An Extensible Framework for Implementing and Validating Byzantine Fault-tolerant Protocols

Hanish Gogada
University of Stavanger
Norway
hanish.gogada@uis.no

Hein Meling
University of Stavanger
Norway
hein.meling@uis.no

Leander Jehl
University of Stavanger
Norway
leander.jehl@uis.no

John Ingve Olsen
University of Stavanger
Norway
johningveolsen@gmail.com

ABSTRACT

HotStuff is a Byzantine fault-tolerant state machine replication protocol that incurs linear communication costs to achieve consensus. This linear scalability promoted the protocol to be adopted as the consensus mechanism in permissioned blockchains. This paper discusses the architecture, testing, and evaluation of our extensible framework to implement HotStuff and its variants. The framework already contains three HotStuff variants and other interchangeable components for cryptographic operations and leader selection.

Inspired by the Twins approach, we also provide a testing framework for validating protocol implementations by inducing Byzantine behaviors. Test generation is protocol-agnostic; new protocols can execute the test suite with little-to-no modifications. We report relevant insights on how we benefited from Twins for validation and test-driven development. Leveraging our deployment tool, we evaluated our implementation in various configurations.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; *Availability*; *Redundancy*; • **Software and its engineering** → **Software development techniques**; **Operational analysis**;

KEYWORDS

Distributed Systems, Blockchains, Byzantine Fault Tolerance, HotStuff

ACM Reference Format:
Hanish Gogada, Hein Meling, Leander Jehl, and John Ingve Olsen. 2023. An Extensible Framework for Implementing and Validating Byzantine Fault-tolerant Protocols. In *The 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED 2023), June 19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3584684.3597266

ApPLIED 2023, June 19, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0128-3/23/06…$15.00
https://doi.org/10.1145/3584684.3597266

1 INTRODUCTION

Many computer systems represent critical infrastructure for business continuity across numerous application domains. Designing and building such critical infrastructure typically require fault-tolerant and highly available systems, where components are replicated to tolerate and handle failures. However, despite decades of research, it remains difficult to design and implement *correct* systems and protocols [26], e.g., that do not violate consistency despite bounded failures and attacks. This has become especially pertinent in this era with a steady stream of new blockchain protocol proposals. While tools can help validate a specific system design [28], there is still a significant gap between the design and implementation. Further, much of the existing systems research is difficult to reproduce [22]. Some of the challenges in reproducing research are replicating the experimental setup and sufficiently exploring rare code paths during experiments.

This paper describes our efforts to reimplement HotStuff [27], a popular Byzantine fault-tolerant (BFT) protocol. HotStuff is implemented using state machine replication (SMR) [23], where a set of replicas reach consensus on the ordering of operations to be executed. HotStuff is leader-based and designed for partially synchronous environments. It is independent of the leader selection policy, and a correct leader can achieve consensus with linear communication costs.

To support our efforts, we have built an extensible framework for implementing fault-tolerant protocols, including modules for network configuration, event subscription, and client handling, as well as evaluation primitives. These primitives, together with a modular design, provide an ideal foundation for the evaluation of both classical [8], current [5, 6], and future protocols.

Using our framework we have implemented several variants of HotStuff [15, 16, 27]. Our implementation provides *fixed*, *round-robin*, and *reputation-based* leader selection policies. The HotStuff paper [27] briefly mentions the notion of a *view synchronizer* that helps to determine the duration of a view without losing responsiveness. In our reimplementation, we use the view synchronizer described in DiemBFT [25]. Finally, the leader uses threshold signatures to prove a proposal's acceptance by a quorum of replicas but does not mandate a specific mechanism. We provide signature schemes based on both ECDSA and BLS12 [4]. Section 3 describes

our system architecture and explains how the different configurable options impact the system's behavior.

Beyond HotStuff, we have also reimplemented the Twins approach for testing BFT implementations [2]. Twins' testing strategy is to introduce *twin* replicas which may send contradicting messages and appear identical to the other replicas. This approach can generate interesting Byzantine behaviors like double-voting and losing the internal state. We can programmatically control the number of test cases to be executed, providing an opportunity to integrate with DevOps tools. Our implementation includes a test suite generation and execution tool and facilitates mocking the necessary interfaces for integration with test suite execution. Section 4 contains a detailed explanation of our optimized test generation logic and test suite execution.

In summary, this paper makes the following contributions:

- **Extensible Framework.** Our architecture provides the necessary foundations to add new protocol implementations without affecting the existing modules. Developers can even replace the framework's core building blocks as long as they abide by the interface's semantics. Our metrics module can be extended to measure protocol-specific metrics.
- **Repeatability.** Our framework is actively being used by the research community to develop and evaluate new protocols [9, 24]. Apart from these, we provide several HotStuff variants, some of which don't provide an open-source implementation. Our extensible framework can be utilized for the repeatability studies of these variants.
- **Validated Implementation** is crucial for the adoption of any BFT SMR system. Developers can dynamically configure our test framework based on the level of testing required. Our test suite generator is a valuable tool for maintaining a validated implementation.
- **Design Takeaways.** We conducted a thorough evaluation of our implementation to identify the impact of our design choices. We identified some crucial takeaways from our evaluation discussed in Section 5.3. For example, "our message translation layer is responsible for a large memory footprint of the replicas." These investigations shed light on areas of improvement in our implementation.

2 BACKGROUND

This section introduces Byzantine fault tolerance, the HotStuff protocol, and the Gorums RPC framework that we leverage to implement HotStuff.

2.1 Byzantine Fault Tolerance

Byzantine fault-tolerant [18] protocols are used to replicate arbitrary applications on multiple servers, called *replicas*, while tolerating arbitrary failures or attacks from a subset of the replicas.

