Quantitative Evaluation of BFT Protocols

Raluca Halalai Thomas A. Henzinger Vasu Singh
IST Austria
A-3400 Klosterneuburg, Austria
{rhalalai,tah,vasu.singh}@ist.ac.at

***Abstract*—Byzantine Fault Tolerant (BFT) protocols aim to improve the reliability of distributed systems. They enable systems to tolerate arbitrary failures in a bounded number of nodes. BFT protocols are usually proven correct for certain safety and liveness properties. However, recent studies have shown that the performance of state-of-the-art BFT protocols decreases drastically in the presence of even a single malicious node. This motivates a formal quantitative analysis of BFT protocols to investigate their performance characteristics under different scenarios. As BFT protocols are generally large-scale distributed systems, conventional model checking techniques do not scale.**

We present HyPerf, a new hybrid methodology based on model checking and simulation techniques for evaluating the performance of BFT protocols. We build a transition system corresponding to a BFT protocol and systematically explore the set of behaviors allowed by the protocol. We associate certain timing information with different operations in the protocol, like cryptographic operations and message transmission. After an elaborate state exploration, we use the time information to evaluate the performance characteristics of the protocol using simulation techniques. We integrate our framework in Mace, a tool for building and verifying distributed systems. We evaluate the performance of PBFT using our framework. We describe two different use-cases of our methodology. For the benign operation of the protocol, we use the time information as random variables to compute the probability distribution of the execution times. In the presence of faults, we estimate the worst-case performance of the protocol for various attacks that can be employed by malicious nodes. Our results show the importance of hybrid techniques in systematically analyzing the performance of large-scale systems.

I. Introduction

Distributed systems are now increasingly common and often span across multiple platforms. Malicious attacks, software errors, and unpredictable network delays in these systems require complex recovery mechanisms that allow the system to function correctly. However, it is difficult to also maintain good performance of the system under these scenarios. For instance, in July 2008, a corrupted bit caused several hours of downtime to the Amazon Simple Storage Service (Amazon S3) [1]. This motivates the need of *fault tolerant* protocols, which enable a system to continue its operation even in the presence of faults. Moreover, it is desired that these protocols provide *high throughput* even in the presence of faults.

In this paper, we focus on Byzantine faults, where a faulty node may behave arbitrarily (even in a malicious manner). The problem of Byzantine fault tolerance was first formulated by Leslie Lamport in 1982 [2]. Several Byzantine fault tolerant (BFT) protocols have been developed in this direction, often relying on state machine replication, a software technique where a service is modeled as a state machine and deployed on several replica servers that aim to execute requests from different clients in the same order.

BFT protocols have been widely studied in the systems research community [2], [3], [4], [5], [6], [7], [8]. Generally, BFT protocols are proved correct with respect to certain qualitative safety and liveness properties. For example, a common safety property specifies that every replica observes the same total order of client requests. A basic question that arises is how faults affect the throughput of BFT protocols. Indeed, Clement et al. [9], [10] empirically show that the performance of BFT protocols drastically degrades as the number of faults increases. They conclude that the current state-of-the-art BFT protocols are dangerously fragile. A single faulty node is capable of reducing the total system throughput with multiple orders of magnitude and even causing complete unavailability of the service. Such performance degradation renders BFT protocols virtually unusable. These studies motivate a formal performance analysis of BFT protocols.

Model checking is a common approach for formal analysis of systems. However, model checking requires a model of the system, which is often time-consuming to produce and error-prone. Moreover, the performance characteristics observed on the model may not accurately represent system performance. In many cases, model checking does not scale to large systems. Another common technique for evaluating performance relies on system-wide simulation, which requires access to the system, an ability to modify the individual nodes in order to evaluate performance in different scenarios, and is often time-consuming. Moreover, it is hard to guide simulation in a manner that expands the set of observed behaviors. Indeed, it is often hard to capture corner cases in protocol execution using simulation techniques. To sum up, while formal techniques like model checking allow the systematic study of different scenarios and their effects, simulation based techniques provide results close to realistic scenarios.

We are inspired by the recent hybrid techniques that have been developed to complement model checking with various techniques in order to scale to large systems. For example, concolic testing [11] mixes symbolic and concrete execution to scale to large systems. Similarly, Godefroid et al. [12] use random testing in conjunction with systematic exploration for verification of C programs.

We present HyPerf (stands for Hybrid Performance evaluation) that brings together the systematic state exploration

of model checking, and the ability of simulation techniques to capture realistic behavior, for quantitative evaluation of BFT protocols. We first present the theoretical framework behind HyPerf. Our framework basically defines a transition system corresponding to a BFT protocol. The transition system captures the notion of time associated with different operations in the protocol. We use basic model checking techniques to systematically explore the set of behaviors of the BFT protocol. We analyze the set of obtained behaviors using Monte Carlo techniques [13]. We implement our framework on top of Mace [14], a toolkit developed for building and verifying distributed systems. This gives us the benefit of evaluating the protocol as it is deployed, and eliminates the need of extracting a model from the protocol (as generally required by model-checking techniques). We model PBFT [4], a popular BFT protocol in Mace. We obtain timing information for the different operations (like message transmissions, cryptography and request executions) by microbenchmarking. We evaluate the behavior of the protocol using MaceMC [16], a model checker for exploring the possible execution paths.

