Scalable Performance Evaluation of Byzantine Fault-Tolerant Systems Using Network Simulation

Christian Berger
University of Passau
Passau, Germany
cb@sec.uni-passau.de

Sadok Ben Toumia
MaibornWolff GmbH
Munich, Germany
sadok.bentoumia@maibornwolff.de

Hans P. Reiser
Reykjavik University
Reykjavik, Iceland
hansr@ru.is

***Abstract*—Recent Byzantine fault-tolerant (BFT) state machine replication (SMR) protocols increasingly focus on scalability to meet the requirements of distributed ledger technology (DLT). Validating the performance of scalable BFT protocol implementations requires careful evaluation. Our solution uses *network simulations* to forecast the performance of BFT protocols while experimentally scaling the environment. Our method seamlessly plug-and-plays existing BFT implementations into the simulation without requiring code modification or re-implementation, which is often time-consuming and error-prone. Furthermore, our approach is also significantly cheaper than experiments with real large-scale cloud deployments. In this paper, we first explain our simulation architecture, which enables scalable performance evaluations of BFT systems through high-performance network simulations. We validate the accuracy of these simulations for predicting the performance of BFT systems by comparing simulation results with measurements of real systems deployed on cloud infrastructures. We found that simulation results display a reasonable approximation *at a larger system scale*, because the network eventually becomes the dominating factor limiting system performance. In the second part of our paper, we use our simulation method to evaluate the performance of PBFT and BFT protocols from the "blockchain generation", such as HotStuff and Kauri, in large-scale and realistic wide-area network scenarios, as well as under induced faults.**

***Index Terms*—simulation, performance, Byzantine fault tolerance, state machine replication, consensus**

I. Introduction

In the last years, distributed ledger technology (DLT) witnessed the following trend: Byzantine fault-tolerant (BFT)-based protocols like PBFT [1] have been envisioned to substitute the energy-inefficient Proof-of-Work [2] mechanism with a more efficient approach to achieving agreement between all correct blockchain replicas regarding which block to append next to the ledger [3]. While traditional BFT protocols like PBFT can accomplish this task, the cost of running agreement among a large number of replicas results in a sharp performance decline in large-scale systems as shown in Figure 1. To address the scalability challenges of BFT, many new protocols have seen the light of day [4]–[9].

Cloud-scale deployments: Asserting that these novel BFT protocols can provide sufficient performance in realistic, large-scale systems, requires careful evaluation of their run-time behavior. For this purpose, research papers describing these protocols contain evaluations with large-scale deployments that are conducted on cloud infrastructures like AWS, where

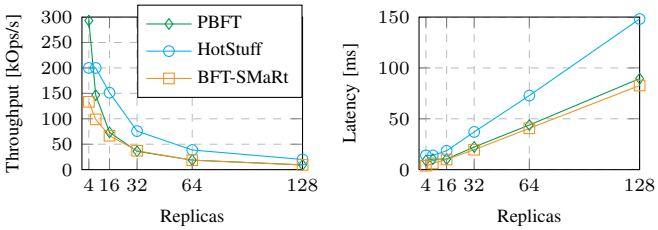


Figure 1: Simulation results of BFT protocols for 1 KiB payload in a data center environment with 10 Gbit/s bandwidth. This is the setup in which HotStuff has been evaluated [10].

experiments deploy up to several hundred nodes (for instance see [5]–[7], [9], [10] and many more) to demonstrate a BFT protocol's performance at large-scale.

Evaluations using real protocol deployments usually offer the best realism but can be costly and time-consuming, especially when testing multiple configurations. Thus, an interesting alternative for cheap and rapid validation of BFT protocol implementations (that are possibly still in the development stage) can be to predict system performance with simulations.

Simulating BFT protocols: BFTSim [11] is the first simulator that was developed for an eye-to-eye comparison of BFT protocols, but it lacks the necessary scalability to be useful for the newer "blockchain generation" of BFT protocols (and apparently only up to $n = 32$ PBFT replicas can be successfully simulated [12]). Moreover, BFTSim demands a BFT protocol to be modeled in the P2 language [13], which is somewhat error-prone considering the complexity of BFT protocols and also time-consuming. A more recent tool [12] allows for scalable simulation of BFT protocols, but it unfortunately also requires a complete re-implementation of the BFT protocol in JavaScript. Further, this tool cannot make predictions on the system throughput and thus its performance evaluation is limited to observing latency.

In our approach, we address a critical gap in the state-of-the-art by introducing a simulation method for predicting BFT system performance that is highly scalable, without necessitating any re-implementation of the protocol. This key feature distinguishes our approach from conventional methods and represents a significant advancement for predicting BFT system performance in a practical way.

Why not just use network emulation?: Emulation tries to duplicate the exact behavior of what is being emulated. Emulators like Kollaps can be used to reproduce AWS-deployed experiments with BFT protocols on a local server farm [14]. A clear advantage of emulation is how it preserves realism: BFT protocols still operate in real time and use real kernel and network protocols. In contrast to simulation, emulation is not similarly resource-friendly, as it executes the real application code in real-time, thus requiring many physical machines at hand to conduct large-scale experiments.

Simulation as (better?) alternative: Simulation decouples simulated time from real time and employs abstractions that help accelerate executions: Aspects of interest are captured through a model, which means the simulation only mimics the protocol's environment (or also its actual protocol behavior if the application model is re-implemented). This has the advantage of easier experimental control, excellent reproducibility (i.e., deterministic protocol runs), and increased scalability when compared to emulation.

These benefits of emulation come at some cost: As a potential drawback remains the question of the validity of results, since the model may not fairly enough reflect reality. Furthermore, another existing limitation of all current BFT simulators is the need to modify or (usually) re-implement the BFT protocol to use it within a simulation engine.

Our contributions: In our approach, we address these limitations and aim at making simulation a useful approach for large-scale BFT protocol performance prediction:

- We define a software architecture for high-performance and scalable network simulation, in which we can plug an *existing, unmodified* BFT protocol implementation into a simulation, without requiring any re-implementation or source code modifications. By doing this, we ensure validity on the application level, since the actual application binaries are used to start real Linux processes that are finally connected into the simulation engine.
- A threat to validity is the fact that when we solely rely on network simulation, it neglects the impact of processing time due to CPU-intensive tasks of BFT protocols on performance, namely, signature generation, and verification. We conducted experiments that show that the performance results of simulations can display a useful approximation to real measurements in large-scale systems. This is because, at a certain number of replicas (often as soon as $n \geq 32$), the overall system performance is mostly dictated by the underlying network, which persists as a performance bottleneck in the system. We provide more detailed insights on this in our validity analysis in Section III-D.
- To demonstrate the usability of our method, we use BFT protocols from the "blockchain generation", namely HotStuff and Kauri, to conduct simulations at a large scale.

This paper is structured as follows: In Section II, we briefly review the basics of BFT protocols to guide the reader through the different communication strategies that these protocols employ. In Section III, the main part of our paper, we explain our methodology, which includes both the design of our simulation architecture and the validation of simulation results using real measurements for comparison. In Section IV, we evaluate the performance of selected BFT protocols under varying boundary conditions. In respect to a possible blockchain use case, we construct large-scale and realistic wide-area network scenarios with up to $n = 256$ replicas and heterogenous network latencies derived from real planetary-scale deployments, and, in some scenarios, with failures. We envision an apples-to-apples comparison of the performance of scalable BFT protocols in realistic networks. Finally, we summarize related work in Section V and conclude in Section VI.