PBFT [8] was the first SMR protocol to tolerate Byzantine failures in asynchronous networks. PBFT can handle f simultaneous failures in a cluster of $3f+1$ replicas. However, the number of signature verifications and message exchanges required to reach consensus grows quadratically with the number of replicas resulting in a scalability challenge. BFT protocols like Zyzzyva [17], SBFT [10], BFT-SMaRt [3], and 700BFT [1] have similar scalability issues in the partial synchronous communication model. Hence, these protocols are unsuitable for large permissioned blockchain deployments.

2.1.1 HotStuff. The HotStuff protocol [27] was designed for permissioned blockchains. The number of signature verifications required to complete a phase grows linearly in the HotStuff protocol. This is possible due to the leader-to-replica-based communication model instead of the mesh-like communication model required in many other protocols. The protocol does not restrict the leader selection policy but assumes every replica knows the view-to-leader mapping. HotStuff requires three phases to commit a request, *prepare*, *pre-commit*, and *commit*. A replica sends its vote for a proposal by generating a partial signature of the proposal's hash. The leader collects and verifies these partial signatures, and if a quorum of valid signatures is received, it uses a threshold signature scheme to form a *quorum certificate*.

Each HotStuff phase mentioned above follows the same pattern: the leader sends the request along with a quorum certificate to the replicas and returns a vote to the leader if deemed correct. This generic behavior provides an opportunity to perform these phases simultaneously, giving rise to Chained HotStuff. In the Chained variant, if a replica votes for a proposal in the *prepare* phase, the replica has voted for the *pre-commit* phase for the proposal's parent and the *commit* phase of its grandparent. This coalescing of phases reduces the protocol's message and signature verification costs.

Fast HotStuff [15] is a variant that claims to provide optimistic responsiveness in two phases instead of three. The premise for this optimization is that the new proposal extends from the highest committed block, and replicas need not change their decision. To prove it, the leader includes a quorum certificate generated from the *New-View* messages sent from the replicas.

Our HotStuff implementation uses Gorums to simplify communication between replicas and between clients and replicas. Gorums [19] is an open-source RPC framework for building quorum-based systems. Gorums is a wrapper over gRPC [11] and uses protocol buffers [12] for marshaling RPC messages. Gorums provides two abstractions, *configurations* and *quorum functions*, that decouple membership and message handling from the protocol execution.

2.2 Validation and Verification

Formal verification tools such as TLA+ [28] has previously been used to prove the safety and liveness of BFT protocols. This approach cannot scale to large models due to state explosion and cannot guarantee the correctness of the implementation.

Twins [2], on the other hand, is a white-box approach for testing BFT protocol implementations. The main idea is to create *twins*, usually a pair of replicas with the same credentials, and selectively send and receive the messages based on some test scenario. This approach can induce the following replica behaviors: (i) *Equivocation*: a replica sends conflicting information to a different set of replicas, (ii) *Amnesia*: the twins may send two valid votes to two conflicting proposals, and (iii) lose the internal protocol state.

Table 1 shows a sample test case generated using the Twins approach with two network partitions and four replicas, where one is a twin replica. Each row consists of the round number and the network partition scenarios for that round. In the context of twins, a round is an abstraction indicating a proposal-vote cycle in

the cluster. Each network partition is a subset of replicas that can send/receive messages among themselves. For example, in round 1, replicas A, B, and C are part of one network partition, and any message sent by A is received only at B and C. A round's leader is indicated with an underscored replica name, and the compromised twin is represented with a $'$ symbol. For example, in round 2, $\underline{A}'$ is the elected leader, and it is the twin of replica A. The protocol is tested by moving through the configuration as specified in each round of the test case.

Table 1: Sample network partition test case generated using Twins.

Round	Network Partitions	
1	$P_1 : \{\underline{A}, B, C\}$	$P_2 : \{A', D\}$
2	$P_1 : \{A, B, C\}$	$P_2 : \{\underline{A}', D\}$
3	$P_1 : \{A, \underline{B}, C\}$	$P_2 : \{A', D\}$
4	$P_1 : \{A, \underline{B}, A'\}$	$P_2 : \{C, D\}$

Testing is performed in two steps, test case generation and test case execution. Test cases are generated based on three parameters: the number of replicas, the number of twin replicas, and the network partitions for each round. All possible permutations of these parameters are enumerated to generate the test scenarios. Hence, the number of test scenarios grows exponentially with the number of replicas. However, developers can optimize the test generation logic to prune test cases with identical or uninteresting behaviors. Network partitions of the test scenario determine the delivery of the protocol messages to the replicas.

After executing a test case, a safety check is performed on all replicas to verify if any of the replicas committed conflicting requests. Liveness violations can be detected if the replicas cannot commit the request after a certain number of rounds.

3 ARCHITECTURE

This section describes the architecture and module system of our HotStuff implementation.

We implemented a set of configurable modules to provide an extensible HotStuff protocol. Every module implements an interface and can have multiple implementations with at least one default implementation. Fig. 1 shows various module interfaces and currently available implementations of these modules. For example, the *crypto* module, which handles the generation and verification of signatures, has two implementations: *ecdsa*, and *bls12*, where *ecdsa* is the default implementation. When deploying the protocol, the system administrator chooses the module implementation.

We apply the separation of concerns design principle. The replica is separated into three layers: communication, protocol, and application. The communication layer handles the network connections and provides broadcast, multicast, and unicast RPC services. The protocol layer contains the actual protocol implementation. The application layer contains the business logic that uses the consensus protocol to provide the service to the end user.

3.1 Building Blocks

The communication layer can be implemented using most communication technologies. We chose Gorums as it simplifies quorum communication and configuration management. The protocol layer comprises seven modules: Consensus, leader selection, crypto, event loop, synchronizer, command cache, and metrics, as shown in Fig. 1.

