raft[3] and RedisRaft[4] along with an implementation of a parametrized version of the example system presented in Section 5.1 in the Coyote framework [Del+21].

Our implementation uses the TLA+ model of the Raft protocol to measure abstract state coverage. We use the TLA+ model made available by the protocol's authors [Git] and extend it[5] to model (i) crash and restart of the processes and (ii) snapshot operations.

We evaluate ModelFuzz compared to pure (unguided) random testing as well as structural code coverage-guided and trace coverage-guided fuzzing strategies. Specifically, we evaluate the performance of ModelFuzz in terms of *test coverage* and *bug finding ability* answering the following research questions:

RQ1 How does the test coverage of ModelFuzz compare to other strategies?
RQ2 Is ModelFuzz more effective at detecting bugs than the other strategies?

We address **RQ1** by comparing the abstract state coverage of ModelFuzz to pure random, line coverage-guided, and trace coverage-guided fuzzing strategies. We address **RQ2** by evaluating the bug-finding effectiveness of different testing strategies using two measures [BSM22] (1) the unique number of bugs found and (2) the number of test executions to find a bug.

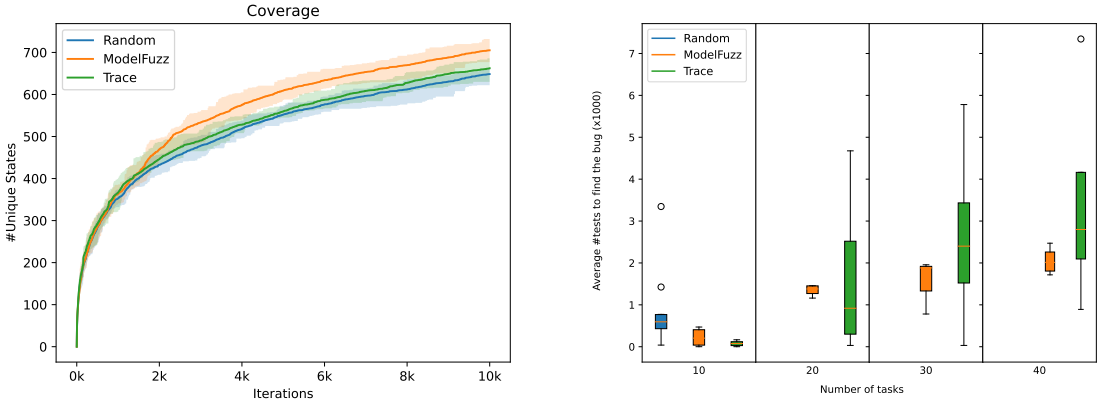
Statistical evaluation [AB14]. We analyze the statistical significance of our coverage results by running the Mann-Whitney U-test [MW47]. We assess ModelFuzz's bug-finding ability compared to the other testing strategies using Vargha and Delaney's $\hat{A}_{12}$ statistic [VD00], with $\hat{A}_{12} = 0.6$ as in previous literature [Men+23b].

Test configuration. We run the fuzzers with an initial set of $|T_0| = 20$ random test cases. For each test case that covers a new state, we create five new test cases by mutating the original test case. We multiply the number of generated test cases proportionally with the number of new states observed in the test execution. We periodically repopulate the set of test cases.

Test oracle. We check the correctness of test executions by checking for assertion violations, exceptions, and crashes. We also check the serializability of the operations in etcd and Redis, running Elle [AK20] on the executed operation history.

Experimental setup. We run the experiments on an Intel(R) Xeon(R) CPU E5-2667 v2 machine with 32 cores and with 252GB of RAM.

[3]https://github.com/etcd-io/raft
[4]https://github.com/RedisLabs/redisraft
[5]https://anonymous.4open.science/r/tlc-controlled-with-benchmarks-8E36/tla-benchmarks/Raft/model/raft_enhanced.tla



(a) (b)

Figure 5.3: Testing the microbenchmark. (a) Test coverage for $m = 6$ workers and $n = 40$ tasks (b) #tests to find the bug for varying #tasks with $m = 6$ workers.

	$m = 5$				$m = 6$				$m = 7$			
n	10	20	30	40	10	20	30	40	10	20	30	40
Random	**0.81**	**1.00**	**1.00**	**1.00**	**0.86**	**1.00**	**1.00**	**1.00**	**0.71**	**1.00**	**1.00**	**1.00**
Trace	0.33	1.55	**0.80**	**0.83**	0.32	0.43	**0.73**	**0.75**	0.14	0.14	0.37	**0.60**

Table 5.1: Pairwise $\hat{A}_{12}$ statistic results against ModelFuzz for the microbenchmark with varying m and n.

5.4.1 Micro benchmark in Coyote

We implemented a parametrized version of the example in Section 5.1 in the Coyote framework [Del+21]. The implementation parametrizes the system in (i) the number of worker processes and (ii) the number of `Execute` task messages that need to be processed to handle a request. For (i), we generate m workers that need to register to the `AppMaster` before `AppMaster` can process a client request. For (ii), we modify the processing of `Execute` so that the `Worker` divides the work into a chain of n number of tasks.

The system's possible executions involve different interleavings of the `Terminate` message with the chain of `Execute` messages sent to the `Worker`. We seeded a concurrency bug that occurs if `Terminate` is processed by the `Worker` just before the last `Execute` message while processing a client request. The bug gets harder to trigger with increasing m and n since it requires all m workers to register to `AppMaster` before `Request` and also to deliver the chain of `Execute` messages except for the last one before `Terminate`. [6]

We ran the microbenchmark with 10K iterations with varying $m = \{5, 6, 7\}$ workers and $n = \{10, 20, 30, 40\}$ tasks over ten runs for each configuration. For this system, we do not compare with line-based coverage as the existing coverage tools do not integrate with the framework we are using.

[6] Available at https://anonymous.4open.science/r/coyote-modelfuzz-5757

Coverage. Figure 5.3a shows the coverage of the abstract states of the microbenchmark with $m = 6$ workers and $n = 40$ tasks, which is representative of different parameter configurations. Since the microbenchmark is a small example with a small state space, the difference in the explored number of unique abstract states among different testing approaches is not large. However, the results show ModelFuzz's ability to cover more abstract states compared to random testing and trace coverage guidance. Our Mann-Whitney U-tests show that ModelFuzz achieves statistically significantly better coverage results at $\alpha = 0.05$, compared to random testing and trace-guided fuzzing with p-values $\{0.0001, 0.0004\}$.

