Tool: An Efficient and Flexible Simulator for Byzantine Fault-Tolerant Protocols

Ping-Lun Wang*, Tzu-Wei Chao*, Chia-Chien Wu*, Hsu-Chun Hsiao*†
* National Taiwan University, Taipei, Taiwan
† Academia Sinica, Taipei, Taiwan

***Abstract*—A Byzantine Fault-Tolerant (BFT) protocol protects a distributed system from faulty participants. To provide both *liveness* and *safety*, many such protocols assume they are dealing with a partially-synchronous network, which will eventually stabilize after a global stabilization time (GST). In a real-world network environment, however, there is no such guarantee of bounded transmission time for network packets. For this reason, even if a BFT protocol is mathematically proven to achieve both liveness and safety, its *overall performance* is difficult to analyze theoretically, especially if there are bad network conditions or adversarial behaviors. Accordingly, we propose a simulator for evaluating the performance of BFT protocols under various network conditions and attacks, and we implement it to empirically compare the performance of eight representative protocols. Experiment results show that our simulator can simulate 16 times as many nodes as an existing simulator supports (512 vs. 32), and it is over 500 times faster when simulating 32 nodes (38 milliseconds vs. 19.4 seconds).**

***Index Terms*—BFT Protocols, Simulation Tool, Byzantine Fault**

I. Introduction

The *Byzantine Generals Problem*, first delineated by Lamport et al. [1], is the problem of reaching an agreement in a distributed system that includes faulty nodes. Such nodes, also called *Byzantine nodes*, can perform arbitrary behaviors (*Byzantine behaviors*), including but not limited to crashing, not following a protocol, and sending misleading messages. A Byzantine fault-tolerant (BFT) protocol prevents inconsistent states in a distributed system, achieving both *liveness* and *safety*, despite the presence of Byzantine nodes. An already extensive body of research on BFT protocols [2]–[5] has been considerably augmented since the rise of blockchains [6]–[10]. Each such protocol has a different design, which leads to varying *performance claims* and *attack resistances* across different *network environments*.

As BFT protocols are usually applied to time-critical systems, such as blockchains and avionics, their inefficiency is a critical security issue. A successful attack that causes performance degradation or even denial of service (DoS) on a blockchain increases the risk of double-spending [11], while a DoS attack on aircraft systems can lead to crashes. However, theoretical analysis of any BFT protocol's correctness and performance is difficult, due both to attacks and to the complex nature of its underlying networks. Thus, such analysis has been simplified via a number of assumptions. Notably, to overcome the FLP impossibility result [12], many analysts of BFT protocols assume a partially-synchronous network, which eventually stabilizes after a global stabilization time (GST). Under a partially-synchronous network model, such protocols are free to focus on maintaining safety throughout their execution—i.e., even when the network is unstable—and then attempt to terminate after the network stabilizes.

Such assumptions are useful for confirming a BFT protocol's liveness and safety properties, and for analyzing its worst-case performance under particular conditions. However, assessing its overall performance (e.g., the time it takes to reach a consensus and terminate) remains difficult in a real network environment, which may be unstable intermittently. It is also unknown how such performance will be impacted by adversarial behaviors. For example, the security and performance analysis came with the HotStuff protocol [10] was limited to scenarios in which its nodes are in the same *view*; but as our evaluation will show, when the HotStuff BFT protocol is using a naive view-doubling synchronizer, nodes in HotStuff can be knocked out of synch easily when the network delay is underestimated, resulting in undesirably long latency. Since our implementation of HotStuff may differ from its intended implementation, we annotate our version as HotStuff+NS (naive synchronizer) in this paper. Some researchers seeking insights into the behaviors of BFT protocols have used experimentation to empirically compare such protocols' performance [13], [14], and several simulators are now available to facilitate the testing process. However, existing simulators are either not general enough to support a broad range of BFT protocols [15], or can only simulate benign failures (e.g., fail-stop) [16], [17]. In addition, they are inefficient and cannot reliably support larger sets of nodes (e.g., beyond 32), causing them unsuitable for simulating newer protocols.

In this paper, therefore, we propose an efficient and flexible simulator for BFT protocols. Our goal is that it should be able to simulate the execution of a BFT protocol under multiple attack scenarios and network conditions in a short period of time; and that it should enable empirical performance evaluation despite attacks occurring.

A key difference between our approach and prior ones lies in how we model Byzantine behaviors. Existing work instantiates Byzantine nodes and controls their individual behaviors to simulate attacks against honest nodes [15]. Moreover, the set of Byzantine nodes is usually fixed before simulation begins, and cannot modify the network messages transmitted by other nodes. This limits the simulated attacker's capabilities, and thus, such simulators cannot support some of the attacks that we introduce in Section III-C. For example, an adaptive attack should be able to compromise nodes during protocol execution, and a rushing attack should be able to observe or modify messages from other nodes. Accordingly, instead of controlling individual Byzantine nodes to model Byzantine behavior, we construct an abstracted global attacker that is able to observe or modify messages between honest nodes and selectively attack some of them.