Some of these modules have only one default implementation, while others have multiple implementations. All module implementations are interoperable. For example, the consensus module has three implementations; Simplified (SHS), Chained (CHS), and Fast HotStuff (FHS), and the selection of any one of these implementations have no impact on any other module implementations. We developed a sample blockchain application to use the underlying consensus protocol. This blockchain stores the blocks by their view number and removes any forked blocks on every commit. The composition of the module implementations determines the behavior of the replicas, and all replicas in the configuration use the same module composition.

A module may interact with other modules to complete its operation. For example, the consensus module invokes the API of the crypto module to generate the signature for voting on a proposal. A *module registry* provides access to all the registered modules and facilitates module interactions.

The module registry exposes two methods: *RegisterModule()* and *GetModule()*. When a module implementation is loaded, the *RegisterModule()* is invoked with a unique name for its implementation to register itself. For example, the Fast HotStuff implementation registers its constructor and the unique name *fasthotstuff*. The *GetModule()* function takes the name of the module implementation and returns the initialized object. Our registry uses the builder design pattern to construct a graph of modules to form a replica object, similar to the approach of existing micro-protocol architectures [13, 14, 20, 21] and also promoted in textbook presentations [7]. A set of chosen or default module objects are passed to the builder object. When the *build()* function is called, a container object is returned, which holds all the initialized objects. This container object is passed to all modules through the *InitModule()* function. With this design, loosely coupled modules are composed to form different behaviors.

Module implementations can gain access to other module interfaces through the module registry. Implementing a module interface

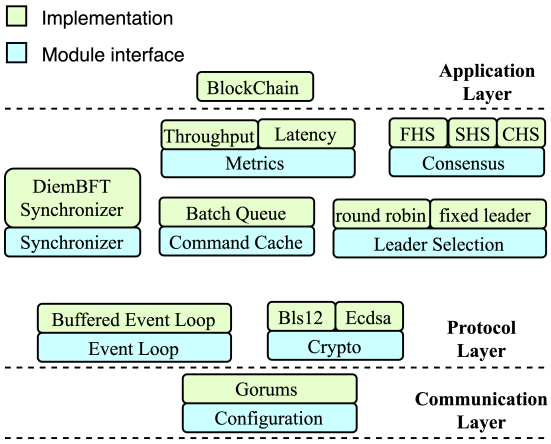

Figure 1: Modular interfaces and implementations of these modules

may sometimes require functionality beyond the module interface API. Thus, the interface definition would have to be changed to allow other modules to use these additional functions. However, this may lead to implementation-specific functions in the interface, resulting in interface explosion and dummy functions. To circumvent this problem, we have introduced an *event loop* mechanism to invoke a module's implementation-specific functionality by passing events between the modules.

All modules have access to the event loop via its API. This allows event-based interactions between modules, such that the events are handled serially. Protocol execution is serialized through the event loop as the messages received from other replicas are converted to hotstuff-defined events and added using the *AddEvent()* method. This avoids the need for mutexes when modules are handling events. Events are passed using a publish-subscribe mechanism. Module implementations can register their interest in specific *event types* using either *RegisterHandler()* or *RegisterObserver()* API calls. There can be multiple observers for each event type, but only one handler. All observers are executed before the handler.

Apart from the publish-subscribe mechanism, the event loop provides two additional functionalities: delayed event handling and periodic event generation. *AddTicker()* method takes a callback function and time interval as input to periodically add the event to the event loop, which is generated by the callback function. *AddTicker()* can also execute periodic operations such as recording protocol metrics. Delayed event handling postpones handling an event until some other event type has been processed and is achieved through the event loop's *DelayUntil()* method. The *DelayUntil()* method helps a replica handle out-of-order messages. For example, in HotStuff, votes for the current view's proposal are sent to the leader of the next view. Thus, there may be scenarios where the previous view's proposal has yet to arrive at the current view's leader, while the proposal's votes could already have arrived. To handle this, the leader calls *DelayUntil()* to delay the processing of these votes until the proposal arrives.

3.1.1 Protocol Execution. Leveraging the *event loop*, a single thread executes the protocol logic synchronously, except for the signature verification. The leader of the view asynchronously verifies the received votes, and when a thread forms the quorum certificate, it is shared with the protocol using the event loop. When a replica receives a new quorum certificate in the proposal, all the signatures in the quorum certificate are verified in parallel. The communication layer implemented with Gorums is multi-threaded, and each thread converts these protobuf messages into protocol-defined events using a *translation layer*. These events are added into the event loop to be processed by the module responsible for handling the events; for example, the view synchronizer handles the event generated upon receiving the *New-View* message.

4 TWINS-BASED TESTING

We have tested our HotStuff implementation by generating and executing the test scenarios obtained using the Twins approach. Twin replicas share the same ID and private key; thus, twins are indistinguishable from a single replica for all the other replicas. Generated test scenarios may contain Byzantine behaviors, which has the potential to discover safety bugs in the implementation.

Our test framework can generate and execute test scenarios as a single step or two separate steps. In the scenario generation step, all the generated scenarios are written to files, and the execution step reads the scenarios from the files to execute them. This prevents the repeated generation of scenarios, thereby saving time and parallelizing the execution. This design of the test framework facilitates the integration of continuous integration and continuous deployment tools. It provides an opportunity to execute specific user-defined scenarios written in the same format as the test generator.

4.1 Test Scenario Generation

The test scenario generator requires the following parameters: (i) *replicas* specifies the number of replicas, (ii) *twins* specifies how many of the replicas should have a twin, (iii) *partitions* represent the number of partitions in the cluster, such that replicas in one partition cannot reach replicas in another. (iv) *rounds* specify the number of views to run for each test scenario. (v) *scenarios-per-file* gives the number of scenarios per output file. (vi) *output* gives the location of the test scenario files. The test scenario generator takes the above parameters as input to the following steps.

Step 1 generates all possible partition scenarios for the specified number of replicas and twins. For example with 4 replicas $\{A, B, C, D\}$, 2 partitions and 1 twin $\{A, A'\}$, some sample partition scenarios could be $\{\{A, B, C\}, \{A', D\}\}$, $\{\{A, C\}, \{A', D, B\}\}$.