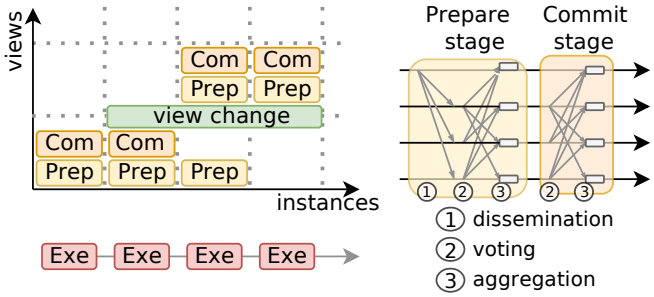


Figure 2: A simplified model for BFT SMR protocols [15].

II. BFT Protocols

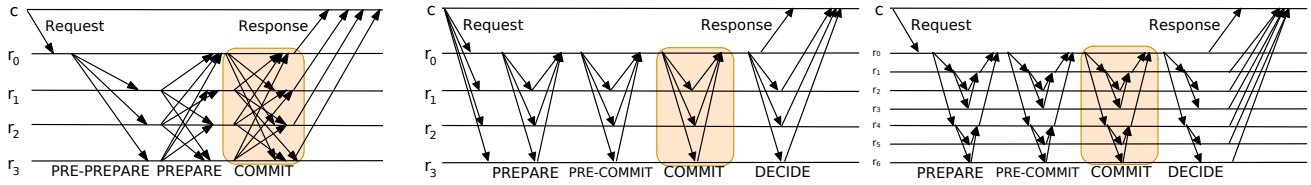
BFT state machine replication (SMR) protocols achieve fault tolerance by coordinating client interactions with a set of independent service replicas [16]. The replicated service remains functional as long as the number of faulty replicas does not surpass a threshold t out of n replicas. In BFT SMR, replicas order operations issued by clients, preserve a consistent state, and then provide matching responses to the client, which needs to collect at least $t+1$ matching responses from different replicas to assert the correctness.

Running agreements: Replicas repeatedly agree on a block of operations. To complete a single agreement instance, replicas must go through multiple phases. In our approach, we model each of the phases such that it consists of several components:

1) *Dissemination* of one or more proposals: In some cases, this component can be omitted if the phase uses the output of the previous phase as its input.
2) *Confirmation:* This component requires voting to confirm a proposal.
3) *Aggregation*: Votes that match across different replicas are collected to form a quorum certificate.

Figure 2 illustrates these phases for the PBFT protocol. The first phase, PREPARE, encompasses both dissemination (PRE-PREPARE messages) and confirmation (PPREPARE messages).

Quorum certificates contain votes from sufficient replicas to guarantee that no two different blocks can receive a certificate, thus *committing* the block. After an agreement completes, replicas execute the operations.



(a) **PBFT**'s agreement operation is fast: It consists of only three communication steps but incurs $O(n^2)$ communication complexity.

(b) The **HotStuff** leader collects the votes from the other replicas and distributes quorum certificates to all others.

(c) In **Kauri**, the leader uses a balanced communication tree to disseminate a proposal and to aggregate and disseminate votes.

Figure 3: Communication patterns of different BFT protocols: (a) all-to-all, (b) linear (star), and (c) tree.

Orthogonal to the agreement instances, replicas operate in *views*, which define a composition of replicas, select leader(s), and in some cases establish the dissemination pattern. Some protocols re-use a single view under a stable leader (as long as agreement instances finish), as shown in Figure 2, but others may change the view for each stage or instance. A *view change* phase synchronizes replicas, replaces the leader, and eventually ensures liveness by installing a new leader under whose regency, agreement instances succeed.

PBFT: In 1999, Castro and Liskov proposed the Practical Byzantine Fault Tolerance (PBFT) protocol which became known as the first practical approach for tolerating Byzantine faults [1]. PBFT's practicality comes from its optimal resilience threshold ($t = \frac{n-1}{3}$) and its high performance, comparable to non-replicated systems. We show the message flow of PBFT's normal operation in Figure 3a. In PBFT, the leader collects client operations in a block and broadcasts the block in a PRE-PREPARE message to all replicas. Subsequently, all replicas vote and collect quorum certificates through the messages PREPARE and COMMIT which are realized as all-to-all broadcasts. PBFT does not scale well for larger system sizes, because all operations are tunneled through a single leader, who must disseminate large blocks to all of the other replicas. This makes the leader's up-link bandwidth a bottleneck for the whole system's performance. Further, PBFT's all-to-all broadcasts incur $O(n^2)$ messages (and authenticators) to be transmitted in the system.

HotStuff: The HotStuff leader implements linear message complexity by gathering votes from all other replicas and disseminating a quorum certificate [4]. To reduce the cost of transmitting message authenticators, the leader can use a simple aggregation technique to compress $n - t$ signatures into a single fixed-size threshold signature. This threshold signature scheme uses the quorum size as a threshold, and a valid threshold signature implies that a quorum of replicas has signed it. Consequently, the threshold signature has a size of $O(1)$, which is a significant improvement over transmitting $O(n)$ individual signatures.

The original implementation of HotStuff (we refer to it as HOTSTUFF-SECP256K1) is based on elliptic curves and does not feature signature aggregation (i.e., combining multiple signatures into a single signature of fixed size). Later, an implementation was made available by [7] that uses BLS signatures [17] (which we refer to as HOTSTUFF-BLS) that features signature aggregation.

As depicted in Figure 3b, the communication flow remains imbalanced where each follower replica communicates exclusively with the leader (which is the center of a *star* topology), while the leader has to communicate with all other replicas.

Kauri: The use of a tree-based communication topology offers an advantage as it distributes the responsibility of aggregating and disseminating votes and quorum certificates, thus relieving the leader. Kauri [7] is a tree-based BFT SMR protocol (see Figure 3c) that introduces a timeout for aggregation to address leaf failures.

To handle the failure of internal tree nodes, Kauri employs a reconfiguration scheme, which guarantees to find a correct set of internal nodes, given that the number of failures lies below a certain threshold.

The added latency caused by the additional number of communication steps is mitigated through a more sophisticated pipelining approach (that can start several agreement instances per protocol stage) than the pipelining mechanism employed in HotStuff which launches only a single agreement instance for each protocol stage.

III. METHODOLOGY

In this section, we first justify and motivate our ambition of evaluating actual BFT protocol implementations through a simulation of the running distributed system. In this simulation, *replica* and *client* components are instantiated using the provided protocol implementations and are co-opted into an event-based simulation that constructs and manages the system environment. Moreover, we explain the selection of protocols, that we made in Section II.

Subsequently, we describe the software architecture of our simulation approach. The approach involves the user inputting a simple experimental description file (EDF), specified in YAML format, to a frontend. The frontend then prepares all runtime artifacts, creates a realistic network topology, and schedules a new experiment. This scheduling is done by launching an instance of the backend, which runs the network simulation. A detailed overview is displayed in Figure 4.

