# RBFT: Redundant Byzantine Fault Tolerance

Pierre-Louis Aublin
Grenoble University

Sonia Ben Mokhtar
CNRS - LIRIS

Vivien Quéma
Grenoble INP

*Abstract*—**Byzantine Fault Tolerant state machine replication (BFT) protocols are replication protocols that tolerate arbitrary faults of a fraction of the replicas. Although significant efforts have been recently made, existing BFT protocols do not provide acceptable performance when faults occur. As we show in this paper, this comes from the fact that all existing BFT protocols targeting high throughput use a special replica, called the primary, which indicates to other replicas the order in which requests should be processed. This primary can be *smartly* malicious and degrade the performance of the system without being detected by correct replicas. In this paper, we propose a new approach, called RBFT for Redundant-BFT: we execute multiple instances of the same BFT protocol, each with a primary replica executing on a different machine. All the instances order the requests, but only the requests ordered by one of the instances, called the master instance, are actually executed. The performance of the different instances is closely monitored, in order to check that the master instance provides adequate performance. If that is not the case, the primary replica of the master instance is considered malicious and replaced. We implemented RBFT and compared its performance to that of other existing robust protocols. Our evaluation shows that RBFT achieves similar performance as the most robust protocols when there is no failure and that, under faults, its maximum performance degradation is about 3%, whereas it is at least equal to 78% for existing protocols.**

## I. Introduction

Byzantine Fault Tolerant (BFT) state machine replication is an efficient and effective approach to deal with arbitrary software and hardware faults [1], [5], [7], [8], [18]. The wide range of research carried out in the field of BFT in the last decade primarily focused on building *fast* BFT protocols, i.e., protocols that are designed to provide the best possible performance in the common case (i.e. in the absence of faults) [4], [9], [11], [12]. More recently, interest has been given to *robustness*, i.e., building BFT protocols that achieve good performance when faults occur. Three protocols have been proposed to address this issue that are, Prime [2], Aardvark [6], and Spinning [17]. Unfortunately, as shown in Table I (details are provided in Section III), these protocols are not effectively robust: the maximum performance degradation they can suffer when some faults occur is at least 78%, which is not acceptable.

| | Prime | Aardvark | Spinning |
|---|---|---|---|
| Maximum throughput *degradation* | 78% | 87% | 99% |

TABLE I: Performance degradation of “robust” BFT protocols under attack.

The reason why the above mentioned BFT protocols are not robust is that they rely on a dedicated replica, called *primary*, to order requests. Even if there exists several mechanisms to detect and recover from a malicious primary, the primary can be *smartly* malicious. Despite efforts from other replicas to control that it behaves correctly, it can slow the performance down to the detection threshold, without being caught. To design a really robust BFT protocol, a legitimate idea that comes to mind is to avoid using a primary. One such protocol has been proposed by Boran and Schiper [3]. This protocol has a theoretical interest, but it has no practical interest. Indeed, the price to pay to avoid using a primary is that, before ordering every request, replicas need to be sure that they received a message from all other correct replicas. As replicas do not know which replicas are correct, they need to wait for a timeout (that is increased if it is not long enough). This yields very poor performance and this explains why this protocol has never been implemented. A number of other protocols have been devised to enforce intrusion tolerance (e.g., [16]). These protocols rely on what is called *proactive recovery*, in which nodes are periodically rejuvenated (e.g., their cryptographic keys are changed and/or a clean version of their operating system is loaded). If performed sufficiently often, node rejuvenation makes it difficult for an attacker to corrupt enough nodes to harm the system. These solutions are complementary to the robustness mechanisms studied in this paper.

In this paper, we propose RBFT (*Redundant Byzantine Fault Tolerance*), a new approach to designing robust BFT protocols. In RBFT, multiple instances of a BFT protocol are executed in parallel. Each instance has a primary replica. The various primary replicas are all executed on different machines. While all protocol instances order requests, only one instance (called the *master instance*) effectively executes them. Other instances (called *backup instances*) order requests in order to compare the throughput they achieve to that achieved by the master instance. If the master instance is slower, the primary of the master instance is considered malicious and the replicas elect a new primary, at each protocol instance. Note that RBFT is intended for open loop systems (such as e.g., Zookeeper [10] or Boxwood [14] asynchronous API), i.e., systems where a client may send multiple requests in parallel without waiting the reception of replies of anterior requests. Indeed, in a closed loop system, the rate of incoming requests would be conditioned by the rate of the master instance. Said differently, backup instances would never be faster than the master instance. RBFT further implements a fairness mechanism between clients by monitoring the latency of requests, which assures that client requests are fairly processed.

We implemented RBFT and compared its performance to that achieved by Prime, Aardvark, and Spinning. Our evaluation on a cluster of machines shows that RBFT achieves comparable performance in the fault-free case to the most robust protocols, and that it only suffers a 3% performance degradation under failures.

The rest of the paper is organized as follows. We first present the system model in Section II. We then present an analysis of state-of-the-art robust BFT protocols in Section III. In Section IV we present the design and principles of Redundant Byzantine Fault Tolerance, and we present in Section V an instantiation of it: the RBFT protocol. In Section VI we present a theoretical analysis of RBFT. In Section VII we present our experimental evaluation of RBFT. Finally, we conclude the paper in Section VIII.

## II. System model

The system is composed of $N$ nodes. We assume the Byzantine failure model, in which any finite number of faulty clients can behave arbitrarily and at most $f = \lfloor \frac{N-1}{3} \rfloor$ nodes are faulty, which is the theoretical lower bound [13]. We consider the physical machine as the smallest faulty-component: if a single process is compromised, then we consider that the whole machine is compromised. Faulty nodes and clients can collude to compromise the replicated service. Nevertheless, they cannot break cryptographic techniques (e.g., signatures, message authentication codes (MACs), collision-resistant hashing). Furthermore, we assume an asynchronous network where *synchronous intervals*, during which messages are delivered within an unknown bounded delay, occur infinitely often. Finally, we denote a message $m$ signed by node $i$'s public key by $\langle m \rangle_{\sigma_i}$, a message $m$ authenticated by a node $i$ with a MAC for a node $j$ by $\langle m \rangle_{\mu_{i,j}}$, and a message $m$ authenticated by a node $i$ with a MAC authenticator, i.e., an array containing one MAC per node, by $\langle m \rangle_{\vec{\mu}_i}$. Our system model is in line with the assumptions of other papers in the field, e.g., [4].

We address in this paper the problem of robust Byzantine Fault Tolerant state machine replication in open-loop systems, i.e., systems in which clients do not need to wait for the reply of a request before sending new requests [15]. In an open loop system, even if the malicious primary of the master instance delays requests, correct primaries of the backup instances will still be able to order new requests coming from clients and thus to detect the misbehaving primary. This is not the case in closed-loop systems. We will consider the robustness of BFT protocols intended for closed-loop systems in our future work.

## III. Analysis of existing robust BFT protocols

We present in this section an analysis of existing robust BFT protocols i.e., Prime [2], Spinning [17] and Aardvark [6]. These three protocols are the only protocols designed to target the robustness problem. Indeed, an analysis of other famous BFT protocols, e.g., PBFT [4], QU [1], HQ [7] and Zyzzyva [12], performed in [6], has shown that these protocols suffer from a robustness issue. Specifically, although they are build to eventually recover from attacks, the throughput of all of them drops to zero during a possibly long time interval corresponding to the duration of the attack, which is not acceptable for their clients. In all these protocols, the system is composed of $N = 3f+1$ replicas, among which one has the role for proposing sequence number to requests, i.e., the primary. Moreover, in order to compute the theoretical performance loss of the various protocols, we examine the protocol actions performed by the different nodes to process a batch of requests of size $b$. We consider the computational costs of the cryptographic operations. Specifically, we consider that verifying and generating a signature costs $\theta$ cycles, verifying and generating a MAC authenticator costs $\alpha$ cycles and generating the digest of a message costs $\delta$ cycles.