To demonstrate the power of our simulator, we use it to implement eight representative BFT protocols (e.g., PBFT, HotStuff+NS, and LibraBFT) and three attacks, and compare their performance. This yields several interesting findings about the behaviors of these protocols. One such finding is that, when the network delay is higher than expected, HotStuff+NS needs to change its leader frequently before deciding one value, and this results in a 5.3-fold higher latency than when network delay is normal.

II. Background

This section provides background about BFT protocols, including the formal definition, common network models, and performance metrics.

A. BFT Protocols

Suppose there are n nodes in a distributed system, and there are f Byzantine nodes among them. The remaining $n - f$ nodes are called *honest nodes*. A BFT protocol allows each honest node to reach a consensus while guaranteeing *safety* and *liveness*. Safety ensures the consistency between any two honest nodes, and liveness requires all honest nodes eventually reach a consensus.

This paper mainly considers BFT state machine replication (SMR) protocols, which continuously reach a sequence of consensuses. To exemplify and unify the terms used in this paper, we describe a simplified BFT SMR protocol below. All nodes locally maintain a monotonically increasing number called *view*. Each view has a dedicated *leader* known to every node, and the remaining nodes are *followers*. Importantly, the leader need not be honest. The leader creates a proposal and sends it, along with the view number, to all followers. If a follower's view is the same as the leader's, and it agrees to the proposal, it sends a *vote* back to the leader. The process consisting of a message being sent from the leader and voted by followers is a *round*. After a certain number of successful rounds, the proposal is *decided*, and it is agreed by all nodes.

B. Common Network Models

Correctness and performance claims about BFT protocols are often rooted in certain assumptions about their underlying networks. The three most common network models are [18]:

- **Synchronous Network.** There is a known upper bound Δ on the message delay among honest nodes.
- **Partially-Synchronous Network.** There is an unknown upper bound Δ on message delay among honest nodes; equivalently, there is an unknown time called *Global Stabilization Time (GST)*, and the message delay is upper bounded by a known value Δ after GST.
- **Asynchronous Network.** There is no upper bound on the message delay among honest nodes.

C. Performance Metrics of BFT Protocols

When analyzing the performance of BFT Protocols, we will mainly focus on two metrics: time usage and message usage. A detailed explanation of each is provided below.

- **Time Usage** is calculated as the time elapsed between the beginning and termination of a BFT protocol. Even though *round complexity*, which is defined as the number of rounds taken before a BFT protocol terminates, is more commonly used in theoretical analysis, the latency of a round can vary considerably across BFT protocols. Thus, we have elected to use the former in our simulations, while our simulator can support round complexity as well.
- **Message Usage** reflects the communication cost of a BFT protocol. As messages may be encoded in various ways, instead of calculating their actual sizes in bytes, we calculate the number of transmitted messages. If necessary, the total bytes can be reconstructed via estimating the size of each message and calculating the sum of size of all messages.

As well as facilitating empirical/quantitative comparison of the performance of different BFT protocols, as mentioned above, using these two low-level metrics enables further investigation of whether a BFT protocol achieves two important properties that greatly influence the performance of a BFT protocol: *view synchronization* (§II-C1) and *responsiveness* (§II-C2). Each is discussed in its own section, below.

1) View Synchronization: Many view-based BFT protocols, including PBFT [3], HotStuff [10], and LibraBFT [19], will only terminate when nodes are in the same view. Thus, if nodes have trouble converging to the same view—for example, due to an unstable network or an attack—the performance of these BFT protocols will be negatively affected. Naor et al. [20] formalized this as the *view synchronization problem*.

The round complexity and the message complexity of each of the above-mentioned protocols have been extensively studied via theoretical analysis. However, such analysis has assumed not only that networks are stable, but also that nodes are synchronized in their views; and even so, it has usually yielded only rough time bounds for termination. Our simulator, in contrast, can quantitatively evaluate the performance of a BFT protocol even when its nodes are out of sync.

2) Responsiveness: A BFT protocol is deemed responsive [21] if it proceeds as soon as the majority of nodes agree. This means that the latency of its agreement process depends only on the actual network latency, rather than on any predefined parameters such as timeouts. Responsiveness is considered a desirable property, because the faster the network is, the faster the nodes can reach a consensus.

III. Tool Design and Implementation

This section describes the design and implementation of our simulator tool [1], beginning with its high-level infrastructure and core components, and then proceeding to its BFT protocols and the attackers we implemented with it. Because

[1]https://github.com/csienslab/BFT-Simulator

this simulator handles the fundamental components required for running and evaluating a BFT protocol—such as network communication, metrics calculation, and data logging—researchers who would like to experimentally evaluate their new algorithms can easily implement both of the core logic of their BFT protocols and customized attacks. This simplicity is reflected in the relatively few lines of code required for our reference implementation, as summarized in Tables I and II.