After that, we validate simulation results by comparing them with measurements from real setups that we mimicked.

A. Why Simulate BFT Protocol Implementations?

One of the main benefits of our concrete methodology is the plug-and-play utility. This means we can guarantee application

realism because the actual implementation is used to start real Linux processes which serve as the application model, thus duplicating actual implementation behavior. In particular, it means in regard to the application level the overall approach can be considered an emulation. At the same time, users experience no re-implementation or modeling effort which can easily introduce errors due to the high complexity of BFT protocols and is also time-consuming.

Furthermore, simulating actual BFT protocol *implementations* rather than specifically crafted models is useful for the purpose of rapid prototyping and validation. Some implementation-level bugs in BFT systems might not occur in "common" $n = 4$ scenarios, and thus simulations can be utilized to conduct integration tests at a larger scale. Similarly, they can be employed by developers for automatic regression tests thinking "*did my last commit negatively affect the protocol performance at a larger scale?*".

Furthermore, there are some advantages that generally exist when using simulations. First of all, simulations make it easy to investigate the run-time behavior of BFT protocols in an inexpensive way, much cheaper than real-world deployments. Nowadays more and more BFT protocol implementations are published open-source. Simulations make it easy to compare these protocols under common conditions and fairly reason about their performance in scenarios in which performance becomes network-bound. In particular, it is possible to explore the parameter space of both protocol parameters and network conditions in a systematic way and in a controlled environment, which even produces deterministic results.

Moreover, in our methodology, we can create large network topologies by using latency maps (such as those provided by Wonderproxy[1]) which can provide more regions than what most cloud providers, e.g., AWS can offer.

Lastly, simulation can also serve a didactical purpose. Simulations can help to achieve a better understanding of how BFT protocols behave at a large scale. For instance, the methodology can be used to support teaching in distributed system labs at universities by letting students gain hands-on experience with a set of already implemented BFT protocols.

BFT Protocol Selection: We justify the selection of BFT protocols in the following way: Our main ambition was to showcase the impact of different communication strategies (i.e., all-to-all, star, and tree) towards system performance, and thus we selected a single "representative" BFT protocol for each strategy (namely PBFT, HotStuff, and Kauri, respectively). As part of future work, we plan to extend evaluations to multi-leader and leaderless BFT protocols which can be also evaluated by following our methodology.

Phantom as Choice for the Backend: We chose Phantom [18] for the part of the backend that finally conducts the simulations. The main reason lies in its high performance, hybrid simulation/emulation architecture which offers the possibility to directly execute applications (thus benefiting realism) while still running them in the context of a cohesive network simulation [18]. We conducted a comparison with other approaches in an earlier workshop paper [19].

B. The Frontend: Accelerating Large-Scale Simulations of BFT Protocol Implementations

Conducting large-scale simulations of BFT protocols requires tackling a set of challenges first. This is because of the following reasons:

1) The simulation quality depends on realistic and large network topologies for arbitrary system sizes. The characteristics of their communication links should ideally resemble real-world deployments. This is crucial to allow realistic simulation of wide-area network environments.
2) We need aid in setting up the BFT protocol implementations for their deployment, since bootstrapping BFT protocol implementations in a plug-and-play manner involves many steps that can be tedious, error-prone, and protocol-specific. Examples include the generation of protocol-specific run-time artifacts like cryptographic key material, or configuration files which differ for every BFT protocol.
3) When developing and testing BFT algorithms, different combinations of protocol settings result in numerous experiments being conducted. Since simulations run in virtual time, they can take hours, depending on the host system's specifications. For the sake of user experience and convenience, we find it necessary for experiments to be specified in bulk and run sequentially without any need for user intervention.
4) We may want to track and evaluate resources needed during simulation runs, such as CPU utilization and memory usage.

These reasons led us to develop a *frontend*[2], a tool on top of the Phantom simulator [18] to simplify and accelerate the evaluation of unmodified BFT protocol implementations.

Experimental Description and Frontend Design: The *frontend* is composed of several components (see Figure 4) and follows a modular architecture, in that it is not tailored to a specific BFT protocol, but is easily extensible.

Scheduler: The toolchain is administered by a scheduler that manages all tools, i.e., for preparing an environment, configuring runtime artifacts for a BFT protocol, and initializing a resource monitor. The scheduler invokes protocol connectors to set up a BFT protocol and loads *experiments description files* (see Figure 5 for an example that specifies a single experiment) which contain a set of experiments to be conducted for the specified BFT protocol. Finally, it starts Phantom, once an experiment is ready for its execution.

The scheduler also initializes a resource monitor to collect information on resource consumption (like allocated memory and CPU time) during simulation runs and also the total simulation time. These statistics can serve as indicators of a possible need for vertically scaling the host machine and as estimates for the necessary resources to run large simulations.

[1] See https://wonderproxy.com/blog/a-day-in-the-life-of-the-internet/ to obtain information on these statistics.

[2] All of our code and the experiment files are open-source available on GitHub (https://github.com/Delphi-BFT/tool).

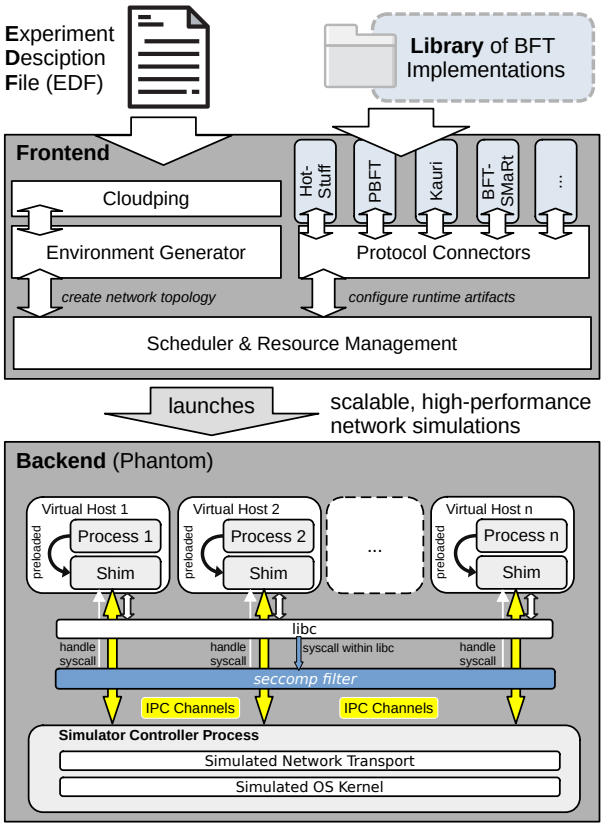


Figure 4: Architecture employed in our simulation method.

Environment Generator: The environment generator creates network topologies as a complete graph for any system size, resembling realistic LAN or WAN settings. To replicate the geographic dispersion of nodes realistically, the environment generator employs a cloudping component, which retrieves real round-trip latencies between all AWS regions from Cloudping[3]. This allows the tool to create network topologies that resemble real BFT protocol deployments on the AWS cloud infrastructure. We also implemented a larger latency map that uses 51 distinct locations sourced from Wonderproxy's latency statistics. The cloudping component can either load an up-to-date latency map from an online source or use one of the existing ones from the repository. Note that using the same latency map is necessary for maintaining determinism and thus a requirement for reproducibility. The `EDF.network` description defines the distribution of replicas and clients on a latency map and configures bandwidth and packet loss.