### A. Prime

In Prime [2], clients send their requests to any replica in the system. Replicas periodically exchange the requests they receive from clients. As such, they are aware of current requests to order and start expecting ordering messages from the primary which should contain them. Furthermore, whether there are requests to order or not, the primary must periodically send (possibly empty) ordering messages, every $\Delta_{pp}$ cycles. This allows non-primary replicas to expect ordering messages with a given frequency. In order to improve the accuracy of the expected frequency at which a primary should send messages, replicas monitor the network performance. Specifically, replicas periodically measure the round-trip time between each pair of them. This measure allows them to compute the maximum delay that should separate the sending of two ordering messages performed by a correct primary. This delay is computed as a function of three parameters: the round-trip time between replicas, noted $rtt$, the time needed to execute a batch of requests, noted $E$, and a constant that accounts for the variability of the network latency, noted $K_{lat}$ which is set by the developer. If the primary becomes slower than what is expected by the replicas, then it is replaced.

The Prime protocol is not robust for the following reason. If the monitoring is inaccurate, the delay expected for a primary to send ordering messages can be too long, which gives the opportunity for a malicious primary to delay ordering messages. We performed the following experiment (in order to increase the round-trip time): a malicious primary colludes with a single faulty client. The latter sends a request that is heavier to process than other requests (1ms vs 0.1ms in our experiments). This increases the monitored round-trip time, and this gives the opportunity for the malicious primary to delay requests issued by correct clients. Figure 1 presents the throughput under attack relative to the throughput in the fault-free case, in percentage, as a function of the requests size, for both a static and a dynamic load (details on the two workloads are given in Section VII). The theoretical performance loss has been computed using the Prime formulas of Table II. We observe that the primary is able to degrade the system throughput down to 22% of the performance in the fault-free case. In other words, the throughput under attack drops by up

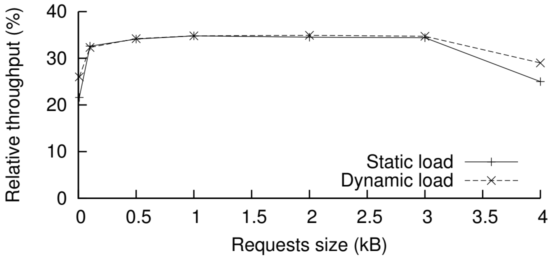


Fig. 1: Prime throughput under attack relative to the throughput in the fault-free case.

to 78% (the theoretical value being 73.2%).

| Fault free case | Under attack |
|---|---|
| $C_{FF} = \frac{\theta(2b+11f+5)+\delta(6f+1)+\Delta_{pp}}{b}$ | $C_{FF} + \frac{(rtt+E)*K_{lat}}{b}$ |

TABLE II: Number of cycles required by Prime to process one request in the fault-free case and under attack.

### B. Aardvark

Aardvark [6] is a BFT protocol based on PBFT [4], the practical BFT protocol presented by Castro and Liskov. An important principle in the robustness of Aardvark is the presence of regular changes of the primary replica. Each time the primary is changed, a new configuration, called *view*, is started. The authors of Aardvark argue that regularly changing the primary allows limiting the throughput degradation a malicious primary may cause. This regular primary changes are performed as follows. A primary replica is required to achieve at the beginning of a view a throughput at least equal to 90% of the maximum throughput achieved by the primary replicas of the last $N$ views (where $N$ is the number of replicas). After an initial grace period of 5 seconds where the required throughput is stable, the non-primary replicas periodically raise this required throughput by a factor of 0.01, until the primary replica fails to provide it. At that point, a primary change occurs and a new replica becomes the primary. In addition to expecting a minimal throughput from the primary, the replicas monitor the frequency at which the primary sends ordering messages. Specifically, replicas start a timer, called heartbeat timer, after the reception of each ordering message from the primary. If this timer expires before the primary sends another ordering message, a primary change is voted by replicas. In addition to regular view changes and heartbeat timers, Aardvark implements a number of robustness mechanisms to deal with malicious clients and replicas. For instance, it uses separate Network Interface Controllers (NICs) for clients and replicas. This avoids client traffic to slow down the replica-to-replica communication. Further, this enables the isolation of replicas that would flood the network with unfaithful messages.

As long as the system is saturated, the amount of damage a faulty primary can do on the system is limited, as the throughput expected by replicas is close to the maximal throughput clients can sustain. We performed an experiment in which the primary tries to delay requests as much as it can, under a static load. Results, depicted in Figure 2, show that the throughput provided by the system under attack is at least 76% of the throughput observed in the fault-free case. When the load is dynamic, however, the performance degradation can potentially be much higher. Indeed, when the load is low, the expected throughput computed by replicas is also low. If the load suddenly increases, a malicious primary can benefit from the low expectations computed by replicas to delay requests. More precisely, the malicious primary measures the observed throughput $obs$, computes the next expected throughput $exp$ and delays the requests by $A_{delay}(obs, exp)$ such that it provides this expected throughput. We performed different experiments under the dynamic load described in Section VII, for different message sizes. Results, depicted in Figure 2, show that because of a malicious primary under a dynamic load, the throughput of the system can drop down to 13% of the throughput that would have been provided if the primary would have been detected (the maximum throughput degradation being 87%), and up to 85.9% using the theoretical formulas presented in Table III.

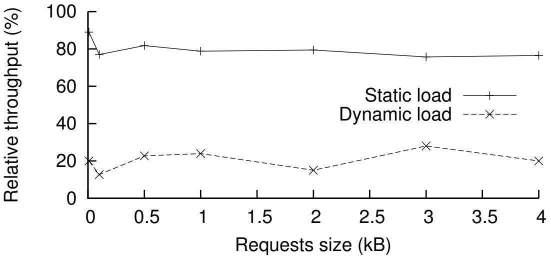


Fig. 2: Aardvark throughput under attack relative to the throughput in the fault-free case.

| Fault free case | Under attack |
|---|---|
| $C_{FF} = max(\theta + \alpha + \delta, \frac{\alpha(10f+b)+\delta}{b})$ | $C_{FF} + A_{delay}(obs, exp)$ |

TABLE III: Number of cycles required by Aardvark to process one request in the fault-free case and under attack. The delay $A_{delay}$ depends on the observed throughput $obs$ and on the expected throughput $exp$.

### C. Spinning

Spinning [17] is a BFT protocol also based on PBFT [4]. Similarly to Aardvark, Spinning performs regular primary changes. The particularity of Spinning is that these primary changes are automatically performed after the primary has ordered a single batch of requests. In this protocol, requests are sent by clients to all replicas. As soon as a non-primary replica receives a request, it starts a timer and waits for a request ordering message from the primary containing this request. If the timer expires, after a duration $S_{timeout}$, then the current primary is blacklisted (i.e., it will no longer become a primary in the future[1]), another replica becomes the primary,

[1] If $f$ replicas are already blacklisted, then the oldest one is removed from the blacklist, to ensure the liveness of the system.

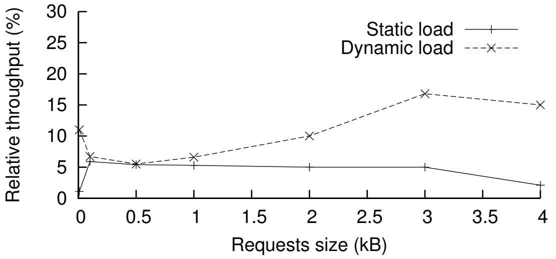


Fig. 3: Spinning throughput under attack relative to the throughput in the fault-free case.

and $S_{timeout}$ is doubled. As soon as a request has been successfully ordered, the primary is automatically changed (i.e., there is no message exchange between replicas) and the value of $S_{timeout}$ is reset to its initial value. Note that the value of $S_{timeout}$ is a system parameter statically defined; it does not depend on live monitoring of the system.