We implemented our simulator using JavaScript programming language and a Node.js runtime environment [22].

A. *Infrastructure*

As shown in Fig. 1, the simulator consists of five main components: controller, event queue, consensus module, network module, and attacker module. For purpose of demonstration, we have implemented eight representative BFT protocols and three attack scenarios using our simulator. Users can also import or write customized BFT protocols or attack scenarios. Specifically, a user of our simulator needs only to write a configuration file specifying the network model and parameters, the BFT protocol, and, optionally, the attack scenario.

Below, we explain the functionality of each module.

1) Controller: The controller initializes all the other modules according to a configuration file provided by the user. It also manages the event queue, which sorts all events according to their event timestamps; dispatches events to the corresponding modules; and updates the simulation clock accordingly. After a consensus is reached, the controller outputs the simulation result calculated from the performance metrics (i.e., time and message usage defined in §II-C).

2) Event Queue: We adopt a technique commonly used by network simulators [23]: using a simulation clock and event queue to calculate simulation time, instead of measuring the wall-clock time. We use a priority queue as our event queue to sort all events by timestamps, and the simulation clock is advanced according to a timestamp attached to an event popped from the event queue. The event queue manages two types of events: **message event** occurs when a node receives a message, and **time event** occurs when a simulation clock reaches a certain time.

3) Consensus Module: The consensus module implements the core logic of a BFT protocol and controls the behaviors of honest nodes; all adversarial behaviors are performed by the attacker module. The consensus module is able to send messages to other nodes through the network module, or register time-triggered events with the controller.

To simulate a customized protocol, a user of our simulator needs only to implement three functions:

- *onMsgEvent*. The callback function that is executed when a node receives a message event.
- *onTimeEvent*. The callback function that is executed when a node receives a time event.
- *reportToSystem*. The communication interface the consensus module uses to send result to the controller.

While the BFT protocols we implemented are mostly single-leader protocol, multi-leader or leaderless protocols are also

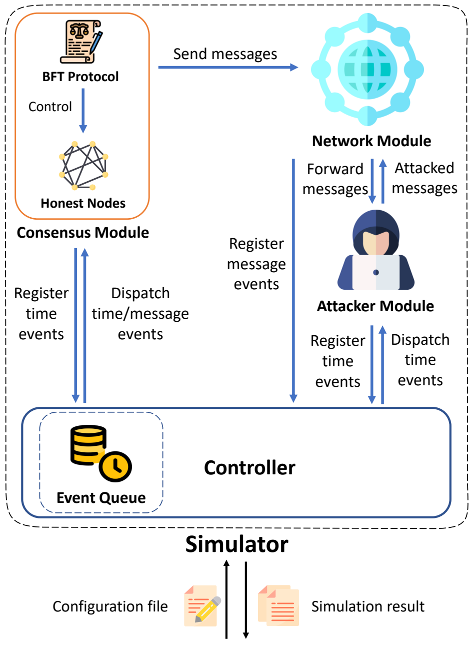


Fig. 1: The infrastructure of the proposed simulator.

supported since the consensus module can freely decide which nodes or no nodes at all are the leaders.

Note that we currently does not calculate the computational cost of an honest node, and therefore measuring the throughput of a BFT protocol is not possible. One way to add this feature is to estimate the computation time through calculating the number of computational extensive operations, such as cryptography operations.

4) Network Module: The network module simulates a peer-to-peer network. To achieve this, each node is connected to this module; to simulate message delivery from one node to another, the sender node sets the *source* (sender) and *destination* (receiver) variables in the message and sends it to the network module. The network module then sets a *delay* variable for each message according to the network configuration, and forwards these messages to the attacker module to simulate attacks. The *delay* variable can be sampled from any distribution, such as a Gaussian distribution or a Poisson distribution, which can easily be changed to simulate various types of networks.

After passing through the attacker module, each message is registered as a message event and delayed according to its *delay* variable. When a message event is triggered, the message is dispatched to its destination by the controller.

By adjusting the delay of each message in the network module, we can simulate common network models:

- **Synchronous**. If a BFT has a predefined network configuration λ, and all packet delays are bounded by a fixed bound $b \leq \lambda$, we are simulating a synchronous network, as the BFT runs with a known network bound λ.
- **Partially-Synchronous**. If a BFT has a predefined network configuration λ, and all packet delays are bounded by a fixed bound b, we are simulating a partially-

synchronous network, as the network has a bound b that is unknown to the BFT protocol.
- **Asynchronous**. If all packet delays are sampled without any bounds, we are simulating an asynchronous network.

5) Attacker Module: The attacker module simulates attacker behaviors. Specifically, it simulates Byzantine nodes, which receive and send messages to deceive honest nodes that do not know which among them are Byzantine nodes. To support the simulation of various attacker capabilities, the attacker module has access to the entire network, and simulates attacks by setting the *delay* variable, dropping messages, or inserting new messages.