Step 2 prunes the partition scenarios considered identical. Two partition scenarios are considered identical if they result in identical behavior by correct replicas. Partition scenarios produce identical behavior only if the correct (non-twin) replicas switch positions in the partitions. For example, the scenarios $\{A, B\}, \{A', C, D\}$ and $\{A, D\}, \{A', B, C\}$, are considered identical as they differ only in the position of B and D replicas. Applying this pruning for the above configuration results in only 6 partition scenarios.

Step 3 takes the partition scenarios generated in Step 2 and assigns each partition scenario to all n possible leaders. This is done by computing the cross-product between partition scenarios and leader assignments. Hence, for a test scenario with p partitions, after Step 3, $n \cdot p$ *leader scenarios* are generated.

Step 4 arranges the scenarios generated in the previous step into the specified number of rounds. Test scenarios are generated by assigning all possible *leader scenarios* to each round. If l *leader scenarios* are generated by Step 3 and r rounds are configured then a total of l^r test scenarios are generated.

We provide an option to randomize the generated test scenarios instead of sequentially generated scenarios. The benefit of randomization is that we can more quickly sample a broader diversity of test scenarios. Thus, hopefully, finding bugs due to Byzantine behaviors more quickly as well. Our experience seems to indicate that this is true. For reproducibility, we can seed the generator to produce the same order of randomized test scenarios.

4.2 Test Scenario Execution

The scenario executor takes the following parameters: (i) *input* is the location of the files containing the scenarios, (ii) the *consensus* algorithm to be used for testing, and (iii) *output* specifies the location to write the failed scenarios.

Test scenario execution uses all available CPUs. Before executing the scenarios, replica objects and their modules are created, as explained in Section 3. The twin replicas are initialized with the same ID and private key. To simulate partitions with Twins, we replace the original Gorums-based configuration module with a mocked configuration module, which is responsible for delivering messages. This mocked configuration module is then initialized with all partition scenarios of the current test scenario.

If replica A broadcasts a message to the configuration in view v, the mocked configuration module delivers the message to replica B *iff* B is in the same partition as replica A for view v. To kickstart a test scenario, the leader of the first view sends the proposal to all reachable replicas. The event loop provides a *Tick()* function to process events one at a time. This function moves the replicas through the scenario's views by executing the event on all replicas before moving to the next event. The consensus module handles messages received through the event loop and replies through the configuration module. The leader module returns the ID of the leader replica based on the leader selection module, so it is mocked to return the leader ID based on the scenarios and rounds. After completing all the views, each replica's commit logs are analyzed for safety violations. A test scenario is written to file if a safety violation is found.

Appendix F of the Twins paper [2] reports on a safety violation found in the Fast HotStuff protocol. The scenario involves four honest replicas with no twins, divided into two partitions, running for 11 rounds. The authors later rectified this safety violation in version 7 of Fast HotStuff [15], and we initially implemented this corrected version. Later we modified our implementation to the faulty version, and we were also able to find the same safety violation using our Twins testing framework.

Apart from the test scenarios derived using the Twins approach, we have developed an extensive suite of unit tests to test our implementation. These test suites guard our code base against mistakes during maintenance and the addition of new features.

5 EVALUATION

Our HotStuff implementation has 14K lines of Go code spread over 100 files. Apart from the protocol implementation, it includes the generated code from the protocol buffers, unit test suite, and deployment tools. Additionally, a metrics module measures the throughput as observed by the replicas and the latency at the clients.

The metrics module registers as a handler for the *CommitEvent*, raised by the replica when a command is committed. These events are counted until a *TickEvent*, which is raised periodically based on a configured measurement interval. The tick event handler logs all the measured metrics and resets the counters. This mechanism can be extended to measure protocol-specific metrics, such as view timeouts and the number of signature verifications.

5.1 Setup

We evaluated our implementation using a local cluster and virtual machines on AWS EC2. Our local cluster consists of 30 machines, and each node has 32 GB of RAM and 12 cores of Intel Xenon processors with Hyper-Threading enabled. A 10 Gbps TOR switch connects the nodes, and the network latency among them is less than 1 ms. All cores have a maximum frequency of 3.3 GHz.

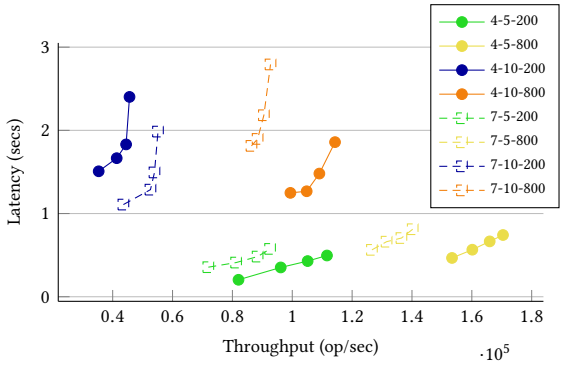

Figure 2: Throughput vs latency for 4 and 7 replicas and 5 and 10 clients for 200 and 800 batch sizes with increasing load from the clients and no payload in the requests

Each node, either physical or virtual machine, can host one or more HotStuff replicas, and unless specified, all experiments are run with one replica/client per node. Communication between the replicas and the clients is secured through TLS, and the public key required to establish trust is shared with replicas and clients over ssh. Unless specified, all experiments are run with zero-sized payloads in the client requests. The deployment procedure for the experiments is explained in Section 5.5.

5.2 Performance Evaluation

We conducted experiments to understand the performance impact of our design choices in common configurations of BFT evaluation.

Experiments show that our implementation provides similar performance to other works [24, 27], is resilient to attacks, and scales well up to 8 cores per machine.