Protocol Connectors: For each BFT protocol implementation that we want to simulate, it is necessary to create protocol configuration files and necessary keys. Since protocol options and cryptographic primitives vary depending on the concrete BFT protocol, we implement the protocol-specific setup routine as a tool called protocol connector, which is invoked by the scheduler. A connector must implement the methods `build()` and `configure()`. This way, it is simple to extend our toolchain and support new BFT protocols, as it only requires writing a new protocol connector (in our experience this means writing between 100 and 200 LoC).

[3]See https://www.cloudping.co/grid.

```
EDF.network
bandwidthUp: 25 Mibits
bandwidthDown: 25 Mibits
latency:
 map: 'aws21'
 replicas: ['eu-west-1': 2, 'us-east-1': 1,
            'sa-east-1': 1]
 clients: ['ca-central-1': 2]
packetLoss: 0.0

EDF.replica
replicas: 4
blockSize: 400
replySize: 1024
timeout: 4000

EDF.client
clients: 16
numberOfHosts: 2
requestSize: 1024
outStandingPerClient: 175

EDF.faults
type: crash
threshold: 0.25
timestamp: 60 s
```

Figure 5: Example for an experimental description file (EDF).

Fault Induction: Our frontend can also induce faults during simulation runs which is important to reason about the performance in "uncivil" scenarios. Since BFT protocols sometimes employ different resilience thresholds, we allow the user to specify a desired threshold of replicas in which faults are induced. We model a static threshold adversary as often assumed by BFT protocols.

A simple and currently supported fault type is `type: crash` which terminates the faulty replica processes at a specific `timestamp` within the simulation.

Another scenario that we can run is a denial-of-service attack by setting `type: dos` and specifying an `overload` parameter, which leads to a malicious client being instantiated that sends a larger number of requests (a multiple, e.g., $100\times$ of what the normal clients send), to test if implementations can withstand such a scenario, i.e., by limiting the number of requests accepted from a single client and ensuring a fair batching strategy. Further, we support `packetloss` which describes the ratio of packets to be dropped on the network level (however this setting currently needs to be configured in the `EDF.network` section) to assess how well BFT implementations can perform when networks behave lossy.

In future work, we want to explore more sophisticated (Byzantine) fault behavior, for instance, by seeking inspiration from the Twins [20] methodology. Twins is a unit test case generator for Byzantine behavior by duplicating cryptographic IDs of replicas (e.g., leading to equivocations).

C. The Backend: Using Phantom to Simulate BFT Protocols as Native Linux Processes

As *backend*, we use Phantom, which employs a hybrid simulation/emulation architecture, in which real, unmodified applications execute as normal processes on Linux and are hooked into the simulation through a system call interface using standard kernel facilities [18]. An advantage of this is that this method preserves application layer realism as real

BFT protocol implementations are executed. At the same time, Phantom is resource-friendly and runs on a single machine.

By utilizing its hybrid architecture, Phantom occupies a favorable position between the pure simulator ns-3 [21] and the pure emulator Mininet [22]. It maintains sufficient application realism necessary for BFT protocol execution while exhibiting greater resource-friendliness and scalability compared to emulators. Because Phantom strikes a balance that caters to the needs of BFT protocol research, we chose it as the backend for conducting protocol simulations.

Simulated Environment: In Phantom, a network topology (the *environment*) can be described by specifying a graph, where *virtual hosts* are nodes and communication links are edges. The graph is attributed: For instance, virtual hosts specify available uplink/downlink bandwidth and links specify latency and packet loss. Each virtual host can be used to run one or more applications. This results in the creation of real Linux processes that are initialized by the simulator controller process as managed processes (managed by a Phantom worker). The Phantom worker uses `LD_PRELOAD` to preload a shared library (called the *shim*) for co-opting its managed processes into the simulation (see Figure 4). `LD_PRELOAD` is extended by a second interception strategy, which uses `seccomp`[4] for cases in which preloading does not work.

Simulation Engine: The shim constructs an inter-process communication channel (IPC) to the simulator controller process and intercepts functions at the system call interface. While the shim may directly emulate a few system calls, most system calls are forwarded and handled by the simulator controller process, which simulates kernel and networking functionality, for example, the passage of time, I/O operations on `file`, `socket`, `pipe`, `timer`, event descriptors and packet transmissions.

Deterministic Execution: Throughout the simulation, Phantom preserves determinism: It employs a pseudo-random generator, which is seeded from a configuration file to emulate all randomness needed during simulation, in particular, the emulation of `getrandom` or reads of `/dev/*random`. Each Phantom worker only allows a single thread of execution across all processes it manages so that each of the remaining managed processes/threads are idle, thus preventing concurrent access of managed processes' memory [18].

In our workflow, Phantom is invoked by the Scheduler as soon as a new simulation experiment is ready for its execution and the host's hardware resources are available.

D. Validation

In this section, we compare measurements of real BFT protocol runs with results that we achieve through simulations.

1) HotStuff-secp256k1*:* In our first evaluation, we try to mimic the evaluation setup of the HotStuff paper (the arXiv, version see [10]) to compare their measurements with our simulation results. Their setup consists of more than a hundred virtual machines deployed in an AWS data center; each machine has up to 10 Gbit/s bandwidth and there is less than 1 ms latency between each pair of machines (we use 1 ms in the simulation). The employed block size is 400. We compare against two measurement series: "p1024" where the payload size of request and responses is 1024 bytes and "10ms" with an empty payload, but the latency of all communication links is set to 10 ms. Our goal is to investigate how faithfully the performance of HotStuff can be predicted by regarding only the networking capabilities of replicas, which manifests at the point where the network becomes the bottleneck for system performance.

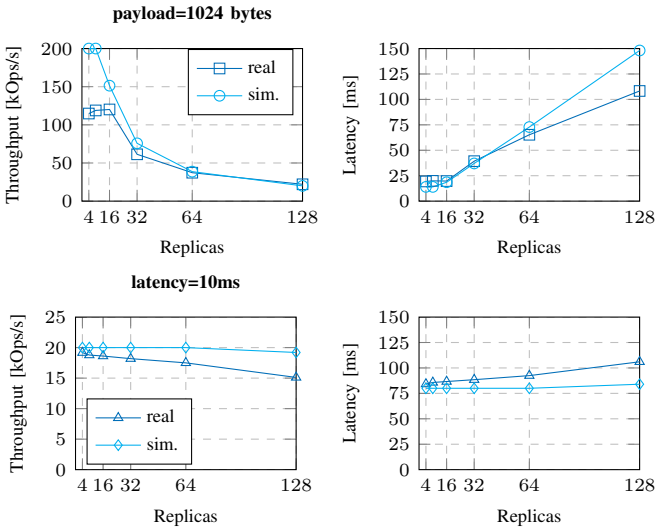


Figure 6: Performance results of HotStuff-secp256k1 vs. its simulated counterpart using a bandwidth of 10 Gbit.