The Spinning protocol is not robust for the following reason. A malicious primary can delay the request ordering messages by a little less than $S_{timeout}$. It will drastically reduce the throughput, without being detected. As a result, it can continue to delay future ordering messages, the next time it becomes the primary. We ran several experiments where the malicious primary was delaying the sending of the ordering messages by 40ms (which is the value used by the authors of Spinning in [17]), for both under a static and a dynamic workload. Results, depicted in Figure 3, show that the throughput drops dramatically down to 1% and 4.5% of the fault-free throughput, under the static and dynamic workloads, respectively. This throughput degradation of up to 99% is clearly not acceptable. These results are close to the theoretical result of 96.9%, computed from the formulas of Table IV.

| Fault free case | Under attack |
|---|---|
| $C_{FF} = \frac{\alpha(2b+13f)+\delta(2b+1)}{b}$ | $\frac{(n-f)*C_{FF}+f*(C_{FF}+S_{timeout})}{n}$ |

TABLE IV: Number of cycles required by Spinning to process one request in the fault-free case and under attack.

### D. Summary

In this section we have seen that so-called robust BFT protocols, i.e., Prime, Aardvark and Spinning, are not effectively robust. A primary node can be *smartly* malicious and cause huge performance degradation without being caught. Prime is robust as long as the network meets a certain level of synchrony. If the variance of the network is too high, then a malicious primary can heavily impact the performance of the system. Aardvark is robust as long as the load is static. If Aardvark is fed up with a dynamic load (for instance, a load corresponding to connections to a website, which may contain many spikes), then it is not guaranteeing good performance anymore. Spinning is robust only $2f+1$ requests over $3f+1$, when the current primary is a correct replica. However, Spinning has an important weakness every time a malicious primary is in place: the malicious primary can delay the sending of the ordering messages up to the maximal allowed time.

## IV. The RBFT protocol

We have seen in the previous section that existing BFT protocols that claim to be robust are not actually robust. In this section, we present RBFT, a new BFT protocol stems from the concepts of Redundant Byzantine Fault Tolerance: multiples instances of a BFT protocol are executed simultaneously, each one with a primary replica running on a different machine. We start by an overview of RBFT. We then detail the various steps followed by replicas. Finally, we detail two key mechanisms that are used in RBFT: the monitoring mechanism and the protocol instance change mechanism.

### A. Protocol overview

As for the other robust BFT protocols, RBFT requires $3f+1$ nodes (i.e., $3f+1$ physical machines). Each node runs $f+1$ protocol instances of a BFT protocol in parallel (see Figure 4). As we theoretically show in Section VI, $f+1$ protocol instances is necessary and sufficient to detect a faulty primary and ensure the robustness of the protocol. This means that each of the $N$ nodes in the system runs locally one replica for each protocol instance. Note that the different instances order the requests following a 3-phase commit protocol similar to PBFT [4]. Primary replicas of the various instances are placed on nodes in such a way that, at any time, there is at most one primary replica per node. One of the $f+1$ protocol instances is called the *master instance*, while the others are called the *backup instances*. All instances order client requests, but only the requests ordered by the master instance are executed by the nodes. Backup instances only order requests in order to be able to monitor the master instance. For that purpose, each node runs a monitoring module that computes the throughput of the $f+1$ protocol instances. If $2f+1$ nodes observe that the ratio between the performance of the master instance and the best backup instance is lower than a given threshold, then the primary of the master instance is considered to be malicious, and a new one is elected. Intuitively, this means that a majority of correct nodes agree on the fact that a protocol instance change is necessary (the full correctness proof of RBFT can be found in Section VI). An alternative could be to change the master instance to the instance which provides the highest throughput. This would require a mechanism to synchronize the state of the different instances when switching, similar to the switching mechanism of Abstract [9]. We will explore this design in our future work.

In RBFT, the high level goal is the same as for the other robust BFT protocols we have studied previously: replicas monitor the throughput of the primary and trigger the recovery mechanism when the primary is slow. The approach we use is radically different. It is not possible for replicas to guess what the throughput of a *non*-malicious primary would be. Therefore, the key idea is to leverage multicore architectures to run multiple instances of the same protocol in parallel. Nodes have to compare the throughput achieved by the different instances to know whether a protocol instance change is required or not. We are confident to say (given the theoretical and experimental analysis) that this approach allows us to build an effectively robust BFT protocol.

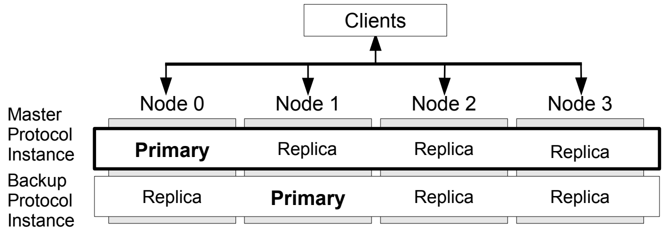


Fig. 4: RBFT overview ($f = 1$).

For RBFT to correctly work it is important that the $f + 1$ instances receive the same client requests. To that purpose when a node receives a client request, it does not give it directly to the $f + 1$ replicas it is hosting. Rather, it forwards the request to all other nodes. When a node has received $f+1$ copies of a client request (possibly including its own copy), it knows that every correct node will eventually receive the request (because the request has been sent to at least one correct node). Consequently, it gives the request to the $f + 1$ replicas it hosts.

Finally, note that each protocol instance implements a full-fledged BFT protocol, very similar to the Aardvark protocol described in the previous section. There is nevertheless a significant difference: a protocol instance does not proceed to a view change by its own. Indeed, the view changes in RBFT are controlled by the monitoring mechanism and apply on every protocol instance at the same time.

### B. Detailed protocol steps

RBFT protocol steps are described hereafter and depicted in Figure 5. The numbering of the steps is the same as the one used in the figure.

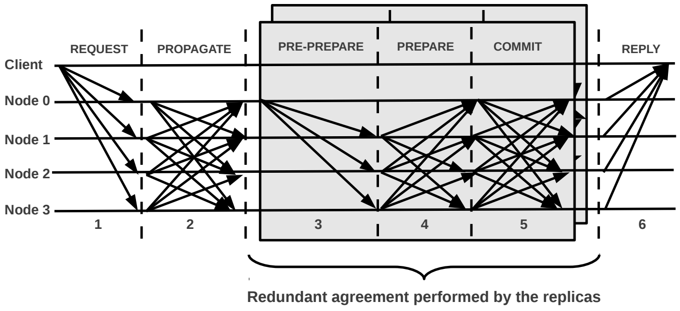


Fig. 5: RBFT protocol steps ($f = 1$).

**1. The client sends a request to all the nodes.** A client $c$ sends a REQUEST message $\langle\langle \text{REQUEST}, o, rid, c\rangle_{\sigma_c}, c\rangle_{\vec{\mu}_c}$ to all the nodes (Step 1 in the figure). This message contains the requested operation $o$, a request identifier $rid$, and the client id $c$. It is signed with $c$'s private key, and then authenticated with a MAC authenticator for all nodes. On reception of a REQUEST message, a node $i$ verifies the MAC authenticator. If the MAC is valid, it verifies the signature of the request. If the signature is invalid, then the client is blacklisted: further requests will not be processed[2]. If the request has already been executed, $i$ resends the reply to the client. Otherwise, it moves to the following step. We use signatures as the request needs to be forwarded by nodes to each other and using a MAC authenticator alone would fail in guaranteeing non-repudiation, as detailed in [6]. Similarly, using signatures alone would open the possibility for malicious clients to overload nodes by sending unfaithful requests with wrong signatures, that are more costly to verify than MACs.