We use our methodology to obtain the response time behavior of BFT protocols under different scenarios: benign case (where no node is faulty), and cases where different replicas are faulty. Our results are consistent with those obtained with a deployed instance of a BFT protocol on a cluster of nodes. The model checking backend allows us to tune the different parameters, like the number of paths to be covered. Moreover, our technique is reasonably fast and allows the programmers to analyze the performance of their protocols at design-time. Clearly, our experiments show the importance of hybrid techniques for performance evaluation of large-scale systems.

II. The HyPerf Framework

We now provide a brief overview of BFT protocols. Then, we present our framework, HyPerf, that expresses BFT protocols and their correctness and quantitative properties. We gradually develop the framework in three steps. First of all, we build a framework to reason about common safety and liveness properties. In the next step, we adapt the framework to reason about worst-case response time: we add a time represented as a natural number in every state. In the last step, we represent time in every state as an expression over random variables. This gradual buildup of framework is intended to aid the reader in understanding how different parts of the framework fit in.

A. BFT Protocols.

The problem of Byzantine fault tolerance was first formulated by Leslie Lamport in 1982 [2]. He described the difficulty of achieving consensus in an environment where nodes can behave arbitrarily. There are two basic techniques for providing Byzantine fault tolerance. The first is state machine replication: a pessimistic approach where replicas communicate with each other to agree on a total order before processing clients requests. The second is an optimistic quorum-based approach, where replicas speculatively execute requests from different clients and rely on versioning to maintain consistency.

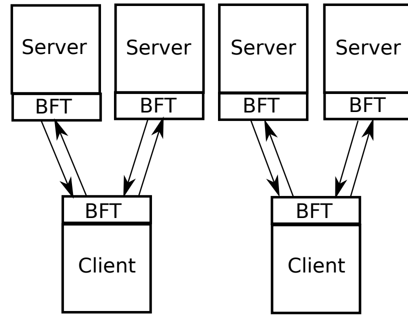


Fig. 1. Generic BFT architecture.

In this work, we focus on the more popular state machine replication approach. State machine replication models the target service as a state machine that is deployed on several servers (replicas). This technique ensures robustness to failures by replicating data, enabling the use of commodity hardware and lowering the cost for data centers. The purpose of the replication protocol is to ensure that replicas eventually agree on a total order in which to process the clients requests, even in the presence of a bounded number of faulty servers and an unbounded number of faulty clients. Figure 1 presents the architecture of a system using a BFT state machine replication protocol. The BFT protocol lies underneath the application layer.

Lamport proposed the Paxos [3] protocol for achieving consensus in the presence of faults. The algorithm ensured safety in the presence of arbitrary behavior. However, it did not ensure liveness (according to the FLP impossibility result [15], it is impossible to achieve both in asynchronous systems, if nodes can fail). The complexity of Paxos made it difficult to be accepted by the community, leading to an 8-year stall between the initial proposal of the algorithm in a technical report and its publication in 1998.

Research in BFT protocols was boosted by the publication of PBFT [4], the first protocol to ensure both safety and liveness as long as the network is not arbitrarily asynchronous (i.e., delays do not grow exponentially). After the publication of PBFT, several systems tried to improve the performance of state machine replication protocols and to make such techniques practical for real-world use [6], [7], [8], [9]. The focus of modern BFT protocols has shifted towards high performance in the absence of faults. Unfortunately, this made them susceptible to optimizations that degrade performance in the presence of Byzantine faults.

B. The Generic BFT Model

Our model captures the clients, their requests, replicas, their states, and the communication (via messages) between the clients and the replicas that forms the basis of the BFT protocol.

Clients and replicas. Let $C = C_m \cup C_c$ be the set of *clients*,

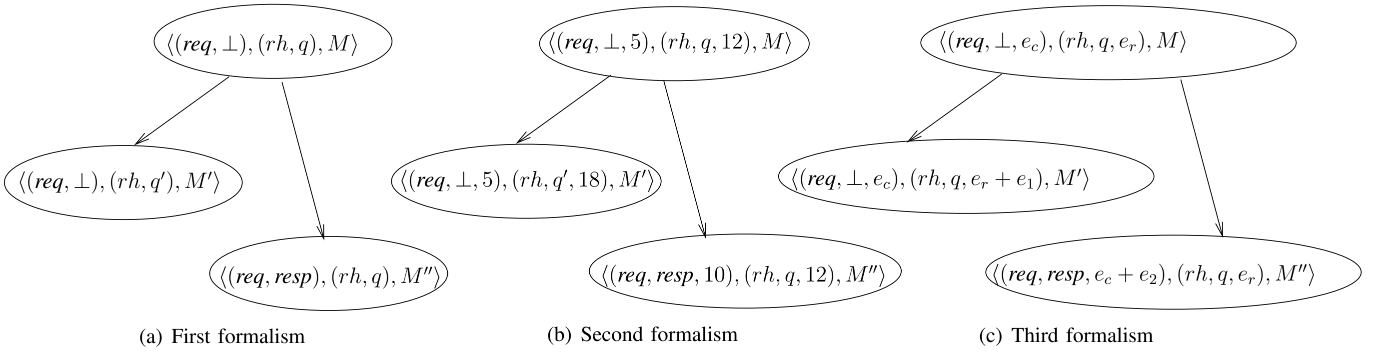


Fig. 2. Sample states in different formalisms. In the first formalism, there is no notion of time. In the second, time corresponds to the worst-case execution time. In the third, time is represented using expressions over random variables.

where C_m is the set of malicious clients and C_c is the set of correct clients. Let $R = R_m \cup R_c$ be the set of *replicas*, where R_m is the set of malicious replicas and R_c is the set of correct replicas. The total number of replicas $|R|$ generally depends on the number of faulty replicas $|R_m|$.