To construct a customized attacker, two functions need to be implemented. These are:

- *attack*. The callback function that is executed when the network module forwards messages to the attacker.
- *onTimeEvent*. The callback function that is executed when the time events registered by this module are dispatched.

Section III-C provides the guidelines for implementing an attacker and the list of attackers that we implemented.

6) Validator Module: The validator module is a special mode of the network module designated for cross-validating the simulation result against a given ground-truth result. The ground-truth event sequence (e.g., *commit*, *pre-prepare*) can be generated by another simulator or the actual implementation of the BFT protocol. When the validator module is enabled, it replays the message events according to the ground-truth event sequence. When a consensus is reached, the validator module checks whether the consensus module produces the same result (i.e., which node agrees on what value) as the ground truth.

B. BFT Protocols Implementations

The BFT protocols we implemented, which collectively correspond to all three of the network models discussed above, are listed in Table I. Each is also described below.

1) Three Versions of ADD+ BA: ADD+ BA [7] is a synchronous BFT protocol with optimal resilience and expected constant-round termination. They began by providing a basic protocol (which we call ADD+v1), and then extended it to include a verifiable random function (ADD+v2) to tolerate a static attack. Finally, they added a *prepare round* to tolerate the adaptive and rushing attacks (ADD+v3). We simulate these three variants to demonstrate our simulator's ability to support various attacks.

2) Algorand Agreement: Algorand Agreement [6] is a fast and partition-resilient BFT protocol. Designed for a synchronous network, it can reach agreement in an expected constant number of rounds if the network latency between honest nodes is smaller than a known bound. We selected Algorand Agreement because it is one of the best-known synchronous BFT protocols with partition resilience.

3) Async BA: Asynchronous Byzantine Agreement (async BA) [2] is a classic binary-value BFT protocol designed for an asynchronous network. Its key contribution is that it limits the behavior of Byzantine nodes using reliable broadcast plus a validation function. Due to the FLP impossibility result [12], async BA only provides probabilistic liveness instead of guaranteeing liveness.

4) PBFT: Practical Byzantine Fault Tolerance (PBFT) [3] is one of the best-known BFT protocols. It has been widely used in many projects, notably including early versions of Tendermint [24] and Hyperledger Sawtooth PBFT [25]. It features responsiveness and can work in a partially-synchronous network by doubling its timeout every time it changes its view.

5) HotStuff+NS: HotStuff [10], for partially-synchronous networks, reduces latency by pipelining the three-phased confirmation process. Like PBFT, HotStuff provides responsiveness. Given an honest leader and a stable network, HotStuff's communication complexity is linear to the number of nodes.

HotStuff decouples liveness from safety via a module called *PaceMaker*: a view-synchronization algorithm responsible for keeping the nodes in the same view. While HotStuff's paper states that PaceMaker can be constructed using an exponential back-off mechanism, it did not provide a reference PaceMaker implementation. Thus, we implemented our own PaceMaker for HotStuff+NS: a view-doubling synchronizer utilizing an exponential back-off mechanism, as described by Naor et al. [20].

6) LibraBFT: LibraBFT [19] is a product based on HotStuff. The main difference between them is how they implement the PaceMaker. In LibraBFT, a node broadcasts a timeout certificate when timeout occurs, and a node advances to the next view when it receives more than a threshold number of such certificates. LibraBFT thus guarantees a time bound on termination after GST, whereas HotStuff does not. As shown in later experiments, this difference allows LibraBFT to have a much better performance when network is unstable.

C. Attacker Implementations

Our construction of an abstracted global attacker allows out simulator to flexibly supports many attacks. As a demonstration, we implemented three attacks, which are listed in Table II. Next, we will list the attacker's capabilities and explain how our simulator implements them.

- **Fail-stop**. A fail-stop node simply stops participating in the protocol. This is the weakest form of Byzantine behavior. To simulate fail-stop nodes, we start the system with $n-f$ honest nodes, and set the total number of nodes to n. This gives us f fail-stop nodes.
- **Partition Attack [6]**. We consider the partition attack described in Algorand [6], which divides a network into two or more subnets. As all messages pass through the attacker module, a partition attack is simulated via between-node packet-filter rules. The attacker can either drop or delay the packets between different subnets.
- **Static and Adaptive [7]**. An attack is static if the attacker has to decide which nodes to control and transform into Byzantine nodes before a BFT protocol starts. On the other hand, an adaptive attack can choose and compromise a node during execution, as long as the number of Byzantine nodes remains $\leq f$. An adaptive attacker can

TABLE I: Implemented BFT Protocols

BFT Protocols	Network Model	LoC
ADD+v1 BA [7]	Synchronous	304
ADD+v2 BA [7]	Synchronous	307
ADD+v3 BA [7]	Synchronous	376
Algorand Agreement [6]	Synchronous	387
Async BA [2]	Asynchronous	265
PBFT [3]	Partially-Synchronous	606
HotStuff+NS [10]	Partially-Synchronous	502
LibraBFT [19]	Partially-Synchronous	568

Note. LoC = lines of code (n).