**2. The correct nodes propagate the request to all the nodes.** Once the request has been verified, the node sends a $\langle \text{PROPAGATE}, \langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, i\rangle_{\vec{\mu}_i}$ message to all nodes. This step ensures that every correct node will eventually receive the request as long as the request has been sent to at least one correct node. On reception of a PROPAGATE message coming from node $j$, node $i$ first verifies the MAC authenticator. If the MAC is valid, and it is the first time $i$ receives this request, $i$ verifies the signature of the request. If the signature is valid, $i$ sends a PROPAGATE message to the other nodes. When a node receives $f + 1$ PROPAGATE messages for a given request, the request is ready to be given to the replicas of the $f + 1$ protocol instances running locally, for ordering. As proved in Section VI, only $f+1$ PROPAGATE messages are sufficient to guarantee that if malicious primaries can order a given request, all correct primaries will eventually be able to order the same request. Note that the replicas do not order the whole request but only its identifiers (i.e., the client id, request id and digest). Not only the whole request is not necessary for the ordering phase, but it also improves the performance as there is less data to process.

**3. 4. and 5. The replicas of each protocol instance execute a three phase commit protocol to order the request.** When the primary replica $p$ of a protocol instance receives a request, it sends a PRE-PREPARE message $\langle \text{PRE-PREPARE}, v, n, c, rid, d\rangle_{\vec{\mu}_p}$ authenticated with a MAC authenticator for every replica of its protocol instance (Step 3 in the figure). A replica that is not the primary of its protocol instance stores the message and expects a corresponding PRE-PREPARE message. When a replica receives a PRE-PREPARE message from the primary of its protocol instance, it verifies the validity of the MAC. It then replies to the PRE-PREPARE message by sending a PREPARE message to all other replicas, only if the node it is running on already received $f$+1 copies of the request. Without this verification, a malicious primary may collude with faulty clients that would send correct requests only to him, in order to boost the performance of the protocol instance of the malicious primary at the expense of the other protocol instances. Following the reception of $2f$ matching

[2] Deviations performed by clients or nodes on the security protocols, e.g., a client that would continuously initiate the key exchange protocol, are considered out of the scope of the paper and will be studied as future work

PREPARE messages from distinct replicas of the same protocol instance that are consistent with a PRE-PREPARE message, a replica $r$ sends a commit message $\langle \text{COMMIT}, v, n, d, r \rangle_{\vec{\mu}_r}$ that is authenticated with a MAC authenticator (Step 5 in the figure). After the reception of $2f + 1$ matching COMMIT messages from distinct replicas of the same protocol instance, a replica gives back the ordered request to the node it is running on.

**6. The nodes execute the request and send a reply message to the client.** Each time a node receives an ordered request from a replica of the master instance, the request operation is executed. After the operation has been executed, the node $i$ sends a REPLY message $\langle \text{REPLY}, u, i \rangle_{\mu_{i,c}}$ to client $c$ that is authenticated with a MAC, where $u$ is the result of the request execution (Step 6 in the figure). When the client $c$ receives $f+1$ valid and matching $\langle \text{REPLY}, u, i \rangle_{\mu_{i,c}}$ from different nodes $i$, it accepts $u$ as the result of the execution of the request.

### C. Monitoring mechanism

RBFT implements a monitoring mechanism to detect whether the master protocol instance is faulty or not. This monitoring mechanism works as follows. Each node keeps a counter $nbreqs_i$ for each protocol instance $i$, which corresponds to the number of requests that have been ordered by the replica of the corresponding instance (i.e. for which $2f + 1$ COMMIT messages have been collected). Periodically, the node uses these counters to compute the throughput of each protocol instance replica and then resets the counters. The throughput values are compared as follows. If the ratio between the throughput of the master instance $t_{master}$ and the average throughput of the backup instances $t_{backup}$ is lower than a given threshold $\Delta$, then the primary of the master protocol instance is suspected to be malicious, and the node initiates a protocol instance change, as detailed in the next section. The value of $\Delta$ depends on the ratio between the throughput observed in the fault-free case and the throughput observed under attack. Note that the state of the different protocol instances is not synchronized: they can diverge and order requests in different orders without compromising the safety nor the liveness of the protocol.

In addition to the monitoring of the throughput, the monitoring mechanism also tracks the time needed by the replicas to order the requests. This mechanism ensures that the primary of the master protocol instance is fair towards all the clients. Specifically, each node measures the individual latency of each request, say $lat_{req}$ and the average latency for each client, say $lat_c$, for each replica running on the same node. A configuration parameter, $\Lambda$, defines the maximal acceptable latency for any given request. Another parameter, $\Omega$, defines the maximal acceptable difference between the average latency of a client on the different protocol instances. These two parameters depend on the workload and on the experimental settings. When the node sends a request to the various replicas running locally for ordering, it records the current time. Then, when it receives the corresponding ordered request, it computes its latency $lat_{req}$ and the average latency for all the requests of this client $lat_c$. If this request has been ordered by the replica of the master instance and if $lat_{req}$ is greater than $\Lambda$, or the difference between $lat_c$ and the average latency for this client on the other protocol instances is greater than $\Omega$, then the node starts a protocol instance change. Note that we define the value of the different parameters, $\Delta$, $\Lambda$ and $\Omega$ both theoretically and experimentally, as described in Section VI: their value depends on the cost of the cryptographic operations and on the network conditions.

### D. Protocol instance change mechanism

In this section, we describe the protocol instance change mechanism that is used to replace the faulty primary at the master protocol instance. Because there is only at most one primary per node in RBFT, this also implies to replace all the primaries on all the protocol instances.

Each node $i$ keeps a counter $cpi_i$, which uniquely identifies a protocol instance change message. When a node $i$ detects too much difference between the performance of the master instance and the performance of the backup instances (as detailed in the previous section), it sends an $\langle \text{INSTANCE\_CHANGE}, cpi_i, i \rangle_{\vec{\mu}_i}$ message authenticated with a MAC authenticator to all the other nodes.

When a node $j$ receives an INSTANCE_CHANGE message from node $i$, it verifies the MAC and handles it as follows. If $cpi_i < cpi_j$, then this message was intended for a previous INSTANCE_CHANGE and is discarded. On the contrary, if $cpi_i \geq cpi_j$, then the node checks if it should also send an INSTANCE_CHANGE message. It does so only if it also observes too much difference between the performance of the replicas. Upon the reception of $2f + 1$ valid and matching INSTANCE_CHANGE messages, the node increments $cpi$ and initiates a view change on every protocol instance that runs locally. As a result, each protocol instance elects a new primary and the malicious replica of the master instance is no longer the primary.

## V. Implementation

We have implemented RBFT in C++ using the Aardvark code base. Similarly to Aardvark, RBFT adopts separate Network Interface Controllers (NICs). Not only this allows to avoid client traffic to slow down node-to-node communication, but it also protects against a flooding attack performed by faulty nodes. In this situation, RBFT closes the NIC of the faulty node for a given time period, which gives time to the faulty node to restart or get repaired without penalizing the performance of the whole system. Moreover, as our implementation of Aardvark, RBFT uses TCP. We use it because it eases the development task: on the contrary of UDP, TCP provides a loss-less, FIFO communication channel. Previous works (e.g., Zyzzyva [12] or PBFT [4]) have shown that the bottleneck in BFT protocols is actually cryptography, not network usage. Interestingly, the BFT protocol that currently provides the highest throughput uses TCP [9]. This protocol, called Chain, implies the smallest number of cryptographic operations at the bottleneck replicas, hence its very good performance. For comparison we also have implemented a UDP version of RBFT. We show, in Section VII-B, that this implementation provides the same performance.

Figure 6 depicts the architecture of a single node in RBFT in the case when one fault is tolerated (i.e. $f = 1$). We observe that two replicas belonging to two different protocol instances are hosted by the node: replica 0 from protocol instance 0 (noted $p_{0,0}$ in the figure) and replica 0 from protocol instance 1 (noted $p_{0,1}$ in the figure). We also observe that the node uses $3f + 1 = 4$ NICs, as Aardvark: 1 NIC for the communication with the clients, and 1 NIC per other node.

As depicted in the figure, a number of modules run on each node. We describe the behavior of these modules below.