System states. The state of the protocol is composed of the client state, the replica state, and the messages. The state of a client is captured by the requests it issues and the responses it receives. The state of a replica consists of the sequence of requests it has processed so far and a protocol specific state. The messages capture the information of the source, the destination, the type of message, and the contents.

Formally, let Req be the *set of operations* that a client can request to replicas. Let $Resp$ be the *set of responses* that a client can compute based on replies from replicas. We use the notation $resp \sim req$ to say that a response corresponds to a client request. We assume that each client issues one request at a time; it waits for a response before sending a new request. A *client state* is given by a function $\sigma_C : C \to (Req \cup \{\bot\} \times Resp \cup \{\bot\})$. A replica state consists of two components: a sequence of executed requests and a protocol specific substate. Let Q_r be a set of protocol specific *replica substates*. Thus, a *replica state* is given by a function $\sigma_R : R \to Req^* \times Q_r$. The messages exchanged between the nodes in a BFT protocol are protocol-specific. So, in our generic model, we simply formalize a *message* m to be of the form $\langle type, src, dest, content\rangle$. *Request* and *reply* messages are the common interface used by BFT protocols to express client - server interaction. For $c \in C$ and $r \in R$, we capture a *request* message as $\langle Request, c, r, req\rangle$ with $req \in Reqs$ and corresponding *reply* messages as $\langle Reply, r, c, resp\rangle$ with $resp \in Resps$. A *message state* σ_M is a set of messages. We define $\Sigma = (\Sigma_C \times \Sigma_R \times \Sigma_M)$ as the set of possible *system states*, where Σ_C is the set of all client states, Σ_R is the set of all replica states, and Σ_M is the set of message states.

Transition system. To capture the dynamics of the protocol, we use the notion of a *transition system*. We define a transition system $ts = \langle \Sigma, \sigma_0, \delta\rangle$, where Σ is the set of system states, σ_0 is the initial state of the system and $\delta \subseteq \Sigma \times \Sigma$ is the transition relation between states. We define the initial state of the system as $\sigma_0 = (\sigma_C^0, \sigma_R^0, \sigma_M^0) \in \Sigma$. The initial state of a client is given by $\sigma_C^0(c) = (\bot, \bot)$. Likewise, the initial state of a replica is given by $\sigma_R^0(r) = (\varepsilon, q)$ for some state $q \in Q_r$ that represents the initial protocol specific substate. The initial state of the system does not include any messages, thus we define $\sigma_M^0 = \varepsilon$. Some sample states of the transition system are shown in Figure 2(a).

C. Correctness Properties

We now describe the common safety and liveness properties that a BFT protocol needs to satisfy.

- S1. *Request/response correspondence.* For all correct clients, the response, when received, corresponds to a previously issued request.
- S2. *State reachability.* For all correct replicas, each state is reachable via a sequence of client requests.
- S3. *Linearizability.* For all correct replicas, requests are executed in a sequential order, consistent with the order seen by the clients.
- L1. *Termination.* All correct clients eventually receive a response.
- L2. *State agreement.* All correct replicas eventually have the same state.

These properties can be formally expressed in our framework. For example, the safety property S1 states that $\forall c \in C_c\ \forall \sigma_C \in \Sigma_C, \sigma_C(c) = (req, resp) \cdot resp = \bot \vee resp \sim req$.

D. Augmenting the Model with Time

The generic BFT model allows one to model check BFT protocols with respect to the safety and liveness properties presented above. However, the model is agnostic to the passage of time, and thus cannot be used to evaluate the performance of BFT protocols. We augment the client and replica states with a notion of time. We now define client and replica states as follows. The state of a client is given by a function $\sigma_C : C \to (Req \cup \{\bot\} \times Resp \cup \{\bot\} \times \mathbb{N})$. Likewise, the state of a replica is given by a function $\sigma_R : R \to (Req^* \times Q_r \times \mathbb{N})$. Indeed, in addition to the earlier formalism, we capture the local time for each client and replica as part of its respective state. This is shown in Figure 2(b).

We assume that every client makes exactly one request, and the request is made at time 0. Moreover, once a client

receives a response, its timestamp does not increase. The extended model allows us to reason about performance of a BFT protocol in terms of the time it takes to respond to a given request. Formally, we have the property S4 that bounds the completion time of each request issued by a correct client.

- S4. *Worst response time.* For all correct clients, each request is completed within a protocol-specific time limit. Formally, $\forall c \in C_c, \forall \sigma_C \in \Sigma_C$, if $\sigma_C(c) = (\mathit{req}, \bot, t)$, then $t < T$.

The safety property S4 allows us to capture the worst case performance of a BFT protocol. However, in an asynchronous environment, the time taken by individual operations is generally not bounded, and a probabilistic model to study the protocol is desirable. Generally, this is achieved by turning the transition system into a Markov decision process, every each transition has an associated probability. However, reasoning about MDPs does not usually scale to large systems. Moreover, converting a transition system with a continuous notion of time to an MDP requires to discretize time.