Bug finding. We observe a trend with ModelFuzz where it consistently detects the injected bug faster than pure random and trace coverage guided testing approaches with increasing m and n.

Figure 5.3b plots the number of test iterations to trigger a bug for increasing n number of task messages with a fixed $m = 6$ processes. The results show that the increasing number of task messages makes the bug more difficult to detect as it is triggered deep in the execution space. This effect is most evidently seen with pure random testing, as it fails to detect the bug after $n = 10$. Trace coverage guided testing achieves a more consistent variance among different campaigns. However, we observe a trend in its median value for the first iteration to detect the bug, where it declines as the bug gets harder to detect with increasing n. The performance degradation is not as significant for ModelFuzz, as its median value does not change drastically among the experiments.

We use Vargha and Delaney's $\hat{A}_{12}$ statistic to analyze the significance of our bug-finding results on all of the parameter configurations. Table 5.1 lists the pair-wise $\hat{A}_{12}$ statistic values against ModelFuzz for testing the microbenchmark with varying m number of workers and n number of tasks. The results show the statistical significance of 17 out of 24 configurations (highlighted in bold), which indicates that ModelFuzz is more effective than the other testing approaches at finding the bug.

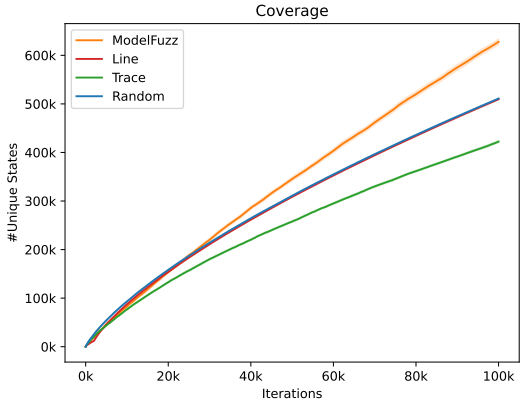
5.4.2 Etcd-raft

Etcd-raft[7] powers the popular distributed key-value store etcd[8]. It is a well-tested, production-ready implementation of Raft used by companies such as Cloudflare. The implementation is 7k lines of go code. We instrument its source code with an additional 1k LOC[9] to implement the fuzzer loop, gain control of the messages exchanged between processes, and implement necessary adapters to communicate with the controlled TLC model checker.

We tested the executions of Etcd-raft with three processes and with five client requests. We ran our tests with a crash quota of 10 and delivered a maximum of 5 messages at each step. We report the results over an average of 20 runs, each with 100k test iterations.

Coverage. Figure 5.4a reports the test coverage of the test harnesses in the number of abstract states observed with different strategies. The results show that model-guided

[7] https://github.com/etcd-io/raft
[8] https://etcd.io
[9] https://anonymous.4open.science/r/etcd-fuzzing-29D8/README.md



(a)

Guidance	Avg. #iterations
ModelFuzz	1013
Random	599
Line	1342
Trace	940

(b)

Figure 5.4: Testing etcd. (a) Test coverage of abstract states (b) Average number of test iterations to find the synthetic bug. The new bug is only detected by ModelFuzz.

Bug	Random	Trace	Line
Seeded	0.57	0.53	0.47

Table 5.2: Pairwise $\hat{A}_{12}$ statistics against ModelFuzz for etcd.

test generation of ModelFuzz outperforms unguided random testing (Random), coverage-guided fuzzing using line coverage (Line), and coverage-guided fuzzing using trace coverage (Trace) in the explored number of unique system states. ModelFuzz covers 1.22x more states than Random, 1.23x more states than Line, and 1.48x more states than Trace coverage approaches. Comparing model-guided fuzzing and structural guidance, we find that in both cases, the code coverage saturates at 47.9%. Similarly, comparing model-guided fuzzing and trace-based fuzzing, we find that in both cases, we explore 10k unique traces.

To mitigate the randomness, we perform a statistical U test (Mann-Whitney U Test) and conclude that ModelFuzz's coverage of model states is significantly higher than all other approaches. The null hypothesis we consider is that ModelFuzz's final coverage is greater than other guidance measures. We obtain p values of $1.53\text{e}{-10}$, which is less than the acceptable measure of 0.05.

Bug finding. Etcd-raft has been the subject of many extensive testing approaches. However, we find one new bug in addition to reproducing a seeded bug. The seeded bug modifies the condition for checking if a process has a quorum of votes. Specifically, we change the valid quorum size from $n/2+1$ to $n/3+1$. The new bug we found is more subtle and leads to a process crash when accessing a missing snapshot. We reported the bug[10] to developers.

To answer **RQ2**, we analyze the number of detected bugs and the number of test iterations to discover a bug using different strategies. While the seeded bug can be found in each of the 20 trials by all of the strategies, the new bug can only be found using ModelFuzz.

[10]https://github.com/etcd-io/raft/issues/108

Figure 5.4b reports the average number of iterations to discover the seeded bug using different strategies. The bug can be found by random search faster on average. This can be explained by the characteristics of the bug, which is easily triggered in executions with quorums of only $n/3+1$ processes. Table 5.2 compares the distributions of the first occurrence of the seeded bug in each trial against ModelFuzz for the different guidance approaches. The Vargha-Delaney ($\hat{A}_{12}$) statistical significance test shows that no approach is significantly better for the seeded bug.

Overall, the results show that ModelFuzz is more effective at detecting bugs, as only ModelFuzz detects the new bug, and all approaches show comparable performance for the seeded bug.

5.4.3 RedisRaft

RedisRaft[11] powers the popular high-performance Redis distributed key-value store. RedisRaft compiles into a module that can be loaded onto the main Redis server. The module enables different Redis servers to behave as a group and commit client requests in the same order. The module internally uses a minimal Raft library written in C. Overall, the module consists of 30k lines of C. Our instrumentation of the module requires an additional 1.5k lines of C[12] and Go[13] code, where the Go code implements the fuzzer loop.

We tested RedisRaft running the fuzzer with three processes and five client requests. Similar to Etcd-Raft, we ran the test executions with a crash quota of 10 and delivered a maximum of five messages at each step. For each test run, we execute 20K test iterations and report the average results of 20 test runs.