5.2.1 Base Performance. We measured the throughput and latency, varying the batch size, number of clients, and replicas. Replicas are created with the default module implementations, and experiments are conducted without customization. We ran the experiments with 5 and 10 clients for 4, 7, and 16 replicas and batch sizes 100, 200, 400, and 800. Fig. 2 depicts the throughput and latency for 4 and 7 replicas with 5 and 10 clients with batch sizes 200 and 800; other results are not presented due to space constraints. Throughput is the average number of commands executed per second at the replicas, and latency is the average time taken to commit a request as measured by the clients. The starting data point of each plot represents the throughput and latency observed when each client sends $20 \times batch\ size$ concurrent requests, and the load is doubled for subsequent data points. At the given start load, replicas can readily fill the batches.

5.2.2 Resiliency Testing. To test the resiliency of our implementation, we introduced two faulty behaviors for the leaders, *silent* and *slow*. A *silent* leader does not send the proposal to replicas, creating a view timeout and causing a delay in committing a request. A *slow* leader tries to induce a fork by creating a proposal by skipping the

parent with the grandparent as the block's parent. This may lead to discarding the block prepared in the previous view.

We performed experiments and observed the impact of slow and silent leader faults on the throughput of a 13-node cluster. The number of *View-Timeouts* increased exponentially for slow leader faults and linearly for silent leader faults in the cluster as shown in Fig. 4. The configuration with *slow leader* faults executed 50 times more commands than the configuration with *silent leader* faults. A *silent* leader causes *timeout* certificate creation and verification, wasting time and CPU resources. Fig. 3 shows the impact of a recurring slow leader fault on the cluster's throughput. This shows that our implementation can handle faulty leaders, and the throughput recovered after the faulty behavior of the leaders ceased.

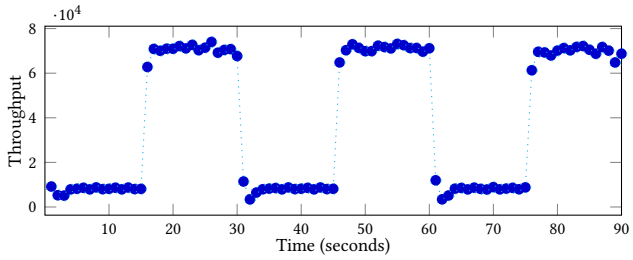

Figure 3: Impact of four slow leaders on the throughput of a 13-node cluster. The experiment was conducted with 5 non-faulty clients at 100 batch size. A replica remains *slow leader* for 15 seconds every 15 seconds throughout the experiment.

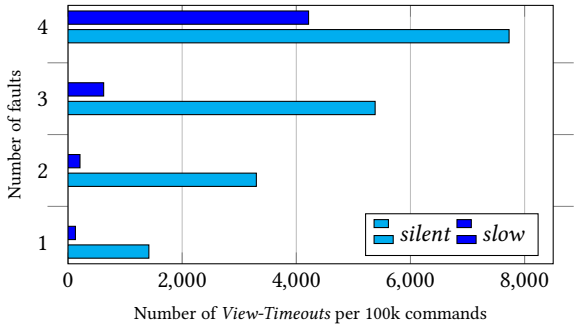

Figure 4: Impact of slow and silent leader faults on a 13-node cluster. The experiment was conducted with 5 non-faulty clients at 100 batch size.

5.2.3 CPU Scaling. Fig. 5 shows how our implementation scales with increasing cores enabled. As the figure shows, with two cores enabled, we observe a throughput increase of more than 100 % compared to a single core. This is because, with two cores, the crypto module can concurrently verify the signatures in the quorum certificate. However, as expected, adding more than four cores only has a limited effect. We note that only connection handling and signature verification are parallelized, while a single thread executes the main protocol logic to reduce possible sources of implementation errors.

5.2.4 Micro Benchmarks. As mentioned in Section 3, a replica's behavior is determined by selecting modules to form the replica object. Since some modules have more than one implementation, we generated micro-benchmarks to compare their performance. We ran each experiment with 16, 32, 64, and 96 replicas with 4 replicas on each physical node. *Ecdsa* and *bls12* implementations of crypto module, *fixed-leader* and *round-robin* implementations of leader-selection module are chosen for this evaluation.

Ecdsa module outperformed *bls12* module as shown in Fig. 6. Apart from the cryptographic complexity, the throughput difference can be partly attributed to the parallelized signature verification in the *ecdsa*. However, with increasing configuration size, the difference in the throughput between *ecdsa* and *bls12* modules is decreasing as expected.

Fig. 7 depicts the throughput and latency for the *round-robin* and *fixed* leader modules. For a configuration size of 16, the throughput of the *round-robin* module is thrice that of the *fixed* leader selection module. Since HotStuff uses a star topology with a fixed leader, the network resources of the leader are quickly overwhelmed, resulting in poor throughput. At a configuration size of 96, both modules performed similarly due to the high network load.

5.2.5 WAN Evaluation. The HotStuff protocol is designed to run in a WAN environment, and we evaluated our implementation in a WAN-based setting using AWS. We used AWS EC2 services to create 9 virtual machines (c4·2xlarge), each with 8 virtual cores with 15 GB RAM and a 1 Gbps network. These servers were distributed across 4 AWS regions in the US; 5 servers were used as clients, and 4 as protocol replicas. Each region has a client and server co-located with one region containing an additional client.

We conducted these experiments to understand the practical performance expectations of our implementation. These experiments are conducted similarly to the base evaluation as explained in Section 5.2.1. At higher batch sizes and with low client load, replicas must wait to fill their batches, leading to increased latency and lower throughput, as shown in Fig. 8. This creates a distinctive 'U' shape for higher batch sizes. However, for experiments with payload, we did not observe the benefits of batching due to the WAN links' bandwidth limitations.