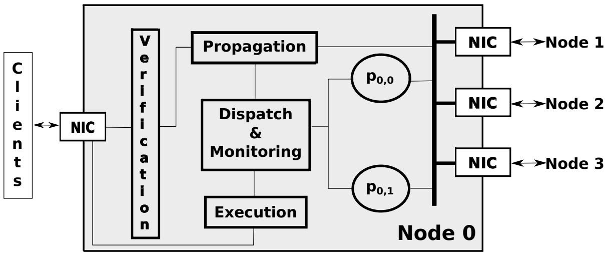


Fig. 6: Architecture of a single node ($f = 1$). The replicas (in circles) are processes that are independent from the different modules, implemented as threads (in rectangles).

When a request is received from a client, it is verified by the *Verification* module. If the request is correct then the *Propagation* module disseminates it to all the other nodes through the node-to-node NICs and waits for similar messages from them. Once it has received $f + 1$ such messages, the *Propagation* module sends the request to the *Dispatch & Monitoring* module, which logs the request and gives it to the replicas of the protocol instances running locally (i.e., $p_{0,0}$ and $p_{0,1}$ in the figure). The replicas interact with other replicas of the same protocol instance (running on the other nodes) via separate NICs, to order the request. Then, each replica gives back the ordered request to the *Dispatch & Monitoring* module. The latter sends ordered requests coming from the master protocol instance to the *Execution* module, which executes them and replies to the client.

The *Verification*, *Propagation*, *Dispatch & Monitoring* and *Execution* modules are implemented as separate threads. Moreover, the replicas are implemented as independent processes. The various threads and processes are deployed on distinct cores (our machines have 8 cores). Leveraging multicore machines improves the performance of the system, as the different protocol instances can really execute concurrently.

## VI. RBFT THEORETICAL ANALYSIS

We present in this section a theoretical analysis of RBFT. We first present a throughput analysis in Section VI-A. Given this analysis, we show how to compute the minimal acceptable ratio between the performance of the master instance and the performance of the backup instances in Section VI-B. Then, in Section VI-C, we prove that having $f+1$ protocol instances is necessary and sufficient. Finally, we prove that, thanks to the PROPAGATE phase, all the correct nodes eventually send the request to the ordering (Section VI-D) and that if $2f + 1$ correct nodes decide to perform a protocol instance change, then a protocol instance change will eventually be performed by every correct node (Section VI-E).

### A. *Throughput analysis*

We analyze in this section the throughput of RBFT during both gracious and uncivil intervals. We compute RBFT peak performance in Theorem 1 and show in Theorem 2 that the performance during an uncivil execution is within a constant factor of the peak performance.

We consider a multi-core machine where each replica and each module (i.e., the verification, propagate, dispatch and monitoring and execution modules) run in parallel on distinct cores. Each core has a speed of $\kappa$ Hz. We consider only the computational costs of the cryptographic operations (i.e., verifying signatures, generating MACs, and verifying MACs, requiring $\theta, \alpha$ and $\alpha$ cycles, respectively). We ignore the cost of request execution and the generation of client replies, as the throughput of the protocol instances is monitored ahead of the execution of client requests. Moreover, this execution takes place in a separate core and thus does not impact the performance of the individual protocol instances.

**Theorem 1.** Consider a gracious view during which the system is saturated, all requests come from correct clients, all nodes are correct and the replicas order batches of requests of size $b$. For each protocol instance, the throughput is at most $t_{peak} = \frac{\kappa}{max(\theta+\alpha(n+f),\ \alpha\frac{2n+4f-1}{b})}$ req/s.

*Proof.* First, we compute the number of operations performed by each node to send a request to the ordering stage. For one request, there is at most one MAC and one signature verification, performed either when the request is received from the client, or when it is received during the PROPAGATE phase. Then, the node generates $n - 1$ MACs to create the PROPAGATE. Each node waits for $f$ PROPAGATE, on which it verifies the MAC. Therefore, the cost for one request up to the ordering stage is $\theta + \alpha(n + f)$.

Then we differentiate the operations performed by the primary of a protocol instance and the non-primary replica. The primary generates $n - 1$ MACs for the PRE-PREPARE, while a non-primary replica verifies the PRE-PREPARE and generates $n - 1$ MACs for the PREPARE. Then, each replica verifies $2f$ PREPARE, after which it generates $n - 1$ MACs for the COMMIT, and finally verifies $2f + 1$ COMMIT. As a result, for each request, the cost at the primary is $\alpha\frac{2n+4f-1}{b}$ and the cost at a non-primary replica is $\alpha\frac{2n+4f}{b}$.

Given that the propagate module and the replicas run in parallel, the throughput is at most $t_{peak} = \frac{\kappa}{max(\theta+\alpha(n+f),\ \alpha\frac{2n+4f-1}{b})}$ req/s. □

**Lemma 1.** Consider an uncivil execution during which the system is saturated, but only a fraction $g$ of requests generated by clients are correct. Moreover the node running the primary of the master instance is correct and at most $f$ nodes are Byzantine. All the nodes receive the same proportion of correct requests and the replicas receive batches of requests of size $b$. For each protocol instance, the throughput is $t_{uncivil} = \frac{\kappa}{max(\frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1),\ \alpha\frac{2n+4f-1}{b})}$ req/s.

*Proof.* We compute the number of operations performed by each node to send a request to the ordering stage. Note that, thanks to the blacklisting mechanism, a node that sends too many invalid messages or too much messages is isolated and no longer takes part in the system. For one correct request, there is at most one MAC and one signature verification. The propagate thread generates $n-1$ MACs to compute the PROPAGATE. Every correct node waits for at most $f$ PROPAGATE, for which it verifies the MAC. Consequently, the cost for one request up to the ordering stage is $\frac{\theta}{g} + \alpha(\frac{1}{g} + n + f - 1)$.

Then, the cost at the ordering step is the same cost as the one in the fault-free case: the cost at the primary is $\alpha\frac{2n+4f-1}{b}$ and the cost at a non-primary replica is $\frac{\alpha(2n+4f)}{b}$.

Consequently, the throughput is $t_{uncivil} = \frac{\kappa}{max(\frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1),\ \alpha\frac{2n+4f-1}{b})}$ req/s. □

**Theorem 2.** Consider an uncivil execution during which the system is saturated, but only a fraction $g$ of requests generated by clients are correct. Moreover the node running the primary of the master instance is correct and at most $f$ nodes are Byzantine. All nodes receive the same proportion of correct requests and the replicas order batches of requests of size $b$. Then the performance is within a constant factor $C \leq 1$ of the performance in a gracious execution in which the batch size is the same.

*Proof.* From Theorem 1 and Lemma 1 we know the performance during a gracious and an uncivil execution. Moreover, the ratio $\frac{\alpha}{\theta}$ tends to 0. Let $v1 = \frac{\theta/\alpha+1-1/b}{10/b-4}$ and $v2 = \frac{\theta/(g\alpha)+1/g-1/b}{10/b-4}$ ($v1 \leq v2$). Moreover, from the formulas of $t_{peak}$ and $t_{uncivil}$, let $A = \theta + \alpha(n + f)$, $B = \alpha\frac{2n+4f-1}{b}$ and $C = \frac{\theta}{g} + \alpha(\frac{1}{g} + n + f - 1)$. We need to distinguish three cases [3]:

1) $A > B$ and $C > B$. It happens when $f < v_1$. In this case, the ratio between the uncivil and gracious throughputs $\frac{t_{uncivil}}{t_{peak}}$ is $C = g$.
2) $A > B$ and $C \leq B$. It happens when $f \geq v_1$ and $f < v_2$. In this case, the ratio between the uncivil and gracious throughputs $\frac{t_{uncivil}}{t_{peak}}$ tends to $C = 0$.
3) $A \leq B$ and $C \leq B$. It happens when $f \geq v_2$. In this case, the ratio between the uncivil and gracious throughputs $\frac{t_{uncivil}}{t_{peak}}$ is $C = 1$.