This brings us to a hybrid approach for performance evaluation. We adapt the notion of time in every state to symbolically capture the passage of time for every client and replica in terms of individual event durations. We define a set V of random variables that denote the duration for processing different events. We define a time expression e using the grammar

$$e ::= v \mid e + e \mid \max(e, e)$$

where $v \in V$. We denote the set of time expressions as *Expr*. We define a partial order $\prec$ on the set *Expr* of expressions as given by the inference rules below.

$$\frac{}{v \prec v + v'} \qquad \frac{}{v \prec \max(v, v')}$$

$$\frac{e \prec e'}{e + e_1 \prec e' + e_1} \qquad \frac{e \prec e' \quad e_1 \prec e}{\max(e, e_1) \prec \max(e', e_1)}$$

To capture these events in the protocol, we now define a client state as $\sigma_C : C \to (\mathit{Req} \cup \{\bot\} \times \mathit{Resp} \cup \{\bot\} \times \mathit{Expr})$, and a replica state as $\sigma_R : R \to (\mathit{Req}^* \times Q_r \times \mathit{Expr})$. The idea of introducing time expressions in states allows us to capture time as expressions over random variables, that represent time required for individual events in the protocol. A sample set of states is shown in Figure 2(c). The response time of a protocol can now be studied by observing the time expressions for the states when the client receives a response for its request. Instead of a correctness property, we are now able to compute the response time.

Q1. *Response time.* Let E be the set of expressions such that $\forall c \in C_c, \forall \sigma_C \in Sigma_C$, if $\sigma_C(c) = (\mathit{req}, \mathit{resp}, e)$ such that $\mathit{resp} \sim \mathit{req}$, then $e \in E$. Note that E is a set of functions of random variables that intuitively represent the possible timing behaviors of the protocol under different conditions. In our protocol, we restrict nondeterminism to the malicious nodes, and thus different values in E represent different possible behaviors of the malicious nodes. In case there are no malicious nodes, the set E is a singleton. In case we have malicious nodes, we can attempt to choose the worst time expression, that is, the expression $e \in E$ such that for all expressions $e' \in E$ such that $e' \neq e$, we have $e' \prec e$). This would correspond to an adversarial behavior of the faulty nodes. In our case study of the PBFT protocol, we are indeed able to find such a worst time expression as explained in Section IV.

E. The PBFT Protocol

We now illustrate how BFT protocols in the literature can be specified in our framework. We focus on PBFT, a Byzantine fault tolerant protocol that re-ignited researchers' interest in the field by demonstrating that Byzantine fault tolerance in asynchronous environments does not necessarily imply impractical performance. PBFT consists of three sub-protocols: *three phase agreement* for imposing a total order among the requests, *view change* for ensuring progress and *checkpointing* for allowing replicas to re-synchronize.

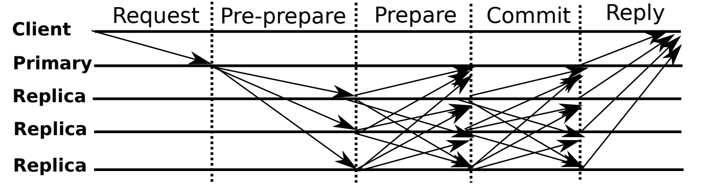


Fig. 3. PBFT: three phase agreement.

As shown in Figure 3, a client requests the execution of an operation by sending a request message to a distinguished replica, called the *primary*. The primary initiates a three-phase agreement protocol by multicasting *pre-prepare* messages to replicas. Upon the arrival of the pre-prepare message, each replica broadcasts *prepare* messages to the others. Once a replica has received $2 \cdot f$ distinct prepare messages, it enters the prepared state and broadcasts a *commit* message. A replica enters the committed state once it has received $2 \cdot f + 1$ distinct commit messages. It then executes the operation requested by the client and sends back a *reply* message containing the result. Each client waits for $f + 1$ replies from different replicas before accepting a response. The *view change* sub-protocol is initiated whenever the primary is suspected to be faulty by some other replica, preventing the replica from slowing down the algorithm arbitrarily. Once it receives a request directly from a client, each replica forwards the request to the primary and starts a timer. If the timer expires before the completion of the respective operation, the replica initiates a view change. If sufficient replicas suspect the primary of being faulty, the primary is replaced. The *checkpointing* sub-protocol is used to maintain replicas in synchrony. We decided to ignore this sub-protocol in our study.

To ensure liveness, PBFT requires at least $3 \cdot |R_m| + 1$ replicas, where $|R_m|$ is the number of malicious replicas to be tolerated. No such constraint is enforced on the number of clients.

III. THE HYPERF IMPLEMENTATION

We implement our framework on top of Mace [14], a mature toolkit for building and verifying distributed systems. In Mace, systems are modeled using a C++ language extension. The Mace compiler can automatically generate C++ code from the model. Thus, HyPerf has the interesting feature that it evaluates the performance of the real system, rather than a simplified model. Moreover, this allows the analysis and design of the protocol to go together.

Models developed in Mace are layers of reactive state transition systems. This enables model checking for safety and liveness properties. MaceMC [16] is the model checker provided within the Mace toolkit; it enables a systematic verification of Mace-built distributed systems under various simulated network conditions. MaceMC focuses on finding liveness bugs and isolating their root cause. It first executes a search to identify states from which it is likely to find liveness violations. The user can set the depth of the search and the number of different execution paths to search. Starting from the identified states, Mace then performs long random execution paths to identify these violations and their cause. Likewise, MaceMC is able to check for violations of the specified safety properties. We now present our implementation of HyPerf. First, we identify operations in the protocol that require a substantial amount of time to complete. Such operations include message transmissions, cryptography and request service. We then augment the BFT model by assigning costs to each identified operation. We use model checking as a tool to reason about various execution paths, both in benign and malicious scenarios.

A. Identifying Costly Operations

In a first step towards the evaluation of BFT protocols, we identified operations that require a substantial amount of time to finish. Since all operations are in components external to the protocol, we developed microbenchmarks to obtain sample execution times. We use this data to estimate the cost corresponding to different execution paths in the protocol.