Observations. We display our results in Figure 6. The simulation results for the payload experiment indicate a similar drop in performance as the real measurements for $n \geq 32$. For a small-sized replica group, the network simulation predicts higher performance: 200k op/s. This equals the theoretical maximum limited only through the 1 ms link latency which leads to pipelined HotStuff committing a block of 400 requests every 2 ms. The difference in throughput decreases once the performance of HotStuff becomes more bandwidth-throttled (at $n \geq 32$). We also achieve close results in the "10ms" setting: 80 ms in the simulation vs 84.1 ms real, and 20k op/s in the simulation vs. 19.2k op/s real for $n = 4$; but with an increasing difference for higher n, i.e., 84 ms vs. 106 ms and 19k.2 op/s vs. 15.1k op/s for $n = 128$. The problem is that this experiment does not use any payload which makes the performance less sensitive to a network bottleneck (which is usually caused by limited available bandwidth).

2) BFT-SMaRt *and* PBFT*:* In our next experiment we validate BFT-SMaRt and PBFT[5], by using measurements taken from [10] and conducting our own experiment on a WAN which is constructed using four different AWS regions.

[4]Installing a secure computing (i.e., seccomp) filter in a process allows interposition on system calls that are not preloadable, see [18] for more details.

[5]We use a Rust-based implementation of PBFT (github.com/ibr-ds/themis), since the original by Castro et al. [1] does not compile on modern computers.

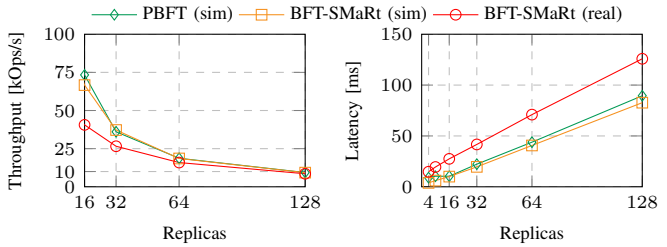
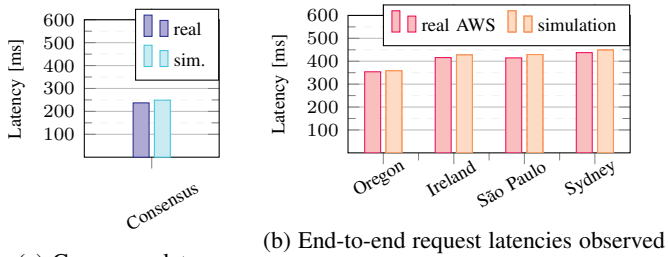


Figure 7: Performance of simulated BFT-SMaRt, simulated PBFT and a real BFT-SMaRt execution in a 10 Gbit LAN.



(a) Consensus latency.

(b) End-to-end request latencies observed by clients in AWS regions.

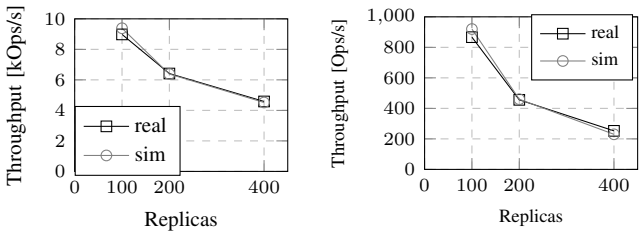
Figure 8: Comparison of a real BFT-SMaRt WAN deployment on the AWS infrastructure with its simulated counterpart.

LAN environment: To begin with, we mimic the "p1024" setup from [10] (as the real measurement data we found for BFT-SMaRt is from [10]), thus creating an environment with 1 ms network speed and 10 Gbit/s bandwidth. We simulate both protocols: BFT-SMaRt and PBFT, because they utilize an identical normal-case message pattern.

Observations. We display our results in Figure 7. We observe that initially ($n \leq 32$) the real BFT-SMaRt performance results are quite lower than what our simulations predict. This changes with an increasing n, i.e., at $n = 128$, we observe 9.280 op/s for BFT-SMaRt (real measurement: 8.557 op/s) and 9.354 op/s for our PBFT implementation. In the latency graph, we observe a noticeable gap between real and simulated BFT-SMaRt. The main reason for this is that we could not exactly reproduce the operation sending rate from [10] as it was not explicitly stated in their experimental setup.

Geo-Distribution: Next, we experiment with geographic dispersion, putting each BFT-SMaRt replica in a distinct AWS region. Our experimental setup is similar to experiments found in papers that research latency improvements (see [23]–[25]). We employ a $n = 4$ configuration and choose the regions Oregon, Ireland, São Paulo, and Sydney for the deployment of a replica and a client application each. We run clients one after another, and each samples 1000 requests without payload and measures end-to-end latency, while the leader replica (in Oregon) measures the system's consensus latency.

Observations. We notice that consensus latency is only slightly higher in the simulation (237 ms vs. 249 ms), and further, the simulation results also display slightly higher end-to-end request latencies in all clients (see Figure 8). The deviation between simulated and real execution is the lowest in Oregon (1.3%) and the highest in São Paulo (3.5%).



(a) KAURI.

(b) HOTSTUFF-BLS.

Figure 9: Reproducing the "`global`" scenario from [7] that uses 100 ms inter-replica latency and 25 Mbit/s bandwitdh.

3) KAURI and HOTSTUFF-BLS: Moreover, we mimic the `global` experiment from Kauri [7], which uses a varying number of 100, 200, and 400 replicas. The global setup assumes replicas being connected over a planetary-scale network, in which each replica possesses only 25 Mbit/s bandwidth and has a latency of 100 ms to every other replica. We validate two implementations that were made by [7]: HOTSTUFF-BLS, an implementation of HotStuff which uses BLS instead of SECP256K1 (this the originally implemented version of HotStuff), and KAURI which enriches HOTSTUFF-BLS through tree-based message dissemination (and aggregation) and an enhanced pipelining scheme.

Observations. Figure 9 shows our results. Overall, we observe that for both implementations, the results we could obtain for system throughput are almost identical. At $n = 400$, for KAURI, we observe 4518 op/s (real: 4584 op/s), and for HOTSTUFF-BLS it is 230 op/s (real: 252.67 op/s).

We also evaluated on the latency of Kauri deciding blocks (as in Figure 8 from [7]) comparing with the $n = 100$ and 25 Mbit/s latency experiment. While the real experiment in the Kauri paper reports a latency of 563 ms, in the simulation, deciding a block seems to take at least 585 ms.

4) Resource Consumption and Implementations: Further, we investigate how resource utilization, i.e., memory usage and simulation time, grows with an increasing system scale. For this purpose, we use the HOTSTUFF-SECP256K1 "10ms" simulations (which display a somewhat steady system performance for increasing system scale) on an Ubuntu 20.04 VM with 214 GB memory and 20 threads (16 threads used for simulation) on a host with an Intel Xeon Gold 6210U CPU at 2.5 GHz. We observe that active host memory and elapsed time grow with increasing system scale (see Figure 10). Based on the practically linear increase in memory utilization in Figure 10, we estimate that 512 replicas will need about 64 GiB memory, and it should be feasible to simulate up to 512 HotStuff replicas with a well-equipped host.