□

### B. Monitoring the protocol instances

We define in this section how to compute the minimal acceptable ratio between the performance of the master instance and the performance of the backup instances (Definition 1).

**Definition 1** In Section IV-C we have seen that the performance of the master instance is compared with the max performance of the backup instances. If the difference is greater than $\Delta$, then the node initiates a protocol instance change.

More precisely, the nodes monitor the throughput of the different protocol instances. They compute the ratio between the throughput of the master instance (noted $t_{master}$) and the max throughput of the backup instances (noted $t_{backup}$). They expect $\frac{t_{master}-t_{backup}}{t_{master}} < \Delta$. We know that the maximum acceptable difference, such that no protocol instance change is initiated, is between the performance during a gracious execution (Theorem 1) and an uncivil execution (Lemma 1). Consequently, $\Delta = \frac{t_{uncivil}-t_{peak}}{t_{uncivil}}$. □

**Lemma 2.** If the master protocol instance is malicious and provides a throughput lower than the throughput during an uncivil execution, then it is detected and a protocol instance change is initiated.

*Proof.* Let $t_{delay}$ be the throughput of the master instance. According to Lemma 1, the throughput of the backup instances is in the worst case $t_{uncivil}$. By Definition 1, a protocol instance change is initiated if $\frac{t_{master}-t_{backup}}{t_{master}} < \Delta$. We now compute $t_{delay}$: $\frac{t_{master}-t_{backup}}{t_{master}} < \Delta \Leftrightarrow \frac{t_{delay}-t_{uncivil}}{t_{delay}} < \frac{t_{uncivil}-t_{peak}}{t_{uncivil}} \Leftrightarrow t_{uncivil}(t_{delay} - t_{uncivil}) < t_{delay}(t_{uncivil} - t_{peak}) \Leftrightarrow t_{uncivil}t_{uncivil} > t_{delay}t_{peak} \Leftrightarrow t_{delay} < \frac{t_{uncivil}}{t_{peak}}t_{uncivil}$.

From Theorem 2, we know that $\frac{t_{uncivil}}{t_{peak}}$ tends to $C$. Thus, a protocol instance change is initiated if $t_{delay} < C{*}t_{uncivil}$. As $C \leq 1$, a protocol instance change is initiated if the throughput during the attack is less than the throughput during an uncivil execution. In other words, the malicious primary of the master instance cannot provide a throughput less than the throughput during an uncivil execution without being replaced. □

### C. Number of protocol instances

In this section we prove that having $f+1$ protocol instances is necessary and sufficient (Theorem 3).

**Lemma 3.** A protocol instance cannot provide a throughput higher than the throughput observed in the fault-free case.

*Proof.* From Theorem 1 we know that the throughput depends on the number of MAC and signature verifications, and on the number of MAC generations. In order to provide a throughput higher than the peak throughput in the fault-free case, a protocol instance needs to perform less verifications

[3] The fourth combination, $A \leq B$ and $C > B$, can not happen as it would mean to have $v_1 > f > v_2$.

and generations. However, the formula of $t_{peak}$ considers the case where, with at most $f$ malicious replicas (i.e. the safety property of the protocol holds), the number of verifications and generations is minimal. Consequently, a protocol instance cannot provide a throughput higher than $t_{peak}$. □

**Invariant 1.** Consider an uncivil execution during which the system is saturated and at most $f$ nodes are byzantine. As long as the primary of the master instance is correct, a protocol instance change will never happen.

*Proof.* We consider an uncivil execution, i.e. a proportion $g$ of requests are correct and at most $f$ nodes are Byzantines. Moreover, the primary of the master instance is correct. We suppose that a protocol instance change eventually happen and show that we find a contradiction.

There is eventually a protocol instance change means that there is eventually at least $2f + 1$ nodes that receive $2f + 1$ INSTANCE_CHANGE.

There is at most $f$ byzantine nodes, which may continuously send valid INSTANCE_CHANGE. Thus it is sufficient that $f + 1$ correct nodes send an INSTANCE_CHANGE. For that, they need to observe better performance at the backup instances than at the master instance by a pre-determined threshold $\Delta$ (cf. Definition 1). Specifically, the malicious nodes have to augment the performance of the backup instances and reduce the performance of the master instance.

We consider the following attack, which is the one where the difference between the master instance and the backup instances is maximal. The clients send a proportion $g < 1$ of correct requests to the master instance only and a proportion $g = 1$ of correct requests to the backup instances. Moreover, the malicious nodes flood only the master instance and the node on which its primary runs (however they do not flood the other instances running on this node). Thus the throughput of the master instance (noted $t_{master}$) is given by Lemma 1 and the maximal throughput of the backup instances (noted $t_{backup}$) is given by Theorem 1.

An INSTANCE_CHANGE is sent by a correct node if the ratio between $t_{master}$ and $t_{backup}$ is lower than $\Delta$. As shown in Definition 1, $\Delta$ is precisely computed as the ratio between the peak throughput and the uncivil throughput. Moreover, we know from Lemma 3 that $t_{backup} \leq t_{peak}$. As a consequence, an INSTANCE_CHANGE is never sent. □

**Invariant 2.** Consider an uncivil execution during which the system is saturated and at most $f$ nodes are Byzantines. If the primary of the master instance is byzantine and if its performance is not acceptable, then a protocol instance change will eventually happen.

*Proof.* We consider an uncivil execution, i.e. a proportion $g$ of requests are correct and at most $f$ nodes are Byzantines. Moreover, the primary of the master instance is malicious. We suppose that a protocol instance change will never happen and show that we find a contradiction.

We consider the following attack, which is the one where the difference between the master instance and the backup instances is maximal. The clients send a proportion $g < 1$ of correct requests to the backup instances. As the master instance is malicious, there is no requirement on the value of $g$ for it. Moreover, the malicious nodes flood all the correct nodes. Thus the throughput of the master instance (noted $t_{master}$) is the throughput resulting of the attack, the throughput of the other malicious instances nodes is also $t_{master}$, and the throughput of the correct instances is $t_{uncivil}$ (cf. Lemma 1). The total number of instances per node is $f + 1$. Consequently, there are $f$ backup instances, among which at most $f - 1$ are malicious and at least 1 is correct. Because $t_{master} < t_{uncivil}$, the maximal throughput of the backup instances is equal to $t_{uncivil}$.

An INSTANCE_CHANGE is sent by a correct node if the ratio between $t_{master}$ and $t_{uncivil}$ is lower than $\Delta$ (cf. Definition 1). That is to say, the ratio between $t_{master}$ and $t_{uncivil}$ is lower than the ratio between the throughput during an uncivil execution and the throughput during a gracious execution. Given Lemma 2, if $t_{master} < t_{uncivil}$, then every correct node will detect the attack and will send an INSTANCE_CHANGE. Therefore, every correct node eventually receives $2f + 1$ INSTANCE_CHANGE and votes a protocol instance change. □

**Theorem 3.** Consider an uncivil execution during which the system is saturated and at most $f$ nodes are Byzantine. Then, having $f + 1$ protocol instances is necessary and sufficient for the malicious nodes and the clients to be unable to alter the performance of the different instances such that the performance provided by the system are not adequate.

*Proof.* From Invariants 1 and 2 we know that if the number of protocol instances is $f + 1$, then the malicious nodes and clients cannot force a protocol instance change or alter the performance provided by the system without triggering a protocol instance change. We now present an attack where, with less than $f + 1$ protocol instances, one of the invariants is violated.

We consider $f$ protocol instances and the attack presented in the proof of Invariant 2. We show that this invariant is violated, i.e. the primary of the master instance is malicious and is not detected. The proof is trivial: considering $f$ malicious nodes, all the protocol instances run a malicious primary. As a result, they can all provide a null throughput. The primary of the master protocol instance is not detected, which violates Invariant 2. □

### *D. Propagate phase*

In this section, we prove that, thanks to the PROPAGATE phase, all the correct nodes eventually send the request to the ordering.