We identified three types of performance-relevant operations within the generic BFT model:

- network operations (i.e., message transmissions),
- cryptography (e.g., using message authentication codes),
- request executions (application dependent).

The timing information for each such operation can be computed during the design phase. Thus, an implementation of the system is not required to estimate the time duration of each operation.

We use the Ping network utility to obtain sample values for message transmission times. Our approach consists of two steps. First, we estimate the average size of messages based on their content. Second, we setup multiple Pings between the nodes in a cluster. We set the payload of Ping to the estimated message size. To obtain the cost of a one-way message transmission time, we divide the round-trip samples by two.

To determine sample values for cryptography operations, we perform microbenchmarks using existing security libraries. Most BFT protocols rely on external cryptography libraries to ensure security. For instance, PBFT relies on the SFS library.

Request service times depend on the application that is built on top of the BFT protocol. Thus, we sample the target application to obtain request execution samples.

B. Modeling BFT Protocols

We model BFT protocols using the C++ language extension provided by Mace. To demonstrate our framework, we develop a case study based on PBFT [4].

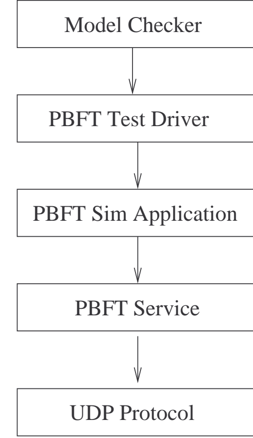


Fig. 4. PBFT layered hierarchy.

We modeled PBFT as a hierarchy of services, as shown in Figure 4. PBFT relies on the UDP transport layer, which is included in the Mace distribution. We implemented a simulated application (PBFT Sim App) and a test driver (PBFT test driver) to enable model checking within the Mace framework. Our model includes two out of the three PBFT sub-protocols: the *three phase-agreement*, which accomplishes a total ordering of client requests consistent among replicas and the *view change*, which ensures progress provided that the current primary is faulty. To keep the PBFT model accurate, we also included cryptography (Message Authentication Codes). We modeled MACs using one boolean value for each replica.

In BFT protocols, nodes are prone to malicious conditions. We expressed the Byzantine behavior for both replicas and clients, by making random choices. This enabled us to reason about the performance of PBFT under malicious conditions that are likely to occur in real environments. As shown in the code snippet below, we augmented each branch statement in the PBFT model with a random condition. We require both branches to be explored by the model checker.

```
// non-malicious
if (replies.size()>maxNumMalicious)
  collateReplies();

// malicious
val = randint(5);
if (replies.size()>maxNumMalicious || val<2)
```

```
collateReplies();
```

C. Capturing Performance

PBFT was proven to guarantee both safety and liveness in the presence of a bounded number of faults. However, in order to achieve liveness, the protocol relies on a weak synchrony assumption, i.e., the time between the moment a client sends a request for the first time and the moment it is able to compute a result is bounded. We used a modified version of the Mace model checker [17] to prevent irrelevant reordering of messages. This constraint restricted the search space to relevant execution paths. The modified model checker uses a synchronized transport, in which the network delivers messages in phases, separating each phase by a tick. The use of ticks also enabled us to avoid the simulated timers offered by Mace. In our experiments, the simulated timers proved to be inconsistent among nodes; for instance, a five-second timer may fire several times on one node before a one-second timer fires on a different node, leading to several irrelevant execution paths. Instead, we modeled PBFT's request and view change timeouts using counters that decrement during each tick of the transport.

In a first phase, we assigned fixed time values to each of the identified operations and used the Mace model checker as a tool to obtain different execution paths and their corresponding total time values (as in the second formalism). The result of this approach is total execution time of each path in the target protocol. This result is useful in order to find the execution path that leads to the worst case performance of the protocol.

In the second phase, we assigned symbolic variables to each significant operation within PBFT, rather than fixed time values (as in the third formalism). We again used the Mace model checker, but this time to obtain symbolic expressions of variables, rather than numerical values. Each symbolic variable we used has an associated statistical distribution for its duration (for instance, network delays can be modeled using the Pareto distribution [18]). This enables us to use statistical methods to reason about the probabilistic distribution of execution times. We use Matlab to compute the probability distributions for such sequences.

The behavior of BFT protocols depends on parameters such as the number of faults to be tolerated, specific timeouts, etc. We specify the number of faulty nodes in the BFT protocol. Model checking a BFT protocol needs additional parameters such as the number of execution paths to explore, the depth of the paths, etc. These parameters impact the accuracy and the speed of the model checking algorithm.

IV. Evaluation of PBFT Using HyPerf

We now use HyPerf for a quantitative evaluation of PBFT. We setup a scenario for at most one malicious server ($f = 1$). Thus, we use $3f+1 = 4$ replicas. We explored 50,000 different paths, each consisting of 10,000 steps.

We first identified the set of operations that impact the performance of the protocol. We used microbenchmarks to estimate the cost of each operation. We then used model checking to find possible sequences of events in the protocol. Based on the obtained operation costs and execution traces, we compute the expected performance of PBFT under various conditions. We use HyPerf to predict the performance of PBFT under different scenarios. In the absence of failures, our predicted results are close to the actual performance of PBFT (Figure 8). We compared our estimated values with those we obtained for a real PBFT deployment in a cluster. We used the same environment for microbenchmarking the cost of each operation, as well as for deploying the real implementation of PBFT. Our prediction matches the real measurements in both scale (the median values are close) and in shape (the distributions are also similar).