Coverage. Figure 5.5a illustrates the average coverage measures for 20K test iterations for each testing strategy. Similar to Etcd, we show that ModelFuzz can obtain better coverage over model states compared to random testing, line coverage guided, and trace coverage-guided fuzzing strategies. On average, ModelFuzz observes 2.58x more states than random exploration, 2.43x more than line coverage, and 2.84x more coverage than trace-guided fuzzing. As with Etcd-Raft, we answer **RQ1** with an observation that model guidance outperforms random exploration and other coverage-guidance strategies in the coverage of explored system states.

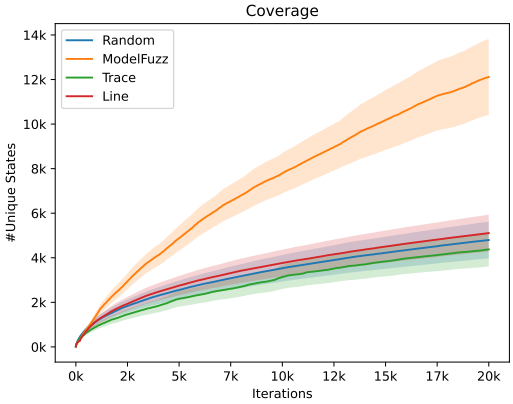
Similar to Etcd, we perform the Mann-Whitney U test over the final coverage numbers to mitigate the effect of randomness. We obtain p values of $1.23\text{e}{-4}$ vs random, $8.04\text{e}{-5}$ vs trace and $5.26\text{e}{-4}$ vs line. The U tests conclude that ModelFuzz covers significantly more states than the other testing strategies.

We also analyze the branch coverage of the tests and observe that guiding the test executions using model coverage does not degrade the coverage over traditional code coverage metrics. Table 5.5b reports the mean and standard deviation branch coverage of the different guidance methods. We measure branch coverage of the source code in C using the

[11] https://github.com/RedisLabs/redisraft

[12] https://anonymous.4open.science/r/instrumented-redisraft-1485/README.md

[13] https://anonymous.4open.science/r/redisraft-fuzzing-E49B/README.md



(a)

Method	Branch coverage
ModelFuzz	149.14 ± 111.80
Random	141.07 ± 87.36
Trace	151.07 ± 107.94
Line	150.64 ± 97.02

(b)

Figure 5.5: Testing RedisRaft. (a) Test coverage of abstract states (b) Average branch coverage of 20 runs.

gcov tool. However, we observe high variance in the data, which we attribute to the coverage measurement tool. As we run multiple copies of the source code during each test iteration, the branch coverage is combined for each copy. However, we found that the gcov tool does not always merge the coverage values of concurrent invocations correctly. Failing to merge coverage information results in the high variance we observe with the branch coverage measures presented in the table.

Bug finding. Our experiments for testing RedisRaft discovered 14 different bugs, two of which are known bugs reported in RedisRaft's issue tracker, and the remaining 12 are new, previously unknown bugs. The bugs occur in the existence of process crashes and restarts with certain orderings of events, and they manifest as thrown exceptions or assertion violations. We investigated the bugs and reported them in the issue tracker of the RedisRaft open-source repository.[14] Table 5.3 briefly describes the new bugs we discovered.

Among all 14 bugs, ModelFuzz found more bugs than other guidance approaches. Specifically, ModelFuzz found 13 bugs, while random exploration, trace-guided, and line-guided found only 10 of them. Furthermore, ModelFuzz is faster in reproducing seven of the bugs compared to other approaches, where Random exploration and Line-guided each find three, and trace-guided finds one bug faster. For each bug and guidance method, the table in Figure 5.4 lists the number of runs that find the bug (in parenthesis), and the average first occurrence of the bug in the successful runs. For each bug, we highlight the lowest average first occurrence in bold. Among the bugs, three of them are found only using ModelFuzz. We classify a bug as rare if it was found in at most five trials. ModelFuzz finds four of the rare bugs faster than the other approaches.

For each of the 14 bugs, we calculate Vargha and Delaney's $\hat{A}_{12}$ statistics to analyze the pair-wise statistical significance of the results. Table 5.5 reports the $\hat{A}_{12}$ statistics against ModelFuzz for each bug, highlighting the results with statistical significance in bold. The analysis shows that ModelFuzz detects the bugs $\{6, 12, 13, 14\}$ statistically significantly

[14]https://github.com/RedisLabs/redisraft/issues (Issue numbers: #643-#649)

Table 5.3: The new bugs found in RedisRaft

ID	Bug description
3	Process crashes when restoring the log from a snapshot stored on disk.
4	Process crashes when polling peer connections. Specifically, a segmentation failure is raised when reading the connection information of a peer.
5	After receiving information of a newly added node, the process crashes when setting a flag indicating the node has been successfully added.
6	Redis server crashes when checking for active client connections.
7	Process crashes when updating the log after receiving `AppendEntries` from the leader. Specifically occurs when the process has to delete existing entries.
8	Process crashes when updating the snapshot index offset
9	Process crashes when sending `AppendEntries` and reading from a corrupt log.
10	Process crashes when updating the state to follower upon receiving a message from the leader.
11	Process crashes when updating the state to follower upon receiving a message of a higher term.
12	Process fails to update its current term upon receiving `AppendEntries` with a higher term.
13	Process fails to update its current term upon receiving `RequestVote` with a higher term.
14	Process fails to update its current term upon receiving `RequestVoteResponse` with a higher term.

ID	ModelFuzz	Random	Trace	Line
1	299(20)	**227**(20)	368(20)	256(17)
2	10409(15)	13420(13)	8518(11)	**7592**(10)
3	48(20)	**19**(20)	32(20)	43(17)
4	**10255**(17)	12823(18)	11600(18)	10581(14)
5	578(20)	696(20)	945(20)	**482**(17)
6	**8334**(3)	-	-	17784(1)
7	6925(1)	14345(4)	-	**6512**(2)
8	-	-	**16275**(1)	-
9	**11155**(16)	12449(12)	12766(13)	15157(13)
10	11748(2)	**6598**(3)	18001(1)	9680(2)
11	**12031**(4)	14041(4)	12158(8)	12261(9)
12	**5709**(1)	11832(2)	16097(1)	-
13	**6563**(1)	-	-	-
14	**862**(1)	-	-	-

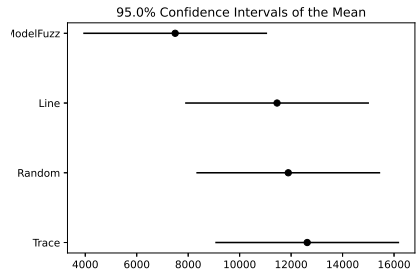
Table 5.4: The number of RedisRaft tests for the first occurrences of the bugs using different guidance strategies.