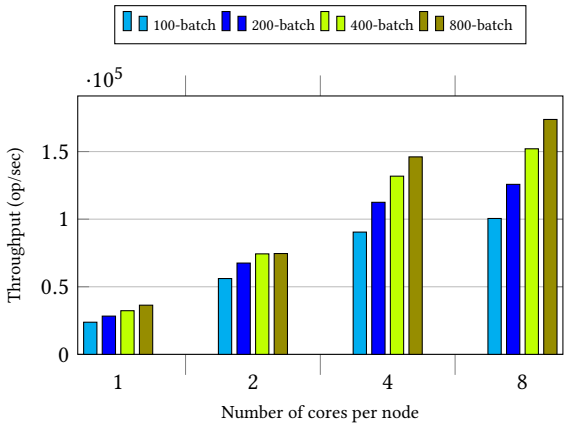

Figure 5: Throughput for 4 replicas and 5 clients with 1, 2, 4, and 8 cores enabled, and with batch sizes 100, 200, 400, and 800.

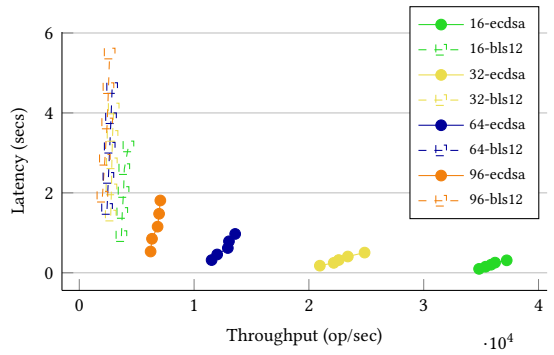

Figure 6: Throughput vs Latency of *ecdsa* and *bls12* crypto modules for 16, 32, 64, and 96 replicas.

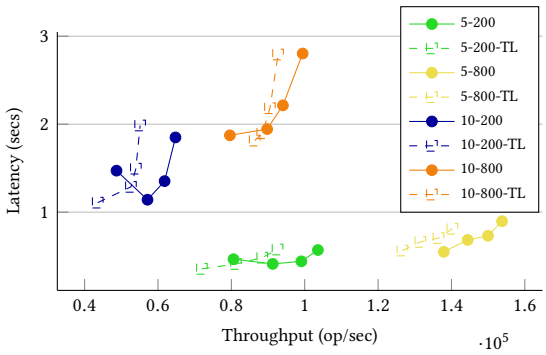

Figure 9: Throughput vs latency for 7 replicas with and without translation layer for 5 and 10 clients with no payload. Graphs labeled with TL are from replicas with the translation layer.

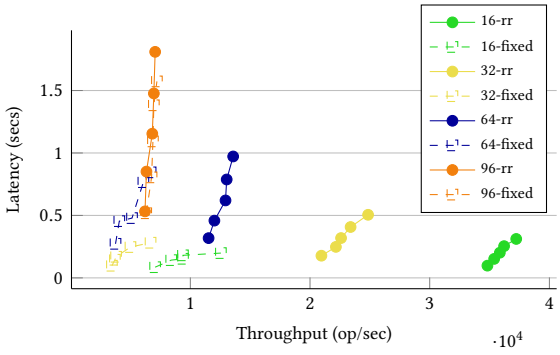

Figure 7: Throughput vs Latency of *round-robin* and *fixed* leader modules for 16, 32, 64, and 96 replicas.

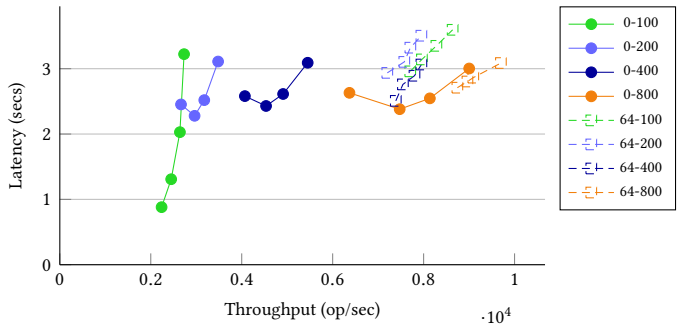

Figure 8: Throughput vs latency: 4 replicas and 5 clients and different batch sizes and payload size 64 bytes with increasing load from the clients in WAN setup.

5.3 Performance Improvements

During the evaluation of our implementation, we realized that some of our design choices were impacting performance. In this section, we evaluate the impact of two performance optimizations.

5.3.1 Removal of Translation layer. To better understand the performance of our implementation, we investigated the CPU and memory profiles of the replicas. Apart from executing the protocol, a significant amount of CPU time (~16 %) is spent on garbage collection. As explained in Section 3.1.1, when messages move from the *Communication* layer to the *Protocol* layer, they are translated from protobuf types to hotstuff-defined types. This translation avoids tight coupling and makes our protocol implementation portable to other RPC frameworks. However, while profiling memory consumption, we identified that the translation layer consumed more than half of the allocated memory. We, therefore, removed the translation layer to decrease the number of allocations and improve throughput and latency. Compared to the base evaluation, we observed a maximum of ~18 % and ~5 % throughput improvement with and without payload, respectively, as shown in Fig. 9 and Fig. 10.

Fig. 11 and Fig. 12 depict the memory footprint of a replica with and without the translation layer during the lifetime of the experiment. The allocated memory is further divided into the memory used by the protocol and communication layers. As expected, the memory occupied by the communication layer remained the same for both variants. Removing the translation layer, we observed ~41 % and ~14 % fewer allocations with and without payload, respectively.

The protocol layer processes the commands as a batch. At lower batch sizes, more batches get executed than at higher batch sizes, so the percentage of saved memory is higher at lower batch sizes. When batching is disabled, we observed that the replica allocated ~44 % less memory without the translation layer. The results presented in the Fig. 11 and Fig. 12 represent the total memory used by a replica during the lifetime of the experiment, and this varies based on the number of commands executed in that configuration. In Fig. 12, our network infrastructure scaled until a payload size of 512 bytes. However, for a payload size of 1024 bytes, we observed decreased throughput, resulting in reduced memory allocation.