A. Microbenchmarking

As described in Section III, we first compte the time required by the time-consuming operations in PBFT: message transmissions, cryptography operations and request service times. We obtained samples for message transmission costs by using the Ping utility. There are several types of messages within the PBFT protocol: Request, Pre-prepare, Prepare, Commit and Reply. Based on the C++ headers in the PBFT source code, we estimate an average message size of 150 bytes. We set the Ping payload to to the average size of PBFT messages (150 bytes). Figure 5(a) shows the distribution of message transmission times that we obtained by performing Pings between the nodes of our private cluster.

Figure 5(b) presents the distribution of execution times for cryptography operations. PBFT uses the SFS cryptography library to implement security. We know the size of the messages that are authenticated from the description of the PBFT protocol. Thus we were able to perform microbenchmarking to obtain sample time values for the cryptography operations.

We show the distribution of request service times in Figure 5(c). Requests are specific to the application that runs above the BFT layer. For PBFT, we first developed a dummy application that reads from a remote file on each request. We then wrote an application-specific test driver and obtained the time samples for the request.

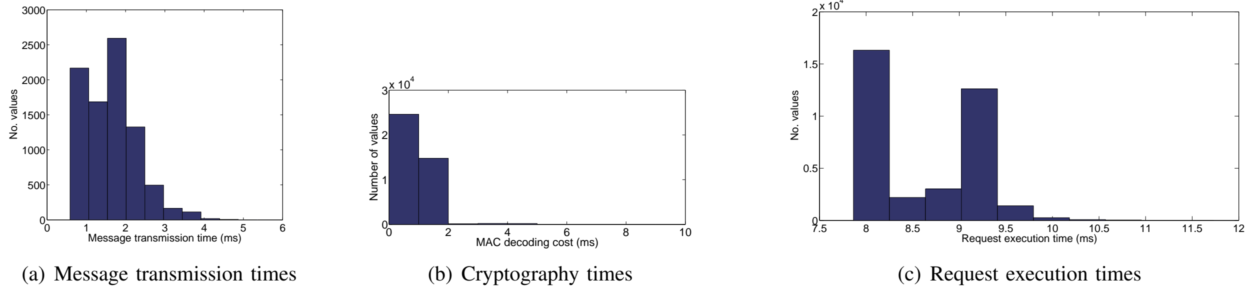
B. Evaluation

We now present the results we obtained in our evaluation of PBFT. We show how our framework can be used to estimate the performance of PBFT under benign and malicious conditions. In a first step, we reason about the expected behavior of the protocol under benign conditions. We compare our results to the measurements we perform on the real PBFT implementation. We then estimate the performance of PBFT under malicious conditions (for example, one faulty replica).

Benign case. In this scenario, we ran the model checker without any malicious nodes, obtaining a single time expression.

$$v_{req}+v_{mac}+v_{prep}+\max(v_{pre},v_{pre})+\max(v_{com},v_{com},v_{com})+ v_{wr}+\max(v_{rep},v_{rep})$$

This sequence corresponds to the "common case" of PBFT (see Figure 3). The v_{req} event corresponds to the client sending



(a) Message transmission times (b) Cryptography times (c) Request execution times

Fig. 5. Microbenchmark results for costly PBFT operations

a request to the primary. Upon the arrival of a request, the primary checks its Message Authentication Code (v_{mac}) and broadcasts a Pre-prepare message (v_{prep}) to other replicas. Each replica broadcasts a Prepare message (v_{prep}) as soon as it receives the Pre-prepare. Each replica then waits for $2f$ different Prepare messages. The waiting time corresponds to the maximum cost among Prepare messages: $\max(v_{pre}, v_{pre})$. Once the waiting time is over, each replica broadcasts a Commit message. Likewise, each replica waits for $2f + 1$ Commits to arrive, for a time equal to $\max(v_{com}, v_{com}, v_{com})$. Then, replicas execute the operation, for example v_{wr}, and send replies to the client. The client is able to compute a response once it has received $f+1$ agreeing replies. The time to wait for these replies to arrive is $\max(v_{rep}, v_{rep})$.

We computed the expected performance distribution of PBFT using the data obtained from our microbenchmarks. We compared the results with the actual measured performance of PBFT. Figure 8(a) shows that the prediction is close to the actual values in both scale and shape. We show both the raw distributions, as well as a box plot comparison. The box plot shows the median values (horizontal red line), 25 and 75 percentiles (solid box), 5 and 95 percentiles (dashed lines) and outliers (red points).

Surprisingly, the prediction is slightly more pessimistic than the real case. We expected our prediction to be slightly more optimistic, since we only modeled the execution time of a subset of the total PBFT code. Possible explanations for this result are OS optimizations that affect the real code (e.g., caching, buffering) or the accuracy of the timing measurements we used in the microbenchmarks (measuring very small events is more inaccurate than measuring the total request time).

