Rocket: A System-Level Fuzz-Testing Framework for the XRPL Consensus Algorithm

Wishaal Kanhai*, Ivar van Loon*, Yuraj Mangalgi*, Thijs van der Valk*, Lucas Witte*,
Annibale Panichella†, Mitchell Olsthoorn†, and Burcu Kulahcioglu Ozkan†
Delft University of Technology, Delft, The Netherlands
*Email: {W.R.Kanhai, I.S.vanLoon, Y.Mangalgi, T.C.J.vanderValk, L.C.Witte}@student.tudelft.nl
†Email: {A.Panichella, M.J.G.Olsthoorn, B.Ozkan}@tudelft.nl

2025 IEEE Conference on Software Testing, Verification and Validation (ICST) | 979-8-3315-0814-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICST62969.2025.10988979

***Abstract*—Byzantine fault tolerant algorithms are critical for achieving consistency and reliability in distributed systems, especially in the presence of faults or adversarial behavior. The consensus algorithm used by the XRP Ledger falls into this category. In practice, the implementation of these algorithms is prone to errors, which can lead to undesired behavior in the system. This paper introduces Rocket, a fuzz-testing framework designed for the XRPL consensus algorithm. Rocket enables researchers and developers to automatically inject network and process faults into a locally simulated network of XRPL validator nodes to test if the system behaves as expected. This technique has previously been shown to be effective in finding implementation errors. Rocket has been designed to focus on extensibility and ease of use, enabling users to run complex test scenarios with minimal setup. Video: https://www.youtube.com/watch?v=O7Z3ufRa51Y**

***Index Terms*—Byzantine fault-tolerance, Consensus algorithms, XRP Ledger**

I. Introduction

Byzantine fault tolerant algorithms, often referred to as protocols, are essential for enabling distributed systems to reach an agreement on a correct value, even when a subset of processes behaves maliciously or unpredictably [1]. The XRP Ledger (XRPL) consensus algorithm [2], [3], which is the foundation of the XRPL and its XRP cryptocurrency, is designed to be Byzantine fault tolerant (BFT). XRPL is an enterprise global payment network with millions of transactions per day across 40+ countries[1]. While the protocol is theoretically robust, its practical implementation might contain errors [4], which can result in vulnerabilities within the XRPL network. Such errors can cause the XRPL network to deviate from its intended consensus behavior under certain conditions. For example, attackers might validate invalid transactions or disrupt the network's progress entirely. Ensuring consistent and reliable outputs in these critical systems is imperative.

The XRPL comprises a network of validator nodes, each having a copy of the ledger history. The validators run the XRPL consensus protocol to agree on which transactions to commit in the decentralized ledger. This protocol operates in multiple phases: validators collect and propose a set of transactions during the open phase. In the proposal round, each validator sends its set of proposed transactions to the other validators in the network. The validators can add or remove certain transactions from their ledger based on the proposal sets of the other validators. Once a predefined agreement on the transaction set is met, the protocol continues with the validation round, where validators finalize and exchange the proposed ledger. The ledger is validated if consensus is achieved with a sufficient quorum; otherwise, an empty ledger is generated.

Given the complexity of the XRPL consensus algorithm and the variety of possible adverse execution scenarios, rigorous and thorough testing is critical. Seemingly small implementation errors may lead to catastrophic results, such as validating a malicious transaction in the existence of certain network and timing configurations. Therefore, it is crucial to test whether the system behaves as expected under adverse execution scenarios. Manual testing is particularly difficult and time-consuming [3], [5], [6] for complex systems like XRPL due to the intricacy of the consensus process. This process comprises concurrent message exchanges, strict time constraints, and fault-tolerant design, making exploring all possible corner-case scenarios challenging. Various automated testing frameworks for distributed systems have been proposed in the literature [7]–[11]. However, they do not support the network and process behaviors and trust configurations of XPRL and cannot be extended by implementing different testing algorithms (e.g., evolutionary and randomized algorithms).