TABLE II: Implemented Attacks

Attacks	Attacker Capability	LoC
Network Partition Attack	Partition	86
ADD+ BA Static Attack	Static	86
ADD+ BA Adaptive Attack	Rushing + Adaptive	117

Note. LoC = lines of code (n).

turn an honest node into a Byzantine one by dropping, modifying, or inserting messages of the Byzantine node from the attacker module. Since nodes can only interact by messages, controlling a node's messages is equivalent to controlling its behavior observed by other nodes.

- **Rushing [7]**. An attack is termed rushing if the attacker waits to decide upon its next action after observing every message sent by the honest nodes. Since all messages pass through the attacker module before being sent to their destinations, it is by nature a rushing attack.

D. Implementation Validation

Formally verifying the correctness of a BFT implementation is challenging. Instead of proving its correctness, we examined whether our simulator generates the same execution traces as another well-known BFT simulator, BFTsim [17]. Our validator module confirms that our simulation of PBFT generated identical event sequences as BFTsim's in cases we tested. This validation is limited within those protocols and attacks that both ours and BFTsim support. To gain more confidence in our simulation results, users can also use our validator module to compare the simulated trace with the actual ones generated by a full-blown BFT network. We do not claim to use simulation to prove the correctness of a BFT protocol.

IV. Experimental Evaluation Using Our Simulator

Classic distributed systems [3], [4], [18] are usually deployed in a size $n = 4, 7, 10$, and newer protocols [10], [19], [26] designed for blockchains are better suited to larger sets of nodes (e.g., 64, 128, 256). However, existing simulators [17] can only support a small number of nodes and require long simulation time, as shown in Fig. 2. While our simulator is capable of simulating a large number of nodes, we use 16 nodes in the following evaluation by default, because this is sufficient for observing protocols' behaviors, and acceptable for both classic BFT and BFT protocols for blockchains.

In the following descriptions of our experiments, the following two notations are used. A predefined parameter, λ, is used in synchronous or partially-synchronous protocols to represent the estimated upper-bound of network delay. Its unit

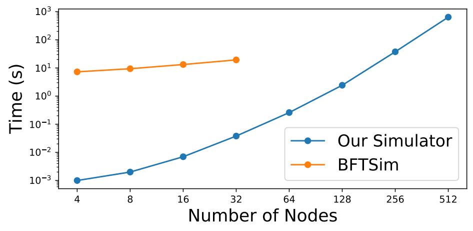


Fig. 2: Simulation time for PBFT using our simulator and BFTSim [17] ($\lambda = 1000$; $\mathcal{N} = (250, 50)$). Note that BFTSim can only simulate 32 nodes due to out-of-memory errors.

is milliseconds. A normal distribution, $\mathcal{N}(\mu, \sigma)$, is used to sample network delays in the network module, with mean μ and standard deviation σ. In a network setting, the units of μ and σ are milliseconds.

As LibraBFT and HotStuff+NS optimize their decision processes via a pipeline technique, they need more successful rounds to decide the values proposed in the first few rounds. To reflect their performance under normal conditions, we measure their time/message usage by calculating the average latency or message count for each decision after ten values are decided. For other protocols, we measure their performance after one value is decided. Each experiment is performed 100 times to calculate the average and standard deviation. The following experiments were run on a Ubuntu 18.04.1 machine with a 4.0GHz 12-core AMD Ryzen 9 3900X CPU and 32 GB RAM.

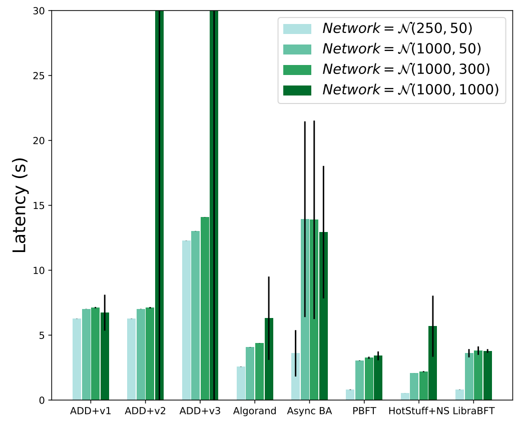
A. Performance across Different Delays

We adjusted our network environments in terms of their means and variances that ranged from fast and stable to slow and unstable, and setting λ to 1,000 ms. The averages and standard deviations of latencies and message counts are shown in Fig. 3a and 3b, respectively. In these figures, the bar height represents mean value, and the vertical line running down the center of each bar represents standard deviation. For time usage, HotStuff+NS had the shortest latency most of the time, except in the case where the network was set to $\mathcal{N}(1000, 1000)$. In that case, PBFT was slightly faster than HotStuff+NS. As for message usage, HotStuff+NS also outperformed the other protocols.