Malicious case. There are two possible malicious scenarios: faulty primary or faulty backup (i.e., non-primary replica). When the primary exhibited malicious behavior, the model checker produced 42 possible time expressions. We parse the time expressions according to the precedence relation in our framework. We get a worst possible event sequence as follows.

$$v_{req}+v_{mac}+v_{rep}+\max(v_{pre}, v_{pre}, v_{pre})+v_{vc}+v_{mac}+v_{prep}+ \\ \max(v_{pre}, v_{pre}, v_{pre}) + \max(v_{com}, v_{com}, v_{com})+ \\ v_{wr} + \max(v_{rep}, v_{rep}, v_{rep})$$

In this scenario, the malicious primary does not send the Pre-prepare message to all replicas, thus inhibiting the execution of the request. The replicas detect the faulty primary (a view change timer, denoted by v_{vc}, expires) and initiate the view change protocol. The primary is changed and the protocol can then resume normal execution. We used this path to compute the distribution of execution times, as shown in Figure 6. The total execution times are significantly larger than in the common case, averaging around 5 seconds. This is due to the view change timer, which is set to 5 seconds, as for the default PBFT deployment.

For the scenario with one malicious replica (not the primary), the model checker produced 3 possible sequence of events. We identified the worst by taking into account the total execution times.

$$v_{req}+v_{mac}+v_{prep}+\max(v_{pre}, v_{pre})+\max(v_{com}, v_{com}, v_{com})+ \\ v_{wr} + \max(v_{rep}, v_{rep})$$

We present the distribution of execution times in the presence of one malicious replica in Figure 7. It can be seen that a malicious replica does not have a large impact on the performance of PBFT. The total execution times are close to the "common case".

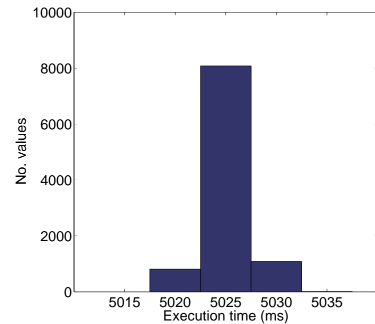
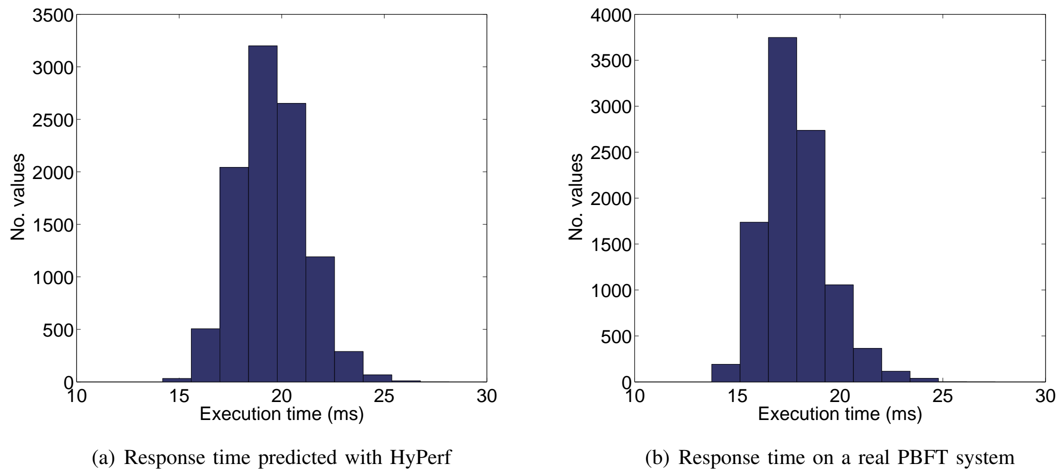


Fig. 6. Predicted execution times for a scenario with 1 malicious replica (primary).

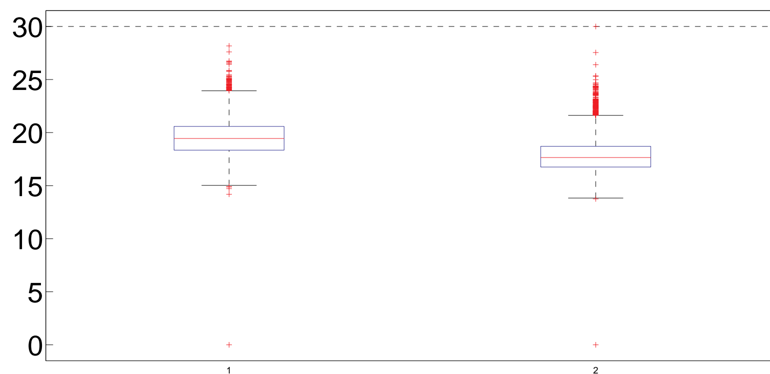
V. Related Work

A number of modeling techniques for analyzing performance of distributed systems have been proposed in the literature. Petri nets [19] and their extensions, timed petri nets [20]



(a) Response time predicted with HyPerf

(b) Response time on a real PBFT system



(c) HyPerf(left) vs. Real PBFT system(right)

Fig. 8. Comparison with real data

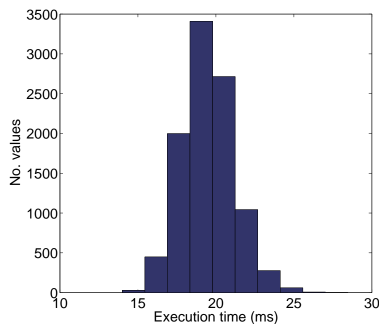


Fig. 7. Predicted execution times for a scenario with 1 malicious replica (non-primary).

and stochastic petri nets [21] have been applied to analyze various properties of distributed systems. Process calculi have also been used in performance evaluation, most notably as PEPA [22], [23]. Tools like PRISM [24], [25] have been developed and extensively used to verify probabilistic systems. Specifically, PRISM has been used to verify randomized Byzantine agreement [26]. All these verification techniques rely on extracting a model from the real system. This often requires certain assumptions, needs manual intervention, and can be error-prone. Moreover, these techniques are geared towards rigorous analysis of small systems, and thus are unable to scale to large distributed systems.