To address these challenges, this paper introduces Rocket, an extensible fuzz-testing framework for the XRPL consensus algorithm. The Rocket framework allows researchers to design, implement, and benchmark testing algorithms on a dedicated, local XRPL network where its topology can be configured dynamically. Our tool generates test scenarios, including controlling message concurrency, network fault injection (e.g., delaying a message), and process fault injection. Rocket implements a network interception layer, which supports various features that can be used to generate test cases. Regarding process faults, it supports both benign and Byzantine faults. It injects benign faults by crashing or restarting a process and injects Byzantine faults by mutating the content of the messages sent over the network. Overall, Rocket offers system-level fuzz-testing functionality, where each test case executes the protocol on a cluster of processes with specific concurrency scenarios of message delivery, network, and process faults. It also automates the setup and teardown of the cluster, ensuring each test begins with a clean network environment.

[1]https://xrpl.org/

Rocket is designed with extensibility and adaptability in mind. It supports the integration of new testing algorithms, such as evolutionary concurrency exploration algorithms [12] for the XRP Ledger, and the ByzzFuzz algorithm [4] for testing Byzantine fault tolerant algorithms, which have been shown to be successful at discovering errors using a fuzzing approach. Moreover, the architectural design of the Rocket framework involves two separate modules, namely the interceptor and the controller, decoupling the execution of the network of validators and the execution of the fuzz tester for scenario generation. The interceptor manages the execution of the network of XRPL validator nodes. The controller decides how to handle (e.g., drop, delay, modify) the messages distributed on the network. This module contains the logic required to construct automated test case scenarios and contains the algorithms for creating and mutating testing scenarios.

Rocket can be used by researchers who are interested in designing more effective and efficient testing and fuzzing algorithms, as well as developers interested in validating and thoroughly testing the implementation of their protocols. Although Rocket focuses on the XRPL, its modular architecture allows testers and researchers to extend it to other blockchain and distributed systems by implementing new interceptors.

II. The Rocket Testing Framework

Rocket is a system-level fuzz testing framework designed to explore different concurrency and fault scenarios, specifically built for the XRPL consensus algorithm. The implementation of our tool is publicly available on GitHub[2] and Zenodo [13]. This section covers Rocket's architecture (II-A), outlines its testing approach for the XRPL consensus algorithm (II-B), describes its workflow (II-C), highlights its key features (II-D), and provides extension points (II-E).

A. Rocket's Architecture

Figure 1 illustrates the high-level architecture of Rocket, which consists of two main components: the *network interceptor* and the *controller*. The *network interceptor* intercepts the messages exchanged between the validators in the network, forwarding them to the *controller* for processing. The *controller* decides the actions to take based on the testing algorithm and returns the message (potentially after mutating them to simulate malicious behaviors) to the network interceptor. During test execution, the *controller* collects an enriched set of operation logs, including the exchanged messages and network and process faults of an execution. At the end of the test execution, the collected logs are analyzed by the *specification checker* module to check the correctness properties of the consensus. This modular architecture separates the responsibilities of configuring and managing the distributed network under test (handled by the interceptor) from the logic for fuzzing and test generation (handled by the controller). This architecture allows us to implement additional interceptors for different distributed systems (e.g., Ethereum) in the future without re-implementing the fuzzers and test generation logic.

[2]https://github.com/diseb-lab/rocket

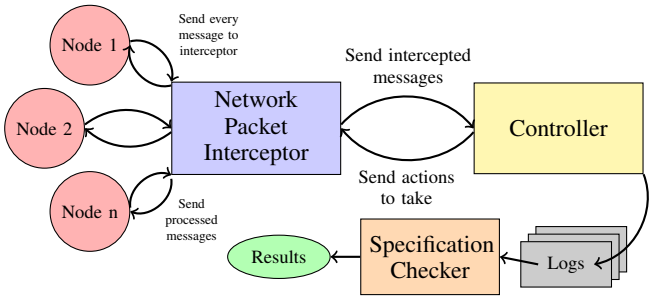

Fig. 1. Rocket tool architecture

As the XRPL consensus has strict time requirements for its different phases, the low latency of message delivery between the modules and the high throughput of the messages are crucial not to disrupt the network's behavior. Rocket uses Remote Procedure Calls (RPC) for communication between the interceptor and controller components, as it offers low latency and high throughput for small messages. We developed the interceptor module using `Rust` and the controller module in `Python`. `Rust` is particularly suitable for low-level network tasks [14], while `Python` has an extensive collection of machine learning and optimization libraries (e.g., evolutionary algorithms [15]). Such libraries are essential for optimizing users' algorithmic strategies.