ID	Random	Trace	Line
1	0.364	0.504	0.462
2	**0.641**	0.412	0.373
3	0.373	0.433	0.443
4	0.559	0.569	0.563
5	0.545	0.455	0.522
6	**1.000**	**1.000**	**1.000**
7	**0.750**	**1.000**	0.500
8	N/A	0.000	N/A
9	0.547	0.567	**0.611**
10	0.167	**1.000**	0.250
11	0.563	0.531	0.500
12	**1.000**	**1.000**	**1.000**
13	**1.000**	**1.000**	**1.000**
14	**1.000**	**1.000**	**1.000**

Table 5.5: Pairwise $\hat{A}_{12}$ statistic results against ModelFuzz.

Method	Mean ± std	CI_l	CI_u
ModelFuzz	7494 ± 5742	3922	11066
Line	11454 ± 7675	7881	15025
Random	11890 ± 7377	8318	15462
Trace	12626 ± 7520	9054	16198

(a)



(b)

Figure 5.6: (a) Mean, standard deviation, lower confidence interval (CI_l), and upper confidence interval values (CI_u) of the first iteration to find a bug for all approaches. (b) The Tukey HSD test plot for all approaches.

faster than all other guidance approaches, $\{2, 7, 9, 10\}$ statistically significantly faster than some of the approaches, and comparably faster for the remainder. While this indicates its ability to detect bugs faster, the $\hat{A}_{12}$ results do not draw clear conclusions on the statistical significance.

Moreover, we test for statistical significance across all bugs in the mean iterations to find the bugs. Since we compare more than two approaches and the number of iterations for all bugs are normal and homoscedastic, we use repeated measures ANOVA [Gir92] as omnibus tests and post-hoc Tukey HSD [Tuk49] for assessing overall significance. We reject the null hypothesis ($p = 0.008$) of the repeated measure ANOVA, as there is a statistically significant difference between the mean values of the approaches. Overall, the tests show that ModelFuzz statistically significantly detects bugs faster.

Figure 5.6a reports the mean, standard deviation, and confidence intervals for all approaches, and Figure 5.6b illustrates the Tukey HSD results for all approaches. We can observe that

the mean iteration to detect a bug for ModelFuzz lies outside of the other approaches. This suggests that the differences between ModelFuzz and all of the remaining approaches are statistically significant, whereas there is no significant difference between random exploration, line coverage, and trace coverage guidance. Note that we perform ANOVA and post-hoc Tukey HSD tests only for RedisRaft, as we analyze numerous bugs with which we could form a sample set.

5.5 Discussion and Perspectives

Fuzzing.

Coverage-guided fuzzing [BPR16; LS18; Zel+19; Man+21; Ba+22; Heu+22] has been extensively studied for test input generation for sequential programs. Recent fuzzing techniques target the generation of different types of program inputs [GHP20; DGZ21; SZ22] and improve the performance of the fuzzer [BMC20; STS23]. Extensions of American Fuzzy Lop (AFL) [Zal] such as AFLNet [PBR20], StateAFL [Nat22] test communication protocols by mutating structured message inputs guided by the states explored. Different from test input fuzzing, utilize the fuzzing approach to generate event schedules.

Fuzzing concurrent and distributed systems.

Fuzzing methods for multithreaded concurrency guide the tests by monitoring races and synchronization events [Sen08; WSG11; Yu+12], execution states caused by thread interleavings [Che+20], coverage of concurrent call pairs [Jia+22], and recently, using the reads-from relation between the memory access operations [Wol+]. These methods are designed for multithreaded programs, and they do not target distributed concurrency.

Recent testing techniques for distributed systems learn from the set of explored executions and adapt reinforcement learning or fuzzing approaches to incorporate feedback information into the generation of new tests. QL [Muk+20] uses reinforcement learning to guide the exploration to unexplored parts of the execution space to improve coverage. Evolutionary search-based testing of distributed systems [MOP23] direct the exploration toward specific parts of the search space defined by a fitness function. CrashFuzz [Gao+23] adopts coverage-guided fuzzing to inject faults into distributed system executions, i.e., it injects crashes or restarts using structural code coverage information as program feedback. Mallory [Men+23a] builds on Jepsen [Kin22] and leverages reinforcement learning to guide the test generation of the fuzzer and uses event timeline abstraction (close to traces) as the feedback information to guide the test generation. Different from structural code or trace guidance, ModelFuzz uses guidance from the abstract system model.

Model-based testing.

Model-based testing uses formal models (e.g., TLA+ specifications) to exhaustively enumerate system executions and enforce them on the implementation. Its applications include testing application programming interfaces [Art+13], fragments of HTTP protocol [LPZ21] and implementation of MongoDB [SDH20]. Protocol fuzzers DTLS-Fuzzer [Fit+22] and EDHOC-Fuzzer [ST23] use model learning to generate a state machine model of the protocol implementations which can be used for model-based testing. Recent work Mocket [Wan+23] adopts model-based testing to test distributed system implementations. Mocket

uses the paths in the model's state space graph as test cases, and it enforces the system under test to run the sequence of actions generated on the system's model on the corresponding states and actions in the implementation. To achieve that, it requires a heavy annotation and instrumentation of the system's source code to mark the variables and messages associated with the system's model variables and actions. Our evaluation does not empirically compare to Mocket since we do not have the annotations to map the source codes of the systems under test to the TLA+ specifications.

ModelFuzz conceptually differs from model-based testing, as it performs an unconstrained exploration of the implementation guided by the abstract model. Model-based testing generates test cases using the paths in the model, and hence, it does not cover parts of the implementation that are abstracted away in the model. In contrast, model-guided fuzzing explores the executions of the implementation, including those not captured by the model.