**Invariant 3.** If a correct node sends a PROPAGATE, then every correct node will eventually receive $f + 1$ PROPAGATE.

*Proof.* By the contraposition: we suppose there exists a correct node $n_1$ which sends a PROPAGATE to all the nodes and there exists a correct node $n_2$ which will never receives $f+1$ PROPAGATE. We show that we arrive at a contradiction.

The node $n_1$ sends a PROPAGATE to all the nodes. It will eventually be received by all the correct nodes (as the system is eventually synchronous). Let $n_3$ be a correct node which receives the PROPAGATE from $n_1$. We need to distinguish three cases:

1) it has already received the REQUEST. According to the protocol algorithm (see Section IV-B), it has already sent a PROPAGATE to all the nodes;
2) it has never received the REQUEST and has already received a valid PROPAGATE for this request. According to the protocol algorithm, it has already sent a PROPAGATE to all the nodes;
3) it has never received the REQUEST nor a valid PROPAGATE for this request. According to the protocol algorithm, it sends a PROPAGATE to all the nodes.

In every cases a PROPAGATE is sent by $n_3$.

As there are at least $2f+1$ correct nodes, at least $2f+1$ valid PROPAGATE are sent and eventually received by every correct node. Thus, every correct node, including $n_2$, eventually receives at least $2f+1$ PROPAGATE. This contradicts the fact that $n_2$ will never receives $f+1$ PROPAGATE. □

**Invariant 4.** If a request is sent to the ordering by a correct node, then it will eventually be sent to the ordering by all the correct nodes.

*Proof.* By the contraposition: we suppose a request is sent to the ordering by a correct node $n_1$ and there exists a node $n_2$ which will never send the request to the ordering. We show that we arrive at a contradiction.

The node $n_2$ will never send the request to the ordering is equivalent to it will never receive at least $f+1$ valid PROPAGATE.

The node $n_1$ sends the request to the ordering is equivalent to it has received $f+1$ valid PROPAGATE, which means there are $f+1$ nodes which have sent a valid PROPAGATE to $n_1$. According to the algorithm a correct node that receives a valid PROPAGATE or a valid REQUEST sends a PROPAGATE. Consequently there exists a correct node ($n_1$ in this case) which has sent a valid PROPAGATE to all the nodes.

According to the Invariant 3, if a correct node sends a valid PROPAGATE, then every correct node will eventually receive $f+1$ PROPAGATE.

Therefore $n_2$, which is correct, eventually receives $f+1$ PROPAGATE. This contradicts the fact that $n_2$ will never receive at least $f+1$ PROPAGATE. □

### *E. Protocol instance change*

In this section we prove that if $2f+1$ correct nodes decide to perform a protocol instance change, then a protocol instance change will eventually be performed by every correct node. This proof refers to the algorithm presented in Section IV-D.

**Invariant 5.** If $2f+1$ correct nodes decide to perform a protocol instance change, then a protocol instance change will eventually be performed by every correct node.

*Proof.* By the contraposition: we suppose that $2f+1$ correct nodes decide to perform a protocol instance change and there exists a correct node that will never perform a protocol instance change. We show that we arrive at a contradiction.

According to the algorithm, saying that $2f+1$ correct nodes decide to perform a protocol instance change is equivalent to saying that $2f+1$ nodes send a valid INSTANCE_CHANGE to all the nodes. Thus every correct node will eventually receive $2f+1$ valid INSTANCE_CHANGE.

We suppose there exists a correct node that will never perform a protocol instance change. This means there exists a correct node that will never receive $2f+1$ valid INSTANCE_CHANGE. This contradicts the fact that every correct node will eventually receive $2f+1$ valid INSTANCE_CHANGE. □

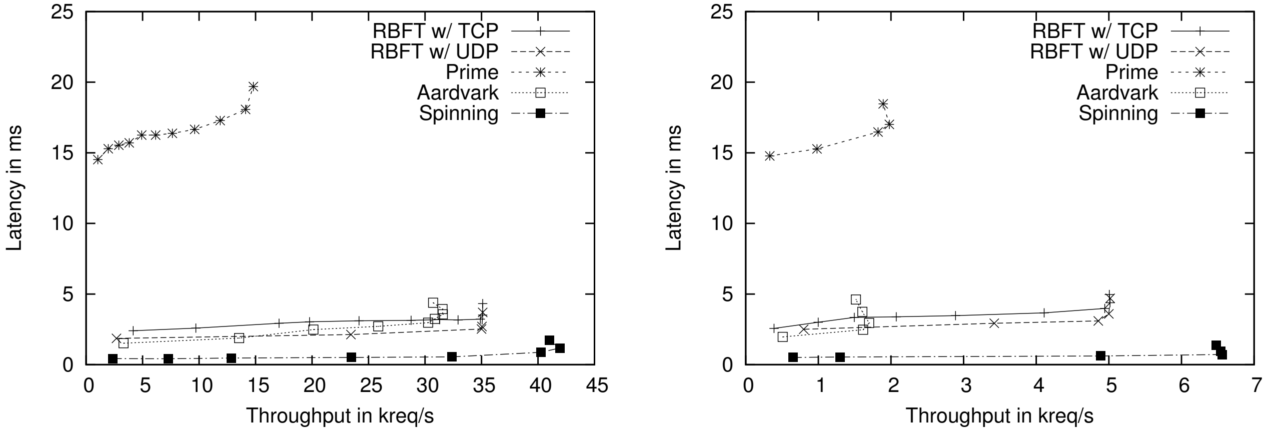
## VII. Performance evaluation

In this section we present a performance analysis of RBFT. After describing the hardware and software settings we use, we start by comparing the performance of RBFT against the state-of-the-art robust BFT protocols described in Section III in the fault-free case. Second, we assess the monitoring precision of RBFT. Finally, we present a performance analysis of RBFT under attack.

Our evaluation makes the following points. We first show that RBFT has comparable performance to state-of-the art protocols in the fault-free case. Second, we show that under the two worst possible attacks, where faulty clients collude with faulty replicas, the throughput degradation caused to RBFT is limited to 3%.

### *A. Experimental settings*

We evaluate the performance of Prime, Aardvark, Spinning and RBFT on a cluster composed of eight Dell PowerEdge T610 and two Dell Precision WorkStation T7400. The T610 host two quad-core Intel Xeon E5620 processors clocked at 2.40GHz with 16GB of RAM and ten network interfaces. The T7400 host two quad-core Intel Xeon E5410 processors clocked at 2.33GHz with 8GB of RAM and five network interfaces. All these machines run a Linux kernel version 2.6.32 and are interconnected via a Gigabit switch. We run experiments with up to two Byzantine faults, i.e., $f \leq 2$. The nodes are always launched on the T610 machines. The remaining machines are used to run the clients. Unless specified, we



(a) With requests of 8B.

(b) With requests of 4kB.

Fig. 7: Latency vs. throughput for various robust BFT protocols ($f = 1$).

consider the TCP implementation of RBFT configured with $f = 1$.

We run our experiments under two different workloads: a static load, where the system is saturated and the clients send their requests at a constant rate, and a dynamic workload, where the incoming throughput varies. We present the workload used for requests of 8B. Similar workloads have been used for the other requests sizes with possibly fewer clients as the peak throughput has been reached with fewer clients. The experiment starts with a single client. We then progressively increase the number of clients up to 10. Then we simulate a load spike, with 50 clients. At last, the number of clients progressively decreases, until there is only one client issuing requests. When the load is dynamic, we consider the average throughput observed on the whole experiment. A similar workload has already been used in [9].

Finally, the clients send their requests in an open-loop as defined in [15], i.e., they do not wait for the reply of a request before sending a new one.