B. Approach to Testing the XRPL Consensus Algorithm

The consensus algorithm of the XRP Ledger guarantees that all nodes in the underlying distributed system agree on an identical set of validated transactions in a *ledger*, even in the presence of faulty or malicious validators. Although network faults may temporarily isolate nodes or partition the network, the protocol guarantees that nodes synchronize and agree on the same sequence of transactions and ledger states once the network recovers.

Unlike traditional distributed consensus algorithms or other blockchain protocols, the XRPL consensus protocol offers a unique approach to trust, supporting subjective, asymmetric trust relationships [16]. While in traditional networks, nodes process messages and votes from all other nodes to achieve consensus, the XRPL consensus protocol allows validator nodes to specify a trusted subset of validators, known as the Unique Node List (UNL). During consensus, the validators consider only the messages from nodes in their UNL.

In each XRPL consensus proposal round, the validators propose their version of the ledger to the other validators. A ledger version contains the current state of all balances and objects stored in the ledger, the set of new transactions compared to the previous ledger, and additional metadata, such as a ledger index. Eventually, through consensus, validators finalize and store a new ledger version once at least eighty percent of them agree on its set of transactions.

A node is considered *faulty* if it does not behave accordingly to the consensus algorithm, regardless of its intentions. On the other hand, a node is considered *correct* if it consistently adheres to the protocol and behaves without error. The consensus

halts when more than twenty, but less than eighty percent of the validators are faulty [17], [18].

To fuzz-test the XRPL consensus algorithm, Rocket simulates attacks (faulty or malicious behaviors of the network and processes) on a local XRPL network. Rocket can simulate both network and process faults through its actions on the intercepted messages. Network faults are simulated by delaying or dropping messages. Our tool can also simulate dynamic partitioning of the network, disconnecting node pairs, and creating partitions of validator nodes that can only communicate with each other. Such conditions are particularly challenging for distributed systems [7], [19], as they can lead to split-brain scenarios, where partitions make conflicting decisions. Recovering from the partitions requires a correct logic of synchronization with the rest of the network. Rocket's ability to combine multiple faults allows for larger-scale testing using approaches like evolutionary-based testing [12], which has proven highly effective against consensus algorithms.

Rocket also simulates process faults, including benign crashes and malicious behaviors, by mutating intercepted messages. Mutating messages alters their contents and, with that, their purposes. Mutations can be as simple as altering arbitrary bits in the messages or high-level semantic modifications, such as altering protocol fields with syntactically valid but semantically invalid content. For example, strategic mutations —like reusing previously sent messages or corrupting critical fields— have been shown to uncover subtle vulnerabilities in protocol implementations [4].

After executing each test, Rocket checks the correctness properties of consensus protocols [20]:

1) *Termination*: Each correct process in the network decides on a value eventually.
2) *Validity*: Correct processes in the network may only decide values that were proposed by other correct processes.
3) *Integrity*: No correct process in the network decides on a value more than once at a time.
4) *Agreement*: Correct processes in the network decide identically.

Rocket verifies the safety properties of agreement, validity, integrity and bounded termination of consensus. To verify these properties, Rocket fetches the ledger info at each node after every validated ledger. To verify the termination property, it will check if the ledger sequence reached its user-configured goal ledger sequence. To verify agreement, it checks that all validators have identical ledger hashes and indexes.

C. Rocket's Workflow

Rocket's workflow, illustrated in Figure 2, comprises three phases: *initialization*, *execution*, and *evaluation*. This workflow represents a single test iteration (e.g., execution of a test case), with the number of iterations configurable by the user.

During the *initialization phase*, the interceptor and the controller modules first establish a gRPC connection for communication. The interceptor then sets up the XRPL network with

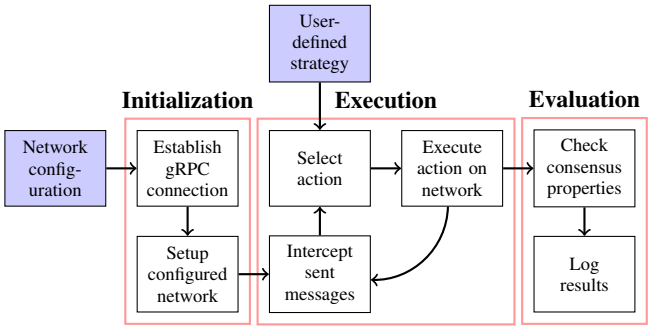

Fig. 2. Rocket tool workflow

the user-specified configuration of validator nodes, preparing the environment for testing.