Chapter 6

Conclusion and Future Work

In this thesis, we contribute to the growing corpus of automated testing techniques to ensure reliability of distributed systems implementations. Our contributions derive intuitions from unit testing, reinforcement learning and fuzzing techniques to develop new algorithms for effective biased exploration and to bridge the gap between the model and the implementation.

Contributions

Distributed systems suffer from a state explosion due to many possible inter-leavings of messages exchanged. Furthermore, implementation details which are typically hidden from the protocol models exacerbate the problem. While many techniques and heuristics have been developed to traverse the state space effectively, very few approaches explore the direction of biased exploration aided by the developer.

We developed two new techniques - Netrix and WaypointRL - that allow a developer testing the implementation to direct the exploration toward specific parts.

In Chapter 3, we introduced Netrix which derives intuition from unit testing to develop a runtime and domain-specific language (DSL) to write unit tests for distributed systems. Netrix, while relying on an existing exploration technique such as PCTCP, allows the developer to write unit tests to bias the exploration. In effect, the developer operates on a spectrum. On one end, a skilled developer familiar with the protocol model and the implementation can craft explicit and detailed executions - using Netrix filters - and check for safety assertions. On the other end, the developer can rely completely on the underlying exploration for a fully automatic and unbiased exploration.

While the filters are easy to write, they require reasoning about the contents and order of specific messages or events in the execution. As a result, the filters can become cumbersome to write. To tackle the challenge, we introduced a syntactic notion of *filter distance* to measure the necessity of a filter in a unit test while relying on PCTCP as the underlying exploration algorithm.

Alternatively, to make it easier for the developer to bias exploration, we introduced WaypointRL in Chapter 4. WaypointRL relies on Reinforcement Learning to automatically

explore the state space and bias based on developer input. With WaypointRL, the developer defines a sequence of state predicates that lead to a target state space. Internally, the predicates determine rewards that automatically bias the exploration towards the target predicate. The predicates are high-level compared to Netrix filters and are easier to write.

WaypointRL uses BonusMaxRL which is based on theoretically optimal reward-free Q learning algorithm UCBZero. Our initial experiments revealed that UCBZero fails to perform in practice. Therefore, motivating the need for BonusMaxRL. We borrowed the idea of a decaying reward from UCBZero and optimized it for better performance for practical exploration of the state space of a distributed system.

We demonstrated the effectiveness of Netrix and WaypointRL on realistic and production-ready benchmarks to show that exploration can be effectively biased to uncover new bugs. Both Netrix and WaypointRL, require user input to bias exploration - filters for Netrix and waypoints for WaypointRL. We provided guidelines to derive user input from the protocol model. However, both techniques do not rely explicitly on the model to test the implementation.

In Chapter 5, we addressed this gap between the model and the implementation by developing a new fuzzing-based algorithm ModelFuzz which guides the test input generation using a traditional fuzzing approach while relying on the model state for coverage. We showed that existing notions of coverage are insufficient using real-world benchmarks. Furthermore, we showed that high coverage of the model translates to better bug-finding capabilities over the implementation.

Future directions

Throughout the research, we identify several interesting directions of research. We list them below:

- While writing filters for Netrix unit tests or waypoints for WaypointRL, it is clear that the inputs to these algorithms can be derived from the protocol model. One possible extension to this work would be to develop a methodology or synthesis procedure to write Netrix filters or WaypointRL waypoints given a protocol model. With such a procedure, we will be addressing the gap currently filled by ModelFuzz and other model based testing approaches.
- WaypointRL and BonusMaxRL are based on theoretically optimal algorithms. However, it remains unknown if the optimal policies translate to probabilistic guarantees on visiting any given state.
- ModelFuzz employs a random input generation mechanism. Given the effectiveness of WaypointRL, ModelFuzz could be improved by leveraging "learning" based algorithms such as reinforcement learning to better generate inputs to test.
- To guide the input generation of ModelFuzz, we rely on an explicit state model checker (TLC) for the models. However, it remains to be seen if a symbolic model checker in the style of Concolic testing [GKS05] would achieve better coverage.
- For a given concrete distributed system implementation, WaypointRL and ModelFuzz are effective under an abstraction of the state space. For example, with raft

implementations, we abstract away the term number of each process when presenting the state to RL or when measuring the coverage over abstract states. In general, one can ask - Is it possible to synthesize the abstraction for a protocol that ensures goal coverage with WaypointRL and meaningful state guidance with ModelFuzz?

Bibliography

[MW47] H. B. Mann and D. R. Whitney. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”. In: *The Annals of Mathematical Statistics* 18.1 (Mar. 1947), pp. 50–60. ISSN: 0003-4851. DOI: 10.1214/aoms/1177730491.

[Tuk49] John W Tukey. “Comparing individual means in the analysis of variance”. In: *Biometrics* (1949), pp. 99–114.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176.

[DLS84] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. “Consensus in the Presence of Partial Synchrony (Preliminary Version)”. In: *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*. Ed. by Tiko Kameda et al. ACM, 1984, pp. 103–118. URL: https://dl.acm.org/citation.cfm?id=1599406.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (1985), pp. 374–382. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121.

[Maz86] Antoni W. Mazurkiewicz. “Trace Theory”. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*. Ed. by Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 255. Lecture Notes in Computer Science. Springer, 1986, pp. 279–324. DOI: 10.1007/3-540-17906-2_30.

[Gir92] Ellen R Girden. *ANOVA: Repeated measures*. 84. sage, 1992.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. “Technical Note Q-Learning”. In: *Mach. Learn.* 8 (1992), pp. 279–292. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[Hol97] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521. URL: https://doi.org/10.1109/32.588521.

[PR97] Ronald Parr and Stuart Russell. “Reinforcement Learning with Hierarchies of Machines”. In: *Advances in Neural Information Processing Systems 10, [NIPS Conference, Denver, Colorado, USA, 1997]*. Ed. by Michael I. Jordan, Michael J. Kearns, and Sara A. Solla. The MIT Press, 1997, pp. 1043–1049. URL: http://papers.nips.cc/paper/1384-reinforcement-learning-with-hierarchies-of-machines.

[CL99] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 173–186. URL: https://dl.acm.org/citation.cfm?id=296824.