5.3.2 Crypto Cache. During the implementation of the protocol, we observed that the quorum certificate is verified twice in the view success scenario. Once in the consensus module before creating a vote and again in the synchronizer module to update the protocol state. Multiple verifications can be prevented if the consensus module can indicate to the synchronizer module that the quorum certificate is already verified. This would create strong assumptions/dependencies among the modules. To prevent such

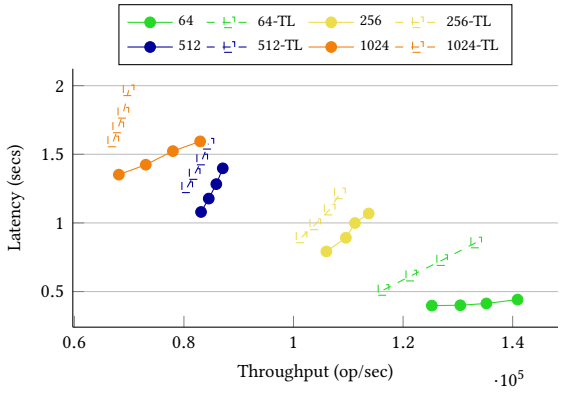

Figure 10: Throughput vs latency for 7 replicas with and without translation layer for 5 clients with 64, 256, 512, and 1024 bytes payload in each request for 100 batch size. Graphs labeled with TL are from replicas with the translation layer.

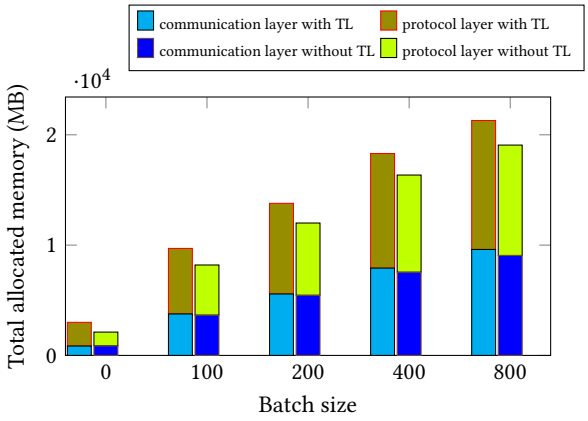

Figure 11: Memory allocated during the lifetime (60 sec) of a replica with and without the translation layer (TL) for different batch sizes. The protocol layer and communication layer memory profiles are captured separately.

dependencies, we created a *crypto cache* as a thin layer over the actual crypto module implementation.

This layer contains an LRU cache to store the quorum certificate and hash of the message. With the cache, modules can verify the quorum certificate multiple times with no performance penalty. All the above experiments are conducted with default module implementations, and the crypto cache is enabled by default.

In this section, we like to quantify the performance benefit of our design decision. Fig. 13 shows the impact of the crypto cache on throughput for various configuration sizes. We observed higher throughput gains for bigger configuration sizes and the maximum throughput benefit of ~60 % in a 19 replica configuration.

5.4 Testing with Twins

We used our Twins-based framework to test our implementation and found an interesting bug. We found the bug in a 4-node configuration with a single twin pair arranged in two partitions over seven rounds. We encountered this bug while executing 600 million generated test scenarios.

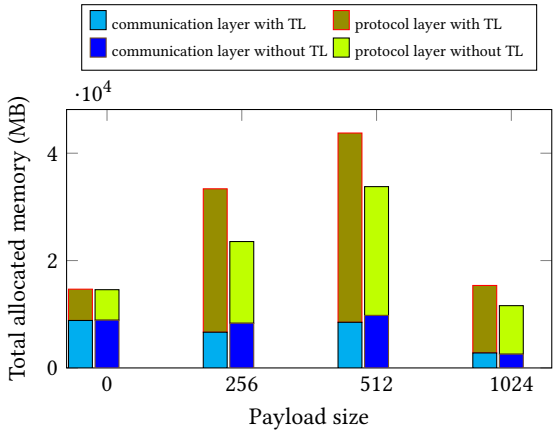

Figure 12: Memory allocated during the lifetime (60 sec) of a replica with and without the translation layer (TL) for different payload sizes. The protocol layer and communication layer memory profiles are captured separately.

A summary of the bug is as follows: to commit a block b, the protocol expects to commit all blocks between b and the last committed block (b_{lc}). The *commit()* function recursively fetches the parent of block b until it reaches b_{lc}. If the block is unavailable at the replica, it tries to fetch the block by its hash from a quorum of other replicas. The recursive *commit()* function did not handle a failure during the fetch operation and went on to commit the current block, causing a safety violation. For example, assume $b_{lc} \rightarrow b'' \rightarrow b' \rightarrow b$ is the correct sequence of the committed blocks. Assume further, a replica has b_{lc} as the last committed block and is about to commit block b. It has to fetch and commit b'' and b'. Failure to fetch b'' may lead to an incorrect chain $b_{lc} \rightarrow b' \rightarrow b$. We fixed the *commit* function to return an error if any of the remote fetches failed and abandon committing the current block.