B. Performance across Different Timeout Configurations

1) Performance when Delay is Overestimated: Fig. 4 illustrates the responsiveness of BFT protocols when we increased the timeout configuration λ from 1000 ms to 3000 ms, while leaving the network delays fixed at $\mathcal{N}(250, 50)$. It shows that increasing λ only affects synchronous protocols.

2) Performance when Delay is Underestimated: Only partially-synchronous protocols are included in this experiment as underestimated delay violates synchronous protocols' assumption, and async BA's performance is not affected by different λ configuration. Fig. 5 shows that LibraBFT was not affected by underestimated delay, while PBFT performed better when λ is closer to the actual delay. HotStuff+NS became very unstable when delay is underestimated, because


(a) Time usage

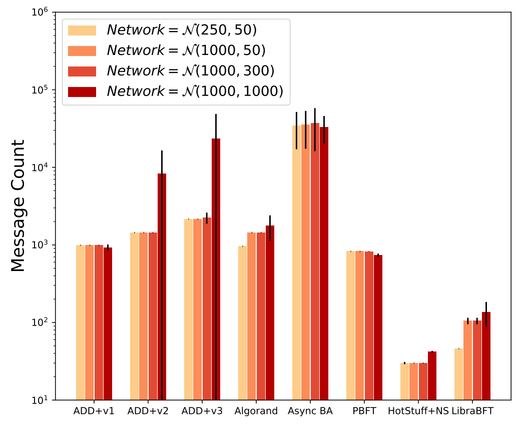

(b) Message usage (in log scale)

Fig. 3: The performance of BFT protocols under four different network environments.

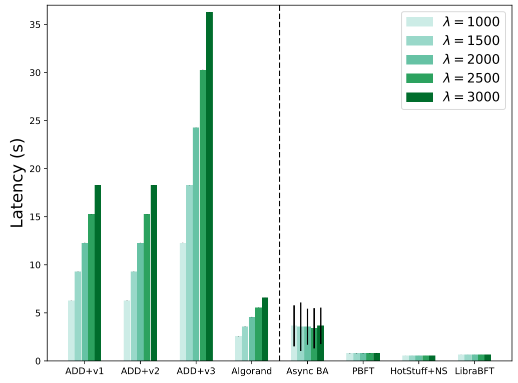


Fig. 4: Latency when timeout is overestimated. Protocols with responsiveness are to the right-hand side of the dotted line.

its PaceMaker sometimes cannot solve the view synchronization problem (see §II-C1) efficiently, and struggle to reach a consensus. Section IV-D discusses this situation in detail.

C. Performance under Attack

We simulated network partition attacks, which target the network, before proceeding to fail-stop attacks and rushing adaptive attacks, which target multiple Byzantine nodes.

1) Network Partition: As synchronous protocols are generally not partition-resilient, we only include Algorand, which is resilient to partition attacks, in this experiment. Most protocols terminated a few seconds after the partition resolved, but HotStuff+NS spent roughly an additional 100 seconds to reach a consensus. Since HotStuff+NS's PaceMaker doubles its delay when a timeout occurred, its delay grew exponentially when the network was partitioned. Therefore, when the partition resolved, it still needs to wait a long period of delay before the protocol proceeds to execute.

2) Performance under Fail-stop Attack: As Fig. 7 shows, partially-synchronous protocols are less resilient to fail-stop nodes. This was because these protocols rely on messages from honest nodes to proceed. Also, we found that the latency of HotStuff+NS degraded drastically.

3) Static Fail-stop Attack on ADD+v1 and ADD+v2: Since ADD+v1 has a deterministic leader sequence, a static attacker can select the first f nodes that will become the leader, and fail-stop them when they do so. All of its nodes are thereby forced to enter the next view, and this will delay its termination for f rounds. The relevant performance results are shown in the left-hand subgraph of Fig. 8. To prevent this attack, ADD+v2 utilizes a verifiable random function (VRF) to randomize leader election.

4) Rushing Adaptive Attack on ADD+v2 and ADD+v3: When confronting a rushing adaptive attacker, i.e., one capable of deciding which node to compromise after receiving each node's VRF value, ADD+2 cannot terminate in an expected constant round. ADD+v3, however, can terminate in an expected constant round despite being targeted by a rushing adaptive attacker model. This is achieved by adding a *prepare* round to decide on the proposed value. Their performance are shown in the right-hand subgraph of Fig. 8.

D. View-synchronization Analysis

In this section, we report on the use of our simulator to 1) analyze the view-synchronization problem and 2) visualize the view of each node during simulation.