[SPS99] Richard S. Sutton, Doina Precup, and Satinder Singh. “Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning”. In: *Artif. Intell.* 112.1-2 (1999), pp. 181–211. DOI: 10.1016/S0004-3702(99)00052-1. URL: https://doi.org/10.1016/S0004-3702(99)00052-1.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA$^+$ Specifications”. In: *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME ’99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. Ed. by Laurence Pierre and Thomas Kropf. Vol. 1703. Lecture Notes in Computer Science. Springer, 1999, pp. 54–66. DOI: 10.1007/3-540-48153-2_6.

[Die00] Thomas G. Dietterich. “Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition”. In: *J. Artif. Intell. Res.* 13 (2000), pp. 227–303. DOI: 10.1613/JAIR.639. URL: https://doi.org/10.1613/jair.639.

[VD00] András Vargha and Harold D. Delaney. “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132. DOI: 10.3102/10769986025002101. eprint: https://doi.org/10.3102/10769986025002101.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[VRC06] Margus Veanes, Pritam Roy, and Colin Campbell. “Online Testing with Reinforcement Learning”. In: *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*. Ed. by Klaus Havelund et al. Vol. 4262. Lecture Notes in Computer Science. Springer, 2006, pp. 240–253. DOI: 10.1007/11940197_16. URL: https://doi.org/10.1007/11940197%5C_16.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[Sen08] Koushik Sen. “Race directed random testing of concurrent programs”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 11–21. DOI: 10.1145/1375581.1375584.

[Yan+09] Junfeng Yang et al. “MODIST: Transparent Model Checking of Unmodified Distributed Systems”. In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. Ed. by Jennifer Rexford and Emin Gün Sirer. USENIX Association, 2009, pp. 213–228.

[Bur+10] Sebastian Burckhardt et al. “A randomized scheduler with probabilistic guarantees of finding bugs”. In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*. Ed. by James C. Hoe and Vikram S. Adve. ACM, 2010, pp. 167–178. DOI: 10.1145/1736020.1736040.

[FP10] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. https://martinfowler.com/books/dsl.html. 2010.

[Lig11] Lightbend, Inc. *Akka Documentation*. 2011. URL: https://akka.io/docs/.

[SBG11] Jirí Simsa, Randy Bryant, and Garth A. Gibson. “dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems”. In: *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*. Ed. by Alex Groce and Madanlal Musuvathi. Vol. 6823. Lecture Notes in Computer Science. Springer, 2011, pp. 188–193. DOI: 10.1007/978-3-642-22306-8_14.

[WSG11] Chao Wang, Mahmoud Said, and Aarti Gupta. “Coverage guided systematic concurrency testing”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ACM, 2011, pp. 221–230. DOI: 10.1145/1985793.1985824.

[Del12] Pamela Delgado. “DISTAL: Domain-specific Language for Implementing Distributed Algorithms”. In: (2012). URL: http://infoscience.epfl.ch/record/187164.

[SB12] João Sousa and Alysson Neves Bessani. “From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation”. In: *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*. Ed. by Cristian Constantinescu and Miguel P. Correia. IEEE Computer Society, 2012, pp. 37–48. DOI: 10.1109/EDCC.2012.32. URL: https://doi.org/10.1109/EDCC.2012.32.

[Yu+12] Jie Yu et al. “Maple: a coverage-driven testing tool for multithreaded programs”. In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 485–502. DOI: 10.1145/2384616.2384651.

[Art+13] Cyrille Valentin Artho et al. “Modbat: A Model-Based API Tester for Event-Driven Systems”. In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. Ed. by Valeria Bertacco and Axel Legay. Vol. 8244. Lecture Notes in Computer Science. Springer, 2013, pp. 112–128. DOI: 10.1007/978-3-319-03077-7_8. URL: https://doi.org/10.1007/978-3-319-03077-7%5C_8.

[Des+13] Ankush Desai et al. “P: safe asynchronous event-driven programming”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 321–332. DOI: 10.1145/2491956.2462184.

[Elm+13] Tayfun Elmas et al. “CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 153–164. ISBN: 9781450320146. DOI: 10.1145/2491956.2462162. URL: https://doi.org/10.1145/2491956.2462162.

[AB14] Andrea Arcuri and Lionel C. Briand. “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering”. In: *Softw. Test. Verification Reliab.* 24.3 (2014), pp. 219–250. DOI: 10.1002/STVR.1486.

[BSA14] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. “State Machine Replication for the Masses with BFT-SMART”. In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43. URL: https://doi.org/10.1109/DSN.2014.43.

[Lee+14] Tanakorn Leesatapornwongsa et al. “SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems”. In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 399–414.

[OO14] Diego Ongaro and John K. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference, USENIX ATC ’14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319.

[Del+15a] Pantazis Deligiannis et al. “Asynchronous Programming, Analysis and Testing with State Machines”. In: *SIGPLAN Not.* 50.6 (2015), pp. 154–164. ISSN: 0362-1340. DOI: 10.1145/2813885.2737996. URL: https://doi.org/10.1145/2813885.2737996.

[Del+15b] Pantazis Deligiannis et al. “Asynchronous programming, analysis and testing with state machines”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 154–164. DOI: 10.1145/2737924.2737996.

[Haw+15] Chris Hawblitzel et al. “IronFleet: proving practical distributed systems correct”. In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 1–17. DOI: 10.1145/2815400.2815428. URL: https://doi.org/10.1145/2815400.2815428.

[New+15] Chris Newcombe et al. “How Amazon web services uses formal methods”. In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: 10.1145/2699417.

[She15] Justin Sheehy. “There is No Now”. In: *ACM Queue* 13.3 (2015), p. 20. DOI: 10.1145/2742694.2745385. URL: https://doi.org/10.1145/2742694.2745385.

[Wil+15] James R. Wilcox et al. “Verdi: a framework for implementing and formally verifying distributed systems”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 357–368. DOI: 10.1145/2737924.2737958. URL: https://doi.org/10.1145/2737924.2737958.

[BPR16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based Greybox Fuzzing as Markov Chain”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 1032–1043. DOI: 10.1145/2976749.2978428.

[CMN16] Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. “Hitting Families of Schedules for Asynchronous Programs”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 157–176. DOI: 10.1007/978-3-319-41540-6_9.

[Pad+16] Oded Padon et al. “Ivy: safety verification by interactive generalization”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 614–630. DOI: 10.1145/2908080.2908118. URL: https://doi.org/10.1145/2908080.2908118.