During our validations, we tested several open-source BFT frameworks (see Table I) which have been written in different programming languages (C++, Rust, Java). From our experience, the Java-based BFT-SMaRt library was the most memory-hungry implementation, but we were still able to

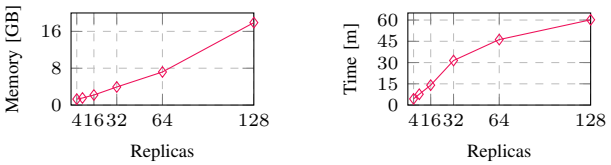


Figure 10: Resource consumption of simulations.

simulate $n = 128$ replicas on our commodity hardware server without problems, which strengthens our belief in the scalability and resource-friendliness of our methodology.

Table I: BFT protocol implementations that we simulated.

framework	BFT protocol	language	repo on github.com
libhotstuff [10]	HotStuff [10]	C++	/hot-stuff/libhotstuff
themis [26]	PBFT [1]	Rust	/ibr-ds/themis
bft-smart	BFT-SMaRt [27]	Java	/bft-smart/library
hotstuff-bls [7]	HotStuff [10]	C++	/Raycoms/Kauri-Public/
kauri	Kauri [7]	C++	/Raycoms/Kauri-Public/

IV. Experimental Results

In this section, we compare different BFT protocol implementations under varying border conditions:

1) *failure-free*: Benchmark protocols in failure-free execution for increasing system size and varying block size
2) *packetloss*: The simulated network behaves lossy
3) *denial-of-service attack*: A specific client tries to overload the system by submitting too many operations
4) *crashing-replicas*: Similiar to (1) but with induced crash faults at a certain point of simulated time

The high-level goal is to present an apples-to-apples comparison of BFT implementations in a controlled environment.

A. General Setup

In our controlled environment, we simulate a heterogeneous planetary-scale network, using 21 regions from the AWS cloud infrastructure, as shown in Figure 11. Latencies are retrieved from real latency statistics by the *cloudping* component of our frontend. Replicas vary in number and are evenly distributed across all regions. We use a 25 Mbit/s bandwidth rate, consistent with related research to model commodity hardware in world-spanning networks (e.g., see the `global` setup of [7]). We utilize a variable number of clients to submit requests to the replicas. Specifically, we carefully select both the client count and concurrent request rate to fully saturate and thereby maximize[6] the observable system throughput.

In our simulations, we use the protocols PBFT, HotStuff-secp256k1, HotStuff-bls, and Kauri. Operations carry a payload of 500 bytes (roughly the average size of a Bitcoin operation), and the default block size is 1000 operations unless stated otherwise. Each simulation deploys replicas and clients and then runs the BFT protocol for at least 120 seconds of simulated time within the environment.

[6]The number of concurrently submitted requests is sufficient to (1) fill the block size of each block that a BFT protocol processes in parallel, and (2) a *block size* of *pending requests* remains to wait at the leader so the next block can be disseminated immediately as soon as an ongoing consensus finalizes.

Figure 11: The `aws21` map mimics a planetary-scale deployment on the AWS infrastructure, with replicas spread across 21 regions, and clients (c) submitting requests to replicas.

B. Failure-Free Scenario: Scalability and Block Size

In our first experiment, we evaluate the baseline performance of the BFT protocols in our constructed environment assuming that no failures happen. Further, we repeat each simulation while varying the system size n (namely, using a total of 64, 128, and 256 replicas) and varying the block size (using both the default size of 1000 operations and a block size that is more optimal for a protocol in respect to observing lower latency). We denote variations in the employed block size by adding the postfix "-blockSize" to the protocol name.

Observations: We present our results in Figure 12. In particular, we can observe striking differences in BFT protocol performance being in different orders of magnitudes.

Kauri-1k displays the highest performance among all protocol implementations. At $n = 256$, Kauri achieves a throughput increase of almost $20\times$ over HotStuff-secp256k1 in our heterogeneous setup (4917 op/s vs. 246 op/s). On a side note, the evaluations of Kauri report a possible increase of *up to* $28\times$ over HotStuff in a setup created entirely with homogeneous latencies [7].

We further observe a surprisingly low performance of our tested PBFT-1k implementation (75 op/s and a latency of 23.45 s at $n = 64$). The problem of this implementation is that it includes full operations in blocks, causing long dissemination delays for large blocks over the limited 25 Mbit/s links. Other protocol implementations include only SHA-256 hashes of operations in the blocks, which then accelerates proposal dissemination time.

For the purpose of a fair comparison, we simulate this PBFT implementation as if it would use the "*big request" optimization*[7] [1] (and denote it by PBFT-opt), which achieves 466 op/s and a latency of 3.7 s at $n = 64$. HotStuff-secp256k1 still beats the PBFT implementation we used because of its use of pipelining and re-use of quorum certificates during aggregation which can improve performance in this setting.

Notably, varying the block size impacts observed latency. Smaller blocks can be more quickly disseminated which then decreases latency, in particular, if clients operate in a closed loop, i.e., only issue a constant number of k operations

[7]The big request operation in PBFT substitutes larger requests by a hash value and only inlines small operations into a block (*batch* in PBFT parlance).

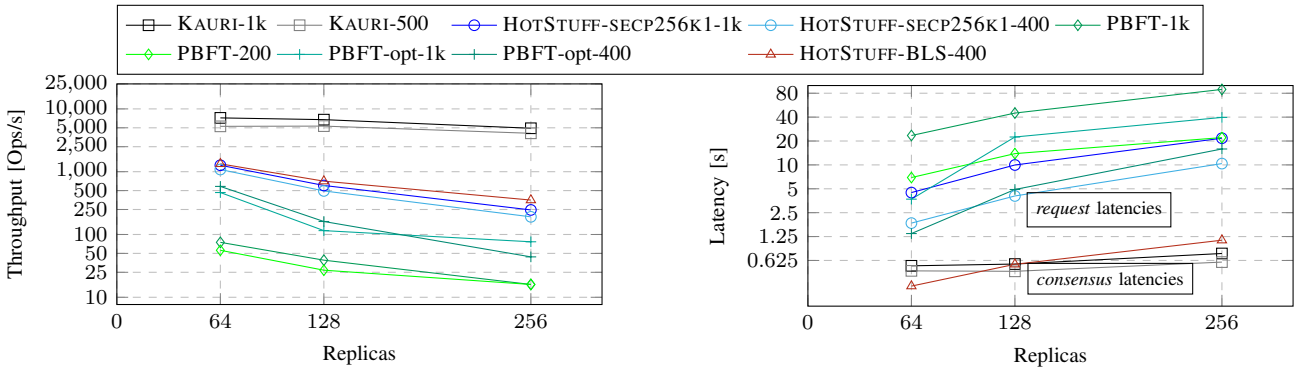


Figure 12: Performance of BFT protocols in a geo-distributed, fault-free scenario using 25 Mbit/s network links.

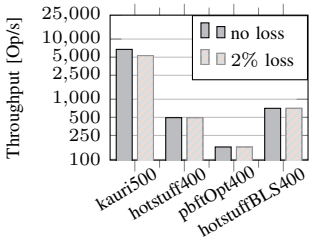


Figure 13: Packet loss.