As shown in Fig. 5, HotStuff+NS experiences severe performance degradation when λ is set to 150 ms and $\mathcal{N} = (250, 50)$. That is, its latency can be as high as 80 seconds in some extreme cases, as shown in Fig. 9. Its nodes are separated into groups of different views after 5 seconds, and this situation prevails for an additional 75 seconds, when the nodes are finally synchronized to the same view. This indicates that underestimated delays can cause nodes to have divergent views, and to spend a considerable amount of time resolving the view-synchronization problem.

V. Related Work

BFTSim [17] is a BFT simulation framework using the P2 declarative logic language and the ns-2 network simulator. It supports testing BFT protocols under various network

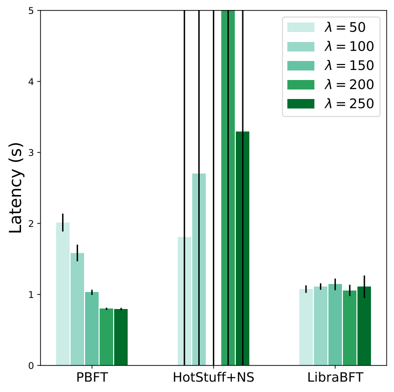


Fig. 5: Time usage of partially-synchronous protocols across different network configurations λ, with $\mathcal{N} = (250, 50)$.

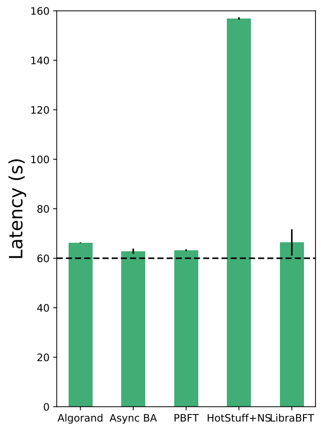


Fig. 6: Time usage under a network-partition attack. The dotted line is the time when partition resolves.

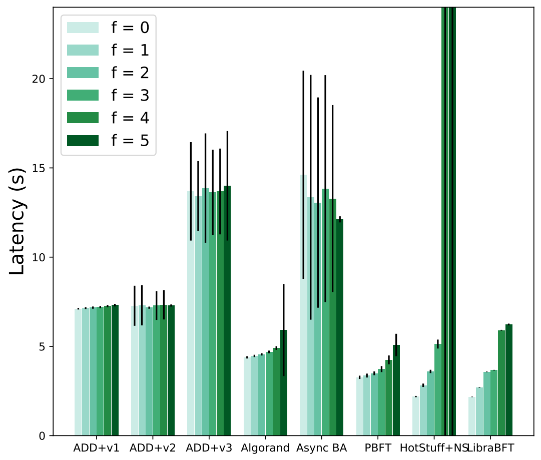


Fig. 7: Time usage across different numbers of fail-stop nodes ($\lambda = 1000$; $N = (1000, 300)$).

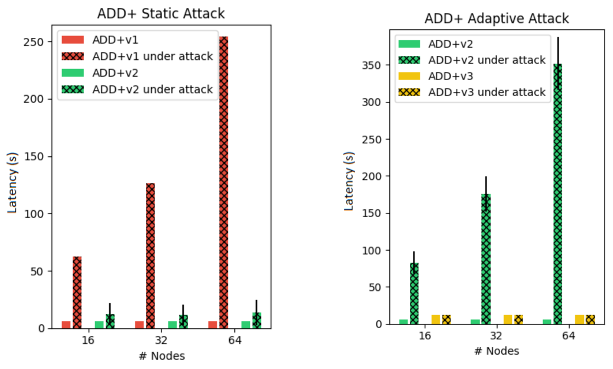


Fig. 8: Latency in seconds (y-axis) associated with static (left) and adaptive (right) attacks on all three ADD+ variants

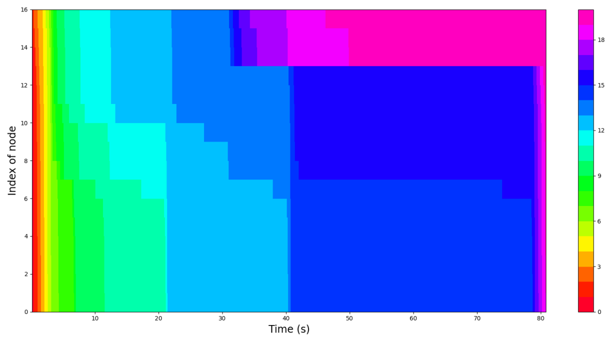


Fig. 9: Each node's view during HotStuff+NS execution ($\lambda = 150$; $N = (250, 50)$). Each color represents a view number.

conditions (e.g., latency and bandwidth) and configurations (e.g., cryptography primitives, timeout, and faults). Thus, its users can compare BFT protocols' performance objectively. However, BFTSim only models performance under benign failures such as misconfiguration or silent replicas; in contrast, our simulator is designed to support the modeling of various attacks. It is unclear whether and how to extend BFTSim and model more complicated logic (e.g., different view synchronization algorithms) or sophisticated attack strategies [27] using the P2 language. Moreover, BFTSim simulates a complete network, including the physical and link layers, using the ns-2 network simulator, which slows down the simulation. In contrast, our simulator focuses on high-level network models closely relevant to BFT protocols and thus scales better than BFTSim.