Once initialization is completed, the *execution phase* starts. The interceptor captures all messages sent between the validator nodes and forwards them to the controller. The controller then takes action on each intercepted message according to the *user-defined testing strategy*, e.g., a fuzz testing algorithm. The testing algorithm steers the execution by selecting the actions to run in the execution. Rocket offers functionality to perform the following actions for each message:

1) *Drop*: The interceptor does not forward the message to the designated node.
2) *Delay*: The interceptor will deliver the message to the designated node after a specified delay.
3) *Send*: The interceptor delivers the message to the designated node immediately, simulating a well-functioning network.
4) *Duplicate*: The interceptor sends multiple copies of the message to the designated node.
5) *Mutate*: The interceptor mutates the message contents before sending it to the designated node.

Actions like *duplicate* and *mutate* simulate process faults, while message interception allows the controller to inject transactions into the network. The *execution phase* ends when the user-defined stopping condition is met, such as a target number of validated ledgers or a maximum testing duration.

In the *evaluation phase*, Rocket verifies the bounded termination and agreement properties, as described in Section II-B.

At the end of each iteration, Rocket resets the entire network state, terminating all validator nodes. A new test iteration then starts with a fresh network setup, ensuring complete isolation between test iterations and preventing carry-over effects from previous runs. For tests configured with multiple iterations, Rocket aggregates the evaluation results across all iterations to provide a comprehensive report.

D. Key Features

Network Partitioning: Rocket enables the simulation of network faults by dynamically partitioning the XRPL network. This is achieved by disconnecting specific node pairs or injecting artificial delays into their communication. Users can define connectivity rules, specifying which nodes should remain connected and allowing the controller module to reconfigure the

network topology during execution. This capability is essential for creating controlled fault scenarios that test the resilience of the XRPL network under degraded communication, partial outages, or other adverse conditions.

Processing Intercepted Messages: Rocket simplifies handling intercepted messages through a streamlined pipeline. Users can easily decode raw byte messages into structured types, such as `TMProposeSet` (ledger proposals) or `TMValidation` (validations), using Ripple's official Protocol Buffer definitions. Within this pipeline, users can analyze the message type, modify its contents, and decide on the appropriate action. Rocket also provides methods for signing and encoding messages after modification. At the end of the pipeline, the potentially mutated message and its corresponding action are returned to the interceptor for execution.

Enriched Execution Logging: Rocket incorporates a logging system for traceability, reproducibility, and analysis of test execution results. The logging framework categorizes data into *iteration* and *test-wide* logs. *Iteration logs* are generated in each test iteration and are divided into three categories:

1) *Action Log*: Records detailed information about all intercepted messages, including their source and destination node IDs and the actions performed.
2) *Node-Info Log*: Contains the configuration details of all validator nodes, documenting their initial state for each iteration.
3) *Ledger Results Log*: Captures the ledger information for all nodes after each validated ledger, providing insights into the consensus process and final outcomes.

In addition to the iteration logs, Rocket generates *test-wide logs* that span the entire test case:

1) *Spec-Check Log*: Records the results of the termination and agreement property checks, indicating whether the properties passed or failed for each iteration.
2) *Aggregated Spec-Check Log*: Provides a comprehensive summary of all spec-check results across iterations in a JSON format, facilitating high-level analysis.

Rocket's logging mechanism captures detailed logs of the test executions and stores them in a structured format. Specifically, it logs each exchanged or dropped message in the network along with their timestamps, maintains the logs for each node in the network, and records the operations on the ledger. This enables users to conduct detailed post-test analyses and evaluate the behavior of the XRPL consensus algorithm. There is currently no built-in functionality to reproduce a test execution. However, the detailed logs provide sufficient information for users to replicate previous test scenarios. Our tool can also be extended to add strategies or implement additional functionality for test reproduction and replay.