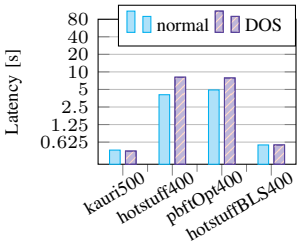


Figure 14: DOS attack.

concurrently then wait for obtaining responses and completing pending operations before submitting new operations. We also observe that at $n = 256$, the BLS implementation of HotStuff-400 increases throughput by $1.85\times$ over its implementation with SECP256K1 (353 vs. 191 op/s).

C. Packet Loss

In our next experiment, we study the impact of lossy network links on system throughput. For this purpose, we employ the same environment as in the last section with a fixed size of $n = 128$ replicas but introduce a packet loss of 2% on every network link.

Observations: We display our simulation results in Figure 13. Overall, we observe the same protocol performance for PBFT, and HOTSTUFF-SECP256K1 while the throughput of KAURI only slightly drops. We conclude that packet loss does not seem to impact BFT protocol performance much.

D. Denial-of-Service Attack

In this experiment, we examine how well BFT protocol implementations can tolerate specific clients trying to overload the system. We use the usual $n = 128$ replicas setup. To overload the system, we let a specific "malicious" client submit a larger number (i.e., $10\times$ more than in Sect. IV-B) of outstanding operations to the system during a short time interval of 120 s and investigate the impact on request latency that normal clients observe.

Observations: We show our simulation results in Figure 14. More outstanding client requests lead to higher observed latency in PBFT (4.9s to 15.2s) and HOTSTUFF-SECP256K1 (4.1s to 8.2s). This is because submitted requests are queued and need to wait for an increasing amount of time to be processed. Interestingly, we found almost identical latency results for KAURI and HOTSTUFF-BLS. After looking into the specifics of their implemented benchmark application, it became clear to us that these implementations only report *consensus* latency of replicas and not the end-to-end *request* latency observed by clients. In this light, the latency results obtained are – at least for this experiment – not helpful for a direct comparison[8].

None of the tested implementations had mechanisms in place that would prevent overloading the system, e.g., limiting the number of requests that are accepted from a single client.

E. Crashing Replicas

Finally, we investigate the crash fault resilience of our tested protocol implementations, using our usual $n = 128$ replicas setup. We induce a crash fault at the leader replica at time $\tau = 60$ s. During our simulations, we noticed that the PBFT implementation's[9] *view change* did not work properly. After contacting the developers we received a patch (a recent commit was missing in the public GitHub repository) that resolved the issue, at least for smaller systems. This illustrates how our methodology can help detect protocol implementation bugs.

Observations: We show our results in Figure 15. The failover time of a protocol generally depends on its timeout parameterization and we cannot exclude that tighter timeouts could have been possible (although noticeable, in [7] it is stated that Kauri can use more aggressive timeout values than Hotstuff). Notably, we observe a few interesting behaviors: In HotStuff, after the failover, the new HotStuff leader pushes a

[8]Note that we would have to modify the benchmarking application of Kauri to obtain *request latency* results that are comparable with the other protocols. A small takeaway message is that different BFT protocol implementations might use slightly different metrics in benchmark suites. It requires caution when comparing results but it is not a hindrance to our general approach.

[9]This statement only applies to the Rust-based implementation of PBFT (github.com/ibr-ds/themis) we tested, not the original by Castro et al. [1].

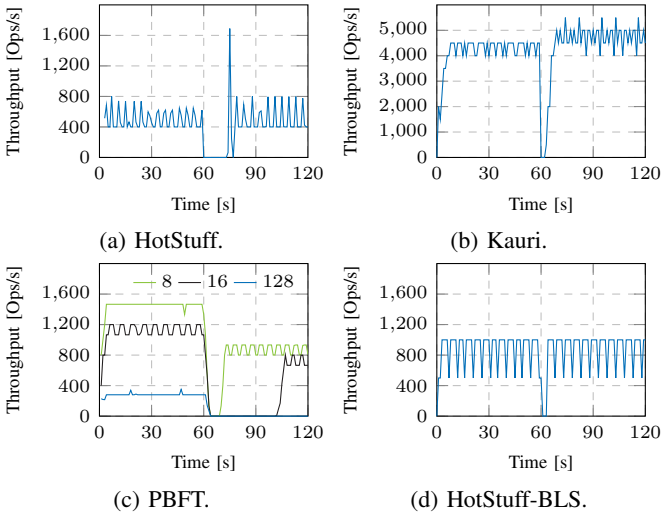


Figure 15: Inducing a single crash fault in the leader.

larger block leading to a throughput spike: It tries to commit the large block fast by building a three-chain in which the large block is followed by empty blocks (this leads to a short, second throughput drop to 0). After that, the throughput stabilizes. In our observation, the new Kauri leader can more quickly recover protocol performance than in HotStuff. The throughput level seems to be slightly higher which is because the new leader seems to be located in a more favorable region of the world (leader location impacts BFT protocol performance).

The PBFT implementation completed the failover for small system sizes only and with a longer failover time than HotStuff or Kauri. We inspected the concrete behavior of the implementation and noticed that the view change was implemented in an inefficient way (with respect to utilized bandwidth): For every request that timed out, the new leader would assign a sequence number and instantly propose it in a new block (containing only a single request). This way, the costs of running the agreement protocol in our simulated wide area network would not amortize over multiple (hundreds of) requests because every timed-out request demanded to collect a quorum.

V. Related Work

The first simulator specifically designed for traditional BFT protocols is BFTSim [11]. It is tailored for small replica groups, and its limited scalability renders BFTsim impractical for newer larger-scale BFT protocols. BFTSim requires modeling BFT protocols in the P2 language, which introduces error-proneness given the complexity of protocols, like PBFT's view change mechanism and Zyzzyva's numerous corner cases. While capable of simulating faults, it only considers non-malicious behavior, lacking functionality to tackle sophisticated Byzantine attacks. BFTsim uses ns-2 for realistic networking and is resource-friendly, running on a single machine.

Recently, Wang et al. [12] introduced a BFT simulator that exhibits resource-friendliness, high scalability, and includes an "attacker module" with predefined attacks such as partitioning, adaptive, and rushing attacks. Similar to BFTSim, it requires the re-implementation of a BFT protocol (in JavaScript). Another current limitation is the inability to measure throughput.

Several simulators were developed for blockchain research, including Shadow-Bitcoin [28], the Bitcoin blockchain simulator [29], BlockSim [30], SimBlock [31], and ChainSim [32]. These simulators primarily concentrate on constructing models that accurately depict the features of Proof-of-Work (PoW) consensus mechanisms, making them less suitable for adoption in BFT protocol research.

Moreover, related work on behavior prediction encompasses stochastic modeling of BFT protocols [33] and validations of BFT protocols through unit test generation [20].