Bano et al. [15] proposed the Twins approach, which tests the security of a BFT protocol by automatically generating several Byzantine attack scenarios. The Byzantine behaviors covered by Twins include equivocation, double voting, and losing internal state. Its creators implemented a unit-testing apparatus for LibraBFT, which showed that Twins could discover protocol flaws within minutes. However, Twins focuses chiefly on test-case generation, whereas we aim to simulate various attacks and network conditions for helping assess BFT protocols' safety and liveness properties.

VI. Future Work and Conclusion

Many BFT protocols have been proposed. To further understand their behavior under a range of network conditions and attacks, we designed and implemented a simulator supporting various attacks and simplified the network structure to reduce simulation time. Using our simulator, we discovered several aspects that significantly influenced a BFT protocol's performance, e.g., view synchronization and network stability. Our tool can facilitate further research, not least by providing crucial insights into the proving, analysis and design of new, more secure BFT protocols.

Acknowledgments

We thank the anonymous reviewers, our shepherd Francois Taiani, and Pawel Szalachowski for their helpful feedback. This research was supported by the Ministry of Science and Technology of Taiwan under grants 110-2628-E-002-002 and 111-2628-E-002-012.

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. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[2] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186. [Online]. Available: http://pmg.csail.mit.edu/papers/osdi99.pdf

[4] D. Ongaro and J. 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*. USENIX Association, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[5] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: https://doi.org/10.1145/279227.279229

[6] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "Algorand agreement: Super fast and partition resilient byzantine agreement," *IACR Cryptology ePrint Archive*, vol. 2018, p. 377, 2018.

[7] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience," *IACR Cryptology ePrint Archive*, vol. 2018, p. 1028, 2018. [Online]. Available: https://eprint.iacr.org/2018/1028

[8] T. H. Chan, R. Pass, and E. Shi, "Communication-efficient byzantine agreement without erasures," *CoRR*, vol. abs/1805.03391, 2018. [Online]. Available: http://arxiv.org/abs/1805.03391

[9] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *J. Cryptology*, vol. 18, no. 3, pp. 219–246, 2005. [Online]. Available: https://doi.org/10.1007/s00145-005-0318-0

[10] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. ACM, 2019, pp. 347–356. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[11] G. Wagner, "Double spending bug in polygon's plasma bridge," 2021. [Online]. Available: https://gerhard-wagner.medium.com/double-spending-bug-in-polygons-plasma-bridge-2e0954ccadf1

[12] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," in *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, 1983, pp. 1–7. [Online]. Available: https://doi.org/10.1145/588058.588060

[13] S. Agrawal and K. Daudjee, "A Performance Comparison of Algorithms for Byzantine Agreement in Distributed Systems," in *2016 12th European Dependable Computing Conference (EDCC)*, Sep. 2016, pp. 249–260.

[14] G. Liang, B. Sommer, and N. Vaidya, "Experimental performance comparison of byzantine fault-tolerant protocols for data centers," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 1422–1430.

[15] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-Glove Approach for BFT Testing," *arXiv e-prints*, p. arXiv:2004.10617, Apr. 2020.

[16] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. USENIX Association, 2009, pp. 153–168. [Online]. Available: http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf

[17] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire," in *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 2008, pp. 189–204. [Online]. Available: http://www.usenix.org/events/nsdi08/tech/full_papers/singh/singh.pdf

[18] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, Apr. 1988. [Online]. Available: https://doi.org/10.1145/42282.42283

[19] The LibraBFT Team, "State machine replication in the libra blockchain," https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain/2020-05-26.pdf, 2020, accessed: 2020-09-06.

[20] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman, "Cogsworth: Byzantine view synchronization," *arXiv e-prints*, 2019.

[21] R. Pass and E. Shi, "Hybrid consensus: Efficient consensus in the permissionless model," in *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, ser. LIPIcs, vol. 91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 39:1–39:16. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2017.39

[22] O. Foundation, "Node.js," 2009. [Online]. Available: https://nodejs.org/en/

[23] A. M. Law, *Simulation Modeling and Analysis, the 5th Edition*. McGraw-Hill Education, 2014.

[24] "Tendermint," https://tendermint.com/, accessed: 2017-06-30.

[25] "Hyperledger sawtooth pbft," https://github.com/hyperledger/sawtooth-pbft, accessed: 2017-06-30.

[26] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 2018.

[27] A. Momose, "Force-locking attack on sync hotstuff," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1484, 2019.