[Fon+17] Pedro Fonseca et al. “An Empirical Study on the Correctness of Formally Verified Distributed Systems”. In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 328–343. DOI: 10.1145/3064176.3064183. URL: https://doi.org/10.1145/3064176.3064183.

[BGS18] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. “Deep Reinforcement Fuzzing”. In: *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*. IEEE Computer Society, 2018, pp. 116–122. DOI: 10.1109/SPW.2018.00026. URL: https://doi.org/10.1109/SPW.2018.00026.

[BKM18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. “The latest gossip on BFT consensus”. In: *CoRR* abs/1807.04938 (2018). arXiv: 1807.04938. URL: http://arxiv.org/abs/1807.04938.

[Des+18a] Ankush Desai et al. “Compositional Programming and Testing of Dynamic Distributed Systems”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276529. URL: https://doi.org/10.1145/3276529.

[Des+18b] Ankush Desai et al. “Compositional Programming and Testing of Dynamic Distributed Systems”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276529. URL: https://doi.org/10.1145/3276529.

[Gao+18] Yu Gao et al. “An empirical study on crash recovery bugs in large-scale distributed systems”. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 539–550. DOI: 10.1145/3236024.3236030. URL: https://doi.org/10.1145/3236024.3236030.

[Kul+18] Burcu Kulahcioglu Ozkan et al. “Randomized testing of distributed systems with probabilistic guarantees”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 160:1–160:28. DOI: 10.1145/3276530.

[LS18] Caroline Lemieux and Koushik Sen. “FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 475–485. DOI: 10.1145/3238147.3238176.

[MN18] Rupak Majumdar and Filip Niksic. “Why is random testing effective for partition tolerance bugs?” In: *Proc. ACM Program. Lang.* 2.POPL (2018), 46:1–46:24. DOI: 10.1145/3158134.

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. “Programming and proving with distributed protocols”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 28:1–28:30. DOI: 10.1145/3158116. URL: https://doi.org/10.1145/3158116.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. The MIT Press, 2018.

[KMO19] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. “Trace aware random testing for distributed systems”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 180:1–180:29. DOI: 10.1145/3360606.

[KLR19] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. “The TLA+ Toolbox”. In: *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*. Ed. by Rosemary Monahan, Virgile Prevosto, and José Proença. Vol. 310. EPTCS. 2019, pp. 50–62. DOI: 10.4204/EPTCS.310.6.

[Le+19] Xuan-Bach Dinh Le et al. “Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis”. In: *ACM SIGSOFT Softw. Eng. Notes* 44.4 (2019), p. 14. DOI: 10.1145/3364452.3364455.

[Pad+19] Rohan Padhye et al. “Semantic fuzzing with zest”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. Ed. by Dongmei Zhang and Anders Møller. ACM, 2019, pp. 329–340. DOI: 10.1145/3293882.3330576.

[XT19] Zhe Xu and Ufuk Topcu. “Transfer of Temporal Logic Formulas in Reinforcement Learning”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 4010–4018. DOI: 10.24963/IJCAI.2019/557. URL: https://doi.org/10.24963/ijcai.2019/557.

[Zel+19] Andreas Zeller et al. *The fuzzing book*. 2019.

[AK20] Peter Alvaro and Kyle Kingsbury. “Elle: Inferring Isolation Anomalies from Experimental Observations”. In: *Proc. VLDB Endow.* 14.3 (2020), pp. 268–280. DOI: 10.5555/3430915.3442427.

[BMC20] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. “Boosting fuzzer efficiency: an information theoretic perspective”. In: *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 678–689. DOI: 10.1145/3368089.3409748.

[Che+20] Hongxu Chen et al. “MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs”. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 2325–2342. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu.

[Dra+20] Cezara Dragoi et al. “Testing consensus implementations using communication closure”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 210:1–210:29. DOI: 10.1145/3428278.

[GHP20] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. “Intelligent REST API data fuzzing”. In: *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 725–736. DOI: 10.1145/3368089.3409719.

[GMZ20] Rahul Gopinath, Björn Mathis, and Andreas Zeller. “Mining input grammars from dynamic control flow”. In: *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 172–183. DOI: 10.1145/3368089.3409679.

[Jin+20] Chi Jin et al. “Reward-Free Exploration for Reinforcement Learning”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 4870–4879. URL: http://proceedings.mlr.press/v119/jin20d.html.

[Muk+20] Suvam Mukherjee et al. “Learning-based controlled concurrency testing”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 230:1–230:31. DOI: 10.1145/3428298.

[PBR20] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. “AFLNET: A Greybox Fuzzer for Network Protocols”. In: *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 460–465. DOI: 10.1109/ICST46399.2020.00062.

[Red+20] Sameer Reddy et al. “Quickly generating diverse valid test inputs with reinforcement learning”. In: *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 1410–1421. DOI: 10.1145/3377811.3380399. URL: https://doi.org/10.1145/3377811.3380399.

[SDH20] Judah Schvimer, A. Jesse Jiryu Davis, and Max Hirschhorn. “eXtreme Modelling in Practice”. In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1346–1358. DOI: 10.14778/3397230.3397233.

[ZMS20] Xuezhou Zhang, Yuzhe Ma, and Adish Singla. "Task-agnostic Exploration in Reinforcement Learning". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/8763d72bba4a7ade23f9ae1f09f4efc7-Abstract.html.

[Bor+21] James Bornholt et al. "Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3". In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. Ed. by Robbert van Renesse and Nickolai Zeldovich. ACM, 2021, pp. 836–850. DOI: 10.1145/3477132.3483540.

[Cha+21] Tej Chajed et al. "GoJournal: a verified, concurrent, crash-safe journaling system". In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. Ed. by Angela Demke Brown and Jay R. Lorch. USENIX Association, 2021, pp. 423–439. URL: https://www.usenix.org/conference/osdi21/presentation/chajed.

[Del+21] Pantazis Deligiannis et al. "Building Reliable Cloud Services Using Coyote Actors". In: *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. Ed. by Carlo Curino, Georgia Koutrika, and Ravi Netravali. ACM, 2021, pp. 108–121. DOI: 10.1145/3472883.3486983.

[DGZ21] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. "FormatFuzzer: Effective Fuzzing of Binary File Formats". In: *CoRR* abs/2109.11277 (2021). arXiv: 2109.11277.