Additionally, there are tools for emulating or simulating *any* distributed applications. Emulators such as Mininet [22], [34] and Kollaps [14] create realistic networks that run actual Internet protocols and application code with time synchronized with wall clock. Both approaches offer a high level of realism but are less resource-friendly. Mininet, although not scalable, had this issue addressed with the introduction of Maxinet [35], enabling distributed emulation using multiple physical machines. Kollaps [14] is a scalable emulator but requires a significant number of physical machines for conducting large-scale experiments. Furthermore, ns-3 [21] is a resource-friendly and scalable network simulator, but it necessitates the development of an application model, thus impeding application layer realism (and preventing plug-and-play utility). Phantom [18] uses a hybrid emulation/simulation architecture: It executes real applications as native OS processes, co-opting the processes into a high-performance network and kernel simulation and thus can scale to large system sizes.

VI. Conclusion

We proposed a methodology to assess the performance of BFT protocols via network simulations. A major advantage of our method compared to related approaches (i.e., [11], [12]) is that we can plug and play existing protocol implementations without requiring an error-prone re-implementation of such a protocol in a modeling language. We found that our method can be useful to study the performance of a protocol at increasing system scale and in realistic environments. A further use case of our method is to spot implementation bugs as simulations can be used to perform integration tests of distributed systems at a large scale in an inexpensive way.

Overall, the proposed simulation-based evaluation method offers a valuable tool for researchers and practitioners working with BFT protocols in the context of DLT applications. It not only streamlines the scalability analysis process but also provides cost-effectiveness and accuracy, enabling the design and deployment of more efficient and resilient BFT-based distributed systems.

Acknowledgments

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) grant number 446811880 (BFT2Chain).

References

[1] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI*. Berkeley, CA, USA: USENIX Association, 1999, p. 173–186.

[2] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Int. Workshop on Open Problems in Network Security*. Cham: Springer, 2015, pp. 112–125.

[4] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 2019, pp. 347–356.

[5] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *IEEE Symp. on Security and Privacy*. Washington, DC, USA: IEEE Comp. Soc., 2021, pp. 466–483.

[6] D. Cason, E. Fynn, N. Milosevic, Z. Milosevic, E. Buchman, and F. Pedone, "The design, architecture and performance of the tendermint blockchain network," in *40th Int. Symp. on Reliable Distributed Systems*. Washington, DC, USA: IEEE Comp. Soc., 2021, pp. 23–33.

[7] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation," in *ACM SIGOPS 28th SOSP*. New York, NY: ACM, 2021, pp. 35–48.

[8] C. Stathakopoulou, T. David, and M. Vukolić, "Mir-bft: High-throughput BFT for blockchains," *arXiv preprint arXiv:1906.05552*, 2019.

[9] P. Li, G. Wang, X. Chen, F. Long, and W. Xu, "Gosig: a scalable and high-performance Byzantine consensus for consortium blockchains," in *11th ACM Symp. on Cloud Computing*. New York, NY: ACM, 2020, pp. 223–237.

[10] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus in the lens of blockchain," *arXiv preprint arXiv:1803.05069*, 2018.

[11] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire." in *NSDI*, vol. 8. Berkeley, CA, USA: USENIX Association, 2008, pp. 189–204.

[12] P.-L. Wang, T.-W. Chao, C.-C. Wu, and H.-C. Hsiao, "Tool: An efficient and flexible simulator for Byzantine fault-tolerant protocols," in *52th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks*. Washington, DC, USA: IEEE Comp. Soc., 2022, pp. 287–294.

[13] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *12th ACM SOSP*. New York, NY: ACM, 2005, pp. 75–90.

[14] P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: decentralized and dynamic topology emulation," in *15th European Conf. on Computer Systems*. New York, NY: ACM, 2020, pp. 1–16.

[15] C. Berger, S. Schwarz-Rüsch, A. Vogel, K. Bleeke, L. Jehl, H. P. Reiser, and R. Kapitza, "Sok: Scalability techniques for BFT consensus," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2023.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[17] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Journal of cryptology*, vol. 17, pp. 297–319, 2004.

[18] R. Jansen, J. Newsome, and R. Wails, "Co-opting linux processes for high-performance network simulation," in *USENIX ATC 22*. Berkeley, CA, USA: USENIX Association, 2022, pp. 327–350.

[19] C. Berger, S. B. Toumia, and H. P. Reiser, "Does my bft protocol implementation scale?" in *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, 2022, pp. 19–24.

[20] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: BFT Systems Made Robust," in *25th Int. Conf. on Principles of Distributed Systems*, vol. 217. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 7:1–7:29.

[21] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Cham: Springer, 2010, pp. 15–34.

[22] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *9th ACM SIGCOMM Workshop on Hot Topics in Networks*. New York, NY: ACM, 2010, pp. 1–6.

[23] J. Sousa and A. Bessani, "Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines," in *34th IEEE Symp. on Reliable Distributed Systems*. Washington, DC, USA: IEEE Comp. Soc., 2015, pp. 146–155.

[24] C. Berger, H. P. Reiser, J. Sousa, and A. N. Bessani, "AWARE: Adaptive wide-area replication for fast and resilient Byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1605–1620, 2022.

[25] C. Berger, H. P. Reiser, and A. Bessani, "Making reads in BFT state machine replication fast, linearizable, and live," in *40th Int. Symp. on Reliable Distributed Systems*. Washington, DC, USA: IEEE Comp. Soc., 2021, pp. 1–12.

[26] S. Rüsch, K. Bleeke, and R. Kapitza, "Themis: An efficient and memory-safe BFT framework in rust: Research statement," in *3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. New York, NY: ACM, 2019, pp. 9–10.

[27] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *44th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. Washington, DC, USA: IEEE Comp. Soc., 2014, pp. 355–362.

[28] A. Miller and R. Jansen, "Shadow-Bitcoin: Scalable simulation via direct execution of multi-threaded applications," in *8th Workshop on Cyber Security Experimentation and Test*. Berkeley, CA, USA: USENIX Association, 2015.

[29] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *ACM SIGSAC CCS*. New York, NY: ACM, 2016, pp. 3–16.

[30] C. Faria and M. Correia, "Blocksim: blockchain simulator," in *IEEE Int. Conf. on Blockchain*. Washington, DC, USA: IEEE Comp. Soc., 2019, pp. 439–446.

[31] Y. Aoki, K. Otsuki, T. Kaneko, R. Banno, and K. Shudo, "Simblock: A blockchain network simulator," in *IEEE Conf. on Computer Communications Workshops*. Washington, DC, USA: IEEE Comp. Soc., 2019, pp. 325–329.

[32] B. Wang, S. Chen, L. Yao, and Q. Wang, "Chainsim: A p2p blockchain simulation framework," in *CCF China Blockchain Conf.* Singapore: Springer, 2020, pp. 1–16.

[33] M. Nischwitz, M. Esche, and F. Tschorsch, "Bernoulli meets pbft: Modeling BFT protocols in the presence of dynamic failures," in *16th Conference on Computer Science and Intelligence Systems*. Washington, DC, USA: IEEE Comp. Soc., 2021, pp. 291–300.

[34] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *8th Int. Conf. on Emerging networking experiments and technologies*. New York, NY: ACM, 2012, pp. 253–264.

[35] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *IFIP Networking Conf.* Washington, DC, USA: IEEE Comp. Soc., 2014, pp. 1–9.