Flexible Configuration Options: Rocket offers a highly flexible configuration system, allowing users to tailor the network setup and fuzzing strategies to their testing needs. The tool provides two configuration files: one for network settings and another for strategy parameters. The *network configuration* allows users to define the XRPL network topology and nodes:

1) *Number of Nodes*: Users can specify the number of validator nodes to simulate.
2) *Port Assignments*: Users can define the ports on which the validators are hosted.
3) *Network Partitions*: This enables users to configure custom network topologies, specifying how nodes are grouped and connected.
4) *Nodes' UNLs*: This allows users to define a UNL of trusted validators for each validator.

The *strategy configuration* defines the parameters for the fuzzing strategies used during execution. For user-defined strategies, users can specify the test parameters specific to their testing algorithms. For example, they can set probability values for certain actions or specify ranges for message delays.

To enhance usability, Rocket supports configuration overrides through the command-line interface. Users can override default settings without modifying the base configuration files, making it easier to experiment with different setups or strategies during development.

E. Extension Points

Rocket offers a robust foundation for fuzz-testing the XRPL consensus algorithm, with several extension points that developers can explore to further enhance its capabilities:

Processing delays: Every intercepted message incurs a natural processing delay. For simple strategies, such as selecting an action naively at random, the delay is negligible. However, computationally intensive strategies, especially those involving machine learning, may increase message latency, which developers should account for when designing such strategies.

Message drops as delays: Rocket simulates message drops by introducing delays of 30 seconds. Developers should be aware that if an iteration runs longer than 30 seconds, these "drops" might still result in delayed messages rather than true drops. This design choice avoids errors in validator nodes that would otherwise halt execution.

Dynamic partitioning: Dynamic partitioning in Rocket operates by simulating disconnections through message drops rather than physically severing links. While reconnections can be simulated by ceasing message drops, new connections between validators initially configured as disconnected cannot be established dynamically. This behavior could be extended by introducing dynamic topology reconfiguration, enabling more flexible partitioning scenarios during runtime.

Distributed systems support: Currently, Rocket exclusively focuses on the XRPL; however, our modular architecture allows developers to extend Rocket to other distributed systems by switching the interceptor that is used.

III. Validation

The effectiveness of Rocket's key features in discovering erroneous executions of the XRPL consensus protocol has been validated by recently published research that discovered buggy executions in the existence of message delays [12] and Byzantine process faults [4]. Rocket supports the development of these algorithms with its support for message delays,

reorderings, network partitions, and injecting Byzantine faults by processing intercepted messages in a flexible set of configurations network configurations and trust assumptions.

The evolutionary approach for concurrency testing of the XRPL consensus algorithm [12] highlights the need for generating test cases that delay the delivery of certain protocol messages to uncover some problematic executions. The work has shown that blockchain systems such as XRPL are prone to concurrency bugs that manifest under message delays and reorderings that do not match the developer's assumptions about message delivery and trigger consensus violations.

The ByzzFuzz [4] randomized testing algorithm demonstrates that stimulating Byzantine process faults by mutating the content of the correct protocol messages is effective in discovering Byzantine fault tolerance bugs in blockchain implementations. It demonstrated that injecting Byzantine faults through small-scale message mutations and network partitions can create subtle execution scenarios that might be overlooked during protocol design or implementation, leading to erroneous executions. These errors may arise from insufficient trust assumptions in the network's UNLs as well as from implementation bugs, such as a recently discovered critical bug that was quickly fixed after its discovery.

We provide Rocket as an open-source framework that can be extended with other testing algorithms to explore the executions of the XRPL consensus algorithm. Users can extend Rocket by implementing new complex algorithms or extending the framework with more utility methods.

IV. Conclusion and Future Work

This paper introduced Rocket, a system-level fuzz testing framework specifically built for testing the XRPL consensus algorithm. Rocket addresses the need for an extensible and practical fuzz-testing framework for the XRPL that supports key features demonstrated to discover erroneous executions in blockchains. Moreover, it is equipped with flexible configuration and enriched logging functionalities to explore various configuration settings and assist in diagnosing test executions.

Rocket is designed for extensibility, practicality, and quality, enabling a rigorous testing framework of the XRP Ledger. Rocket currently provides an in-built blackbox random fuzzer, which can be extended with evolutionary and search-based algorithms, making Rocket a competitive testing framework for developers. Rocket can also be extended with additional utility methods to develop more sophisticated testing strategies.