[Jab+21] Nouraldin Jaber et al. "QuickSilver: modeling and parameterized verification for distributed agreement-based systems". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–31. DOI: 10.1145/3485534. URL: https://doi.org/10.1145/3485534.

[Jia+21] Yuqian Jiang et al. "Temporal-Logic-Based Reward Shaping for Continuing Reinforcement Learning Tasks". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 7995–8003. DOI: 10.1609/AAAI.V35I9.16975. URL: https://doi.org/10.1609/aaai.v35i9.16975.

[Jot+21] Kishor Jothimurugan et al. "Compositional Reinforcement Learning from Logical Specifications". In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc'Aurelio Ranzato et al. 2021, pp. 10026–10039. URL: https://proceedings.neurips.cc/paper/2021/hash/531db99cb00833bcd414459069dc7387-Abstract.html.

[LPZ21] Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. “Model-based testing of networked applications”. In: *ISSTA ’21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*. Ed. by Cristian Cadar and Xiangyu Zhang. ACM, 2021, pp. 529–539. DOI: 10.1145/3460319.3464798.

[Man+21] Valentin J. M. Manès et al. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[Wan+21] Daimeng Wang et al. “SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael D. Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 2741–2758. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng.

[Zha+21] Shaohua Zhang et al. “FIGCPS: Effective Failure-inducing Input Generation for Cyber-Physical Systems with Deep Reinforcement Learning”. In: *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 555–567. DOI: 10.1109/ASE51524.2021.9678832. URL: https://doi.org/10.1109/ASE51524.2021.9678832.

[Alu+22] Rajeev Alur et al. “A Framework for Transforming Specifications in Reinforcement Learning”. In: *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Ed. by Jean-François Raskin et al. Vol. 13660. Lecture Notes in Computer Science. Springer, 2022, pp. 604–624. DOI: 10.1007/978-3-031-22337-2_29. URL: https://doi.org/10.1007/978-3-031-22337-2%5C_29.

[Ba+22] Jinsheng Ba et al. “Stateful Greybox Fuzzing”. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 3255–3272.

[BSM22] Marcel Böhme, László Szekeres, and Jonathan Metzman. “On the Reliability of Coverage-Based Fuzzer Benchmarking”. In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1621–1633. DOI: 10.1145/3510003.3510230.

[Fit+22] Paul Fiterau-Brostean et al. “DTLS-Fuzzer: A DTLS Protocol State Fuzzer”. In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 2022, pp. 456–458. DOI: 10.1109/ICST53961.2022.00051.

[Heu+22] Marc Heuse et al. *AFL++*. Version 4.00c. Jan. 2022. URL: https://github.com/AFLplusplus/AFLplusplus.

[Ica+22] Rodrigo Toro Icarte et al. “Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning”. In: *J. Artif. Intell. Res.* 73 (2022), pp. 173–208. DOI: 10.1613/JAIR.1.12440. URL: https://doi.org/10.1613/jair.1.12440.

[Jia+22] Zu-Ming Jiang et al. “Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection”. In: *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[Kin22] Kyle Kingsbury. *Jepsen*. http://jepsen.io/. 2022.

[Nat22] Roberto Natella. “StateAFL: Greybox fuzzing for stateful network servers”. In: *Empir. Softw. Eng.* 27.7 (2022), p. 191. DOI: 10.1007/s10664-022-10233-3.

[SZ22] Dominic Steinhöfel and Andreas Zeller. “Input invariants”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 583–594. DOI: 10.1145/3540250.3549139.

[Gao+23] Yu Gao et al. “Coverage Guided Fault Injection for Cloud Systems”. In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2211–2223. DOI: 10.1109/ICSE48619.2023.00186.

[MOP23] Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. “Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm”. In: *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 36–47. DOI: 10.1109/ICSE-SEIP58684.2023.00009.

[Men+23a] Ruijie Meng et al. “Distributed System Fuzzing”. In: *CoRR* abs/2305.02601 (2023). DOI: 10.48550/arXiv.2305.02601. arXiv: 2305.02601.

[Men+23b] Ruijie Meng et al. “Greybox Fuzzing of Distributed Systems”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng et al. ACM, 2023, pp. 1615–1629. DOI: 10.1145/3576915.3623097.

[PDG23] Lauren Pick, Ankush Desai, and Aarti Gupta. “Psym: Efficient Symbolic Exploration of Distributed Systems”. In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 660–685. DOI: 10.1145/3591247.

[ST23] Konstantinos Sagonas and Thanasis Typaldos. “EDHOC-Fuzzer: An EDHOC Protocol State Fuzzer”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. Ed. by René Just and Gordon Fraser. ACM, 2023, pp. 1495–1498. DOI: 10.1145/3597926.3604922.

[Sha+23] Upamanyu Sharma et al. “Grove: a Separation-Logic Library for Verifying Distributed Systems”. In: *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. Ed. by Jason Flinn et al. ACM, 2023, pp. 113–129. DOI: 10.1145/3600006.3613172. URL: https://doi.org/10.1145/3600006.3613172.

[STS23] Chaofan Shou, Shangyin Tan, and Koushik Sen. "ItyFuzz: Snapshot-Based Fuzzer for Smart Contract". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. Ed. by René Just and Gordon Fraser. ACM, 2023, pp. 322–333. DOI: 10.1145/3597926.3598059.

[Wan+23] Dong Wang et al. "Model Checking Guided Testing for Distributed Systems". In: *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. Ed. by Giuseppe Antonio Di Luna et al. ACM, 2023, pp. 127–143. DOI: 10.1145/3552326.3587442.

[Win+23] Levin N. Winter et al. "Randomized Testing of Byzantine Fault Tolerant Algorithms". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 757–788. DOI: 10.1145/3586053.

[Bor+24] James Bornholt et al. *awslabs/shuttle*. Mar. 29, 2024. URL: https://github.com/awslabs/shuttle.

[Kov+24] Nikita Koval et al. *JetBrains/lincheck*. Apr. 24, 2024. URL: https://github.com/JetBrains/lincheck.

[Git] Github. *TLA+ specification for the Raft consensus algorithm*. https://github.com/ongardie/raft.tla.

[Wol+] Dylan Wolff et al. "Greybox Fuzzing for Concurrency Testing". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2024, San Diego, USA, April 27 - May 1, 2024*. to appear.

[Zal] Michal Zalewski. *American fuzzy lop*. http://lcamtuf.coredump.cx/afl/.