5.5 iago

We deploy our clients and replicas on many nodes, collecting logs, metrics, and resource usage profiles from the nodes. These tasks typically involve a fair amount of manual labor. We tried to automate these tasks with existing DevOps tools like Ansible and

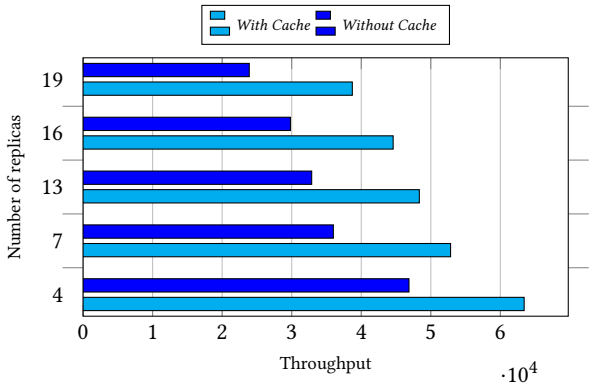

Figure 13: Impact of crypto cache on the throughput of 4, 7, 13, 16, and 19 replicas with 5 clients for a batch size of 100.

puppet, but these tools were complicated, unreliable, and lacked programmatic control over remote nodes. Therefore, we built an in-house deployment framework *iago*, which can create clients and replicas and fetch logs, metrics, and profiles from the remote nodes. Currently, *iago* uses a wrapper over secure shell (SSH) to securely communicate with the other nodes but is flexible enough to support other mechanisms.

Iago defines a *group* abstraction representing all physical nodes available for deployment. A task is executed simultaneously on the nodes using the *group* object and determines the remote replica's behavior in failure scenarios. Using this abstraction, we load the executable binary and create the necessary environment for it to run. A designated *controller* node is used to start a deployment or experiment. Using *iago*, the controller node creates SSH sessions for all the nodes. These sessions perform RPC calls on the remote nodes to create replicas and clients and to start the protocol based on the administrator's configuration. After completing the experiment, the nodes send the logs, metrics, and profiles to the controller node over the same SSH sessions.

6 CONCLUSIONS

Implementing systems and protocols whose purpose is to tolerate and handle failures must not become the root cause of such failures. The difficulty of implementing fault-tolerant systems correctly is apparent from a history spanning several decades of research. We have replicated several variants of the HotStuff protocol using an extensible module framework and a typical event loop. Our implementation has undergone significant evaluation, modifying it for better performance.

Our repository is publicly available at github.com/relab/hotstuff and we welcome well-written and documented contributions. Following state-of-the-art research, we are continuously adding new implementations to crypto, synchronizer, and consensus modules. A CI/CD pipeline is being set up to monitor the correctness and performance impact of the new features.

In summary, we have presented an extensible, resilient, and performance-driven framework to build new BFT protocols or reproduce existing ones.

ACKNOWLEDGMENTS

This work is partially funded by the BBChain and Credence projects under grants 274451 and 288126 from the Research Council of Norway.

REFERENCES

[1] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4, Article 12 (jan 2015), 45 pages. https://doi.org/10.1145/2658994

[2] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. 2020. Twins: BFT Systems Made Robust. https://doi.org/10.48550/ARXIV.2004.10617

[3] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Institute of Electrical and Electronics Engineers, Atlanta, GA, USA, 355–362. https://doi.org/10.1109/DSN.2014.43

[4] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact Multi-signatures for Smaller Blockchains. In *Advances in Cryptology ASIACRYPT 2018*. Springer International Publishing, Brisbane, QLD, Australia, 435–464.

[5] Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Ph. D. Dissertation. University of Guelph, Guelph, ON, Canada.

[6] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. https://doi.org/10.48550/ARXIV.1710.09437

[7] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer Publishing Company, Incorporated, Salmon Tower Building, New York City, USA.

[8] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[9] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-Commit Proofs: Defeating Livelock in BFT. Cryptology ePrint Archive, Paper 2021/1308. https://eprint.iacr.org/2021/1308

[10] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Institute of Electrical and Electronics Engineers, Portland, OR, USA, 568–580. https://doi.org/10.1109/DSN.2019.00063

[11] Google. 2022. gRPC: A high performance, open source universal RPC framework. https://grpc.io Accessed: 2022-05-24.

[12] Google. 2022. Protocol Buffers. https://developers.google.com/protocol-buffers Accessed: 2022-05-24.

[13] Mark Hayden. 1998. *The Ensemble System*. Ph. D. Dissertation. Dept. of Computer Science, Cornell University.

[14] Norm C. Hutchinson and Larry L. Peterson. 1991. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Software Eng.* 17, 1 (Jan. 1991), 64–76.

[15] Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. 2020. Fast-HotStuff: A Fast and Resilient HotStuff Protocol. https://doi.org/10.48550/ARXIV.2010.11454

[16] Leander Jehl. 2021. Formal Verification of HotStuff. In *Formal Techniques for Distributed Objects, Components, and Systems: 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021, Proceedings* (Valletta, Malta). Springer-Verlag, Berlin, Heidelberg, 197–204. https://doi.org/10.1007/978-3-030-78089-0_13

[17] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4, Article 7 (jan 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (jul 1982), 382–401. https://doi.org/10.1145/357172.357176

[19] Tormod Erevik Lea, Leander Jehl, and Hein Meling. 2017. Towards New Abstractions for Implementing Quorum-Based Systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. "Institute of Electrical and Electronics Engineers", Atlanta, GA, USA, 2380–2385. https://doi.org/10.1109/ICDCS.2017.166

[20] Hein Meling and Alberto Montresor. 2009. Type-safe Dynamic Protocol Composition in Jgroup/ARM. In *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction* (Lisbon, Portugal) *(MAI '09)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/1566966.1566967

[21] Hugo Miranda, Alexandre Pinto, and Luis Rodrigues. 2001. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing*. IEEE, Phoenix, Arizona, 707–710.

[22] Gina Moraila, Akash Shankaran, Zuoming Shi, and Alex M Warren. 2014. Measuring reproducibility in computer systems research. *PLoS Comput Biol* 9 (2014), 37 pages.

[23] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[24] Xiao Sui, Sisi Duan, and Haibin Zhang. 2022. Marlin: Two-Phase BFT with Linearity. In *2022 52nd Annual IEEE/IFIP DSN*. 54–66. https://doi.org/10.1109/DSN53405.2022.00018

[25] The Diem Team. 2021. DiemBFT v4: State Machine Replication in the Diem Blockchain. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf

[26] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[27] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[28] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–66.