Our future work will use Rocket to design and develop a more comprehensive set of algorithms for testing the XRPL. Moreover, the open and flexible design of Rocket encourages collaboration and extension of the framework within the research community. Furthermore, the developed ideas can be transferred to testing other blockchain systems, enhancing the reliability of the XRPL as well as other blockchain systems.

Acknowledgments

This work was conducted as part of the University Blockchain Research Initiative (UBRI), funded by Ripple.

References

[1] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[2] D. Schwartz, N. Youngs, A. Britto, *et al.*, "The Ripple Protocol Consensus Algorithm," *Ripple Labs Inc White Paper*, vol. 5, no. 8, p. 151, 2014.

[3] B. Chase and E. MacBrough, "Analysis of the XRP ledger consensus protocol," *CoRR*, vol. abs/1802.07242, 2018.

[4] L. Winter, F. Buşe, D. de Graaf, K. von Gleissenthall, and B. Kulahcioglu Ozkan, "Randomized testing of byzantine fault tolerant algorithms," *PACMPL*, vol. 7, no. OOPSLA(1), pp. 757–788, 2023.

[5] F. Armknecht, G. O. Karame, A. Mandal, F. Youssef, and E. Zenner, "Ripple: Overview and outlook," in *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings* (M. Conti, M. Schunter, and I. G. Askoxylakis, eds.), vol. 9229 of *Lecture Notes in Computer Science*, pp. 163–180, Springer, 2015.

[6] I. Amores-Sesar, C. Cachin, and J. Mićić, "Security Analysis of Ripple Consensus," in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)* (Q. Bramas, R. Oshman, and P. Romano, eds.), vol. 184 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 10:1–10:16, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[7] K. Kingsbury., "Jepsen," 2022. http://jepsen.io/.

[8] GitHub, "Namazu: Programmable fuzzy scheduler for testing distributed systems." https://github.com/osrg/namazu.

[9] J. Soares, R. Fernandez, M. Silva, T. Freitas, and R. Martins, "ZERMIA - A fault injector framework for testing byzantine fault tolerant protocols," in *Network and System Security - 15th International Conference, NSS 2021, Tianjin, China, October 23, 2021, Proceedings* (M. Yang, C. Chen, and Y. Liu, eds.), vol. 13041 of *Lecture Notes in Computer Science*, pp. 38–60, Springer, 2021.

[10] C. Dragoi, S. Nagendra, and M. Srivas, "A domain specific language for testing distributed protocol implementations," in *Networked Systems - 12th International Conference, NETYS 2024, Rabat, Morocco, May 29-31, 2024, Proceedings* (A. Castañeda, C. Enea, and N. Gupta, eds.), vol. 14783 of *Lecture Notes in Computer Science*, pp. 100–117, Springer, 2024.

[11] E. B. Gulcan, J. Neto, and B. K. Ozkan, "Generalized concurrency testing tool for distributed systems," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024* (M. Christakis and M. Pradel, eds.), pp. 1861–1865, ACM, 2024.

[12] M. van Meerten, B. K. Ozkan, and A. Panichella, "Evolutionary approach for concurrency testing of ripple blockchain consensus algorithm," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 36–47, 2023.

[13] W. Kanhai, I. van Loon, Y. Mangalgi, T. van der Valk, L. Witte, A. Panichella, M. Olsthoorn, and B. Kulahcioglu Ozkan, "diseblab/rocket: v1.0.0," Feb. 2025.

[14] A. Chanda, *Network Programming with Rust: Build fast and resilient network servers and clients by leveraging Rust's memory-safety and concurrency features*. Packt Publishing Ltd, 2018.

[15] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *Ieee access*, vol. 8, pp. 89497–89509, 2020.

[16] C. Cachin and B. Tackmann, "Asymmetric distributed trust," in *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland* (P. Felber, R. Friedman, S. Gilbert, and A. Miller, eds.), vol. 153 of *LIPIcs*, pp. 7:1–7:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[17] XRP Ledger Foundation, "Consensus protocol," 2024. https://xrpl.org/docs/concepts/consensus-protocol/.

[18] B. Chase and E. MacBrough, "Analysis of the xrp ledger consensus protocol," 2018.

[19] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 46:1–46:24, 2018.

[20] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.