For these reasons, the behavior of BFT protocols (and in general large-scale distributed systems) is usually analyzed by simulation. Models of BFT protocols have been developed and deployed in simulation environments that emulate various network conditions. Also, real BFT implementations have been injected with faults while running on machine clusters. Several authors use model checking as a tool to reason about their BFT protocols. Singh et al. [5] built BFTSim, a simulation environment that allows evaluating BFT protocols under various network conditions. Protocols are modeled using a high level declarative language and then released in a simulated environment that allows the emulation of various network conditions such as message loss, message delays, etc. BFTSim is focused on analyzing the network characteristics for the protocol. We, on the other hand, focus on exploring various

execution paths through the protocol itself and also assess the impact of malicious nodes on performance.

Clement et al. [9] perform an empirical analysis of the state-of-the-art BFT protocols under various scenarios. They discover that all protocols are severely impacted by malicious nodes; for every studied protocol the authors produce at least one scenario that causes the performance to drop to zero. The authors propose a new approach towards building BFT protocols, rather than an approach that discovers such faults automatically. Our work complements that of Clement et al. by proposing a framework that can help BFT developers evaluate the expected performance of their protocol before they actually implement it.

Killian et al. extend their Mace model checker [16] by adding a technique to automatically find abnormally long executions. The detection algorithm works in two phases: a training phase, in which the algorithm explores different executions of the target protocol in order to find the "average" execution length, and an exploration phase, in which the algorithm searches for executions that are abnormally long. The tool also provides anomaly analysis that compares abnormal executions with average executions, trying to pinpoint the divergence point of the protocol execution and help the developer locate the bug. Our framework differs in two significant ways: first, we also model malicious nodes in the protocol and asses their impact on performance; second, our framework produces a statistical distribution of the protocol execution times, helping the developer estimate the expected performance of their algorithm, potentially filtering out executions that are very long, but have an extremely low probability of occurring (e.g., a bit corruption bypassing the checksum).

Guerraoui et al. [27] provide basic building blocks for developing BFT protocols, thus reducing the developer effort for both verification and implementation. They showed how state-of-the-art and new BFT protocols can be designed using their abstraction. They used model checking to prove the formal correctness of their proposed abstraction.

We believe that our work fits well with these existing tools and approaches. Formal techniques like model checking usually target correctness properties in small-scale systems. The obvious benefit of these is their systematic approach. On the other side of the spectrum, simulation techniques provide in-depth understanding of the actual large-scale systems. Our goal with HyPerf is to develop techniques that bring together the advantages of these approaches. We exploit formal techniques in capturing possible scenarios and reducing system behavior to mathematical expressions over individual events. We use simulation to capture the behavior of these events and compute performance characteristics.

VI. Conclusion

We presented HyPerf, an approach that combines systematic state exploration and simulation techniques to analyze the performance of distributed systems. In particular, we developed a case study for PBFT, a practical BFT state machine replication protocol, and analyzed its response time. We showed that HyPerf allows us to reason about real systems at design time, making programmers aware of the weaknesses of their protocol before wide-scale deployment. We compared the predictions obtained with HyPerf to the real performance of PBFT and found that our approach can successfully estimate both the scale and the statistical distribution of execution times.

As future work, we plan to extend the methodology presented in this paper to general distributed systems for analyzing performance properties.

References

[1] "Amazon s3 availability event: July 20, 2008," http://status.aws.amazon.com/s3-20080720.html.

[2] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, 1982.

[3] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, 1998.

[4] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the third symposium on Operating systems design and implementation*, 1999.

[5] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[6] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[8] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[9] A. Clement, E. 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*, 2009.

[10] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin, "BFT: the time is now," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008.

[11] K. Sen, "Concolic testing," in *ASE*, 2007, pp. 571–572.

[12] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[13] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 2004.

[14] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: Language support for building distributed systems," *SIGPLAN Not.*, vol. 42, 2007.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," in *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, 1983.

[16] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, 2007.

[17] C. Killian, "Programming assignment 3: Debugging a solution to the byzantine generals problem," http://www.cs.purdue.edu/homes/ckillian/courses/sp10/cs505, 2010.

[18] W. Zhang and J. He, "Modeling end-to-end delay using pareto distribution," in *Proceedings of the Second International Conference on Internet Monitoring and Protection*, 2007.

[19] J. L. Peterson, "Petri nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, 1977.

[20] B. Berthomieu and M. Diaz, "Modeling and verification of time dependent systems using timed Petri nets," *IEEE Transactions on Software Engineering*, vol. 17, pp. 259–273, 1991.

[21] M. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Transactions on Computers*, vol. 31, pp. 913–917, 1982.

[22] J. Hillston, *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[23] S. Gilmore and J. Hillston, "The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling," in *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, ser. Lecture Notes in Computer Science, no. 794. Springer-Verlag, May 1994, pp. 353–368.

[24] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *Computer Performance Evaluation / TOOLS*, 2002, pp. 200–204.

[25] M. Z. Kwiatkowska, G. Norman, and R. Segala, "Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM," in *CAV*, 2001, pp. 194–206.

[26] M. Kwiatkowska and G. Norman, "Verifying randomized Byzantine agreement," in *Proc. Formal Techniques for Networked and Distributed Systems (FORTE'02)*, ser. LNCS, vol. 2529. Springer, 2002, pp. 194–209.

[27] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proceedings of the 5th European conference on Computer systems*, 2010.