### B. *Fault-free case*

In this section we present the fault-free performance of RBFT, Prime, Aardvark and Spinning. We also show the performance of RBFT when the communication protocol used between the nodes is UDP instead of TCP. We run the experiments under a static load, with requests of either 8B or 4kB. Figures 7a and 7b present the latency achieved by the different protocols as a function of the throughput, for requests of 8B and 4kB, respectively.

Fist of all, we observe that Spinning provides the highest peak throughput and the lowest latency. Specifically, with requests of 8B, its throughput is 20% higher than the throughput of RBFT and Aardvark, and 183% higher than the throughput of Prime. Similarly, with requests of 4kB the throughput of Spinning is 30% higher than the throughput of RBFT. Its good performance is mostly due to the fact that it relies only on MACs, while the other protocols (RBFT, Prime and Aardvark) use signatures in addition to MACs. Although signatures are an order of magnitude more costly than MACs, they are necessary to prevent certain attacks [6]. Moreover, Spinning uses UDP multicast for both the communication between the replicas and the communication between the clients and the replicas. These two properties allow Spinning to provide a very low latency.

The second observation we make is that the performance of RBFT is higher than the performance of Aardvark. For requests of 8B, the peak throughput of RBFT is 35 kreq/s, while the peak throughput of Aardvark is 31.6 kreq/s. Similarly, for requests of 4kB, the peak throughput of RBFT is 5 kreq/s, while the peak throughput of Aardvark is 1.7 kreq/s. This might seem surprising as the BFT protocol that is run by each protocol instance in RBFT mimics Aardvark and uses the same code base. The reason why RBFT is more efficient is that it does not perform regular view changes (remember that view changes are replaced by protocol instance changes that occur only when there are faults). To confirm this explanation, we disabled the view changes in Aardvark and we obtained the same performance as RBFT for small requests. For bigger requests, the better performance of RBFT is due to the fact that the protocol instances only order requests identifiers instead of the whole request. When they order the whole requests, the peak throughput drops to 1.8 kreq/s for requests of 4kB.

Third, we observe that Prime provides the worst performance, especially in terms of latency. Indeed, its latency is an order of magnitude higher than the latency of the other protocols for both request sizes. This high latency is due to the fact that the Prime protocol solely relies on signatures, which are known to be slower than MACs. Moreover, it is due to the fact that in Prime, the primary does not send request ordering messages following the flow of arrival of requests, but periodically.

Finally, we observe that the UDP and TCP implementations of RBFT exhibit the same peak throughput. The only difference is that the UDP implementation provides a latency 22% lower (resp. 18%) than the TCP implementation, for requests of 8B (resp. 4kB). This increase in latency is due to the mechanisms TCP uses to enforce robustness (acknowledgements, flow control, etc.). Note that the choice of the communication mechanism has no impact on the performance of RBFT under attack, as we found similar results while using TCP or UDP.

### *C. RBFT monitoring precision*

RBFT detects that the primary of the master protocol instance is faulty by periodically monitoring the performance of the protocol instances. In this section, we analyse the precision of this monitoring under both a static and a dynamic workload. More specifically, we are interested in the impact of the monitoring period on the minimal throughput ratio observed by the nodes, and on the distribution of the observed throughput ratios. The minimal ratio allows one to detect if the primary of the master protocol instance is faulty or not. A node considers that the primary of the master protocol instance is faulty as soon as the observed ratio is lower than a specific threshold. If the minimal ratio is too low in the fault-free case, then it will become more difficult to detect an attack. Furthermore, if the monitoring period is high, then an attacker will be able to cause damage to the system during a long time. Finally, the distribution of the observed ratios assess their volatility.

Table V presents the minimal ratio observed respectively under a static and a dynamic workload, for requests of 8B and 4kB. We vary the monitoring period from 1ms up to 1 second. Note that a monitoring period greater than 1 second does not improve the monitoring precision significantly. We observe that for a period of 1ms, which is lower than the system latency, the nodes cannot detect an attack as the minimal observed ratio is $-\infty$. As the monitoring period increases, the minimal ratio becomes closer to 0. That is to say, the nodes observe, in the worst case, less difference between the performance of the protocol instances. For instance, with requests of 8B and under a static load, it is of $-7$ with a monitoring period of 10ms, and only $-0.003$ with a monitoring period of 1 second.

| | 1ms | 10ms | 100ms | 1s |
|---|---|---|---|---|
| 8B | $-\infty$ | $-7$ | $-0.2$ | $-0.003$ |
| 4kB | $-\infty$ | $-4$ | $-0.1$ | $-0.002$ |

(a) Static workload.

| | 1ms | 10ms | 100ms | 1s |
|---|---|---|---|---|
| 8B | $-\infty$ | $-1$ | $-0.07$ | $-0.004$ |
| 4kB | $-\infty$ | $-0.7$ | $-0.05$ | $-0.005$ |

(b) Dynamic workload.

TABLE V: RRBFT monitoring precision: minimal ratio observed for different monitoring periods, for requests of 8B and 4kB, under both a static and a dynamic workload.

Figure 8 presents the cumulative percentage of the observed ratios for different monitoring periods in the fault-free case with requests of 4kB and a static load. Results are similar with requests of 8B and/or under a dynamic workload. We observe that, as the monitoring period increases, the distribution of observed ratios becomes closer to 0. This implies that the difference between the observed performance of the different protocol instances becomes smaller, hence a faulty primary at the master protocol instance has less liberty to attack the system.

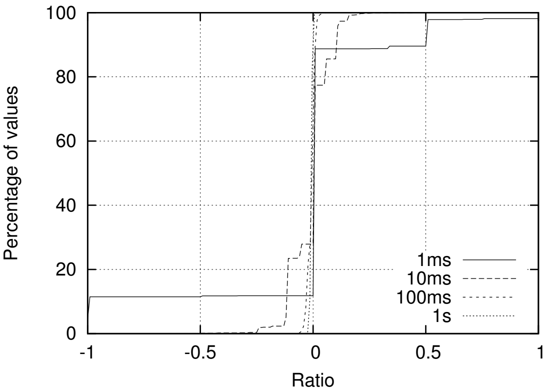


Fig. 8: Cumulative percentage of the observed ratios for different monitoring periods in the fault-free case with requests of 4kB and a static load.

In the rest of the evaluation, we set the monitoring period to 1 second.

### *D. RBFT under attack*

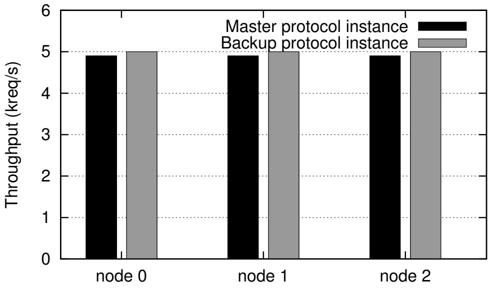
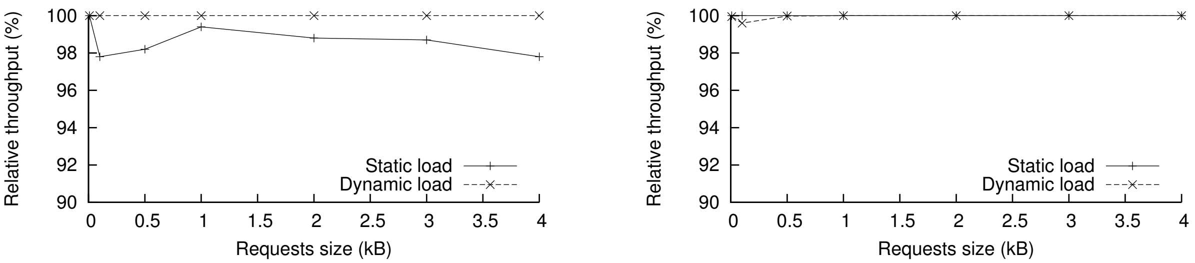


Fig. 10: Throughput measured by the different nodes under worst attack 1 ($f = 1$, static workload, 4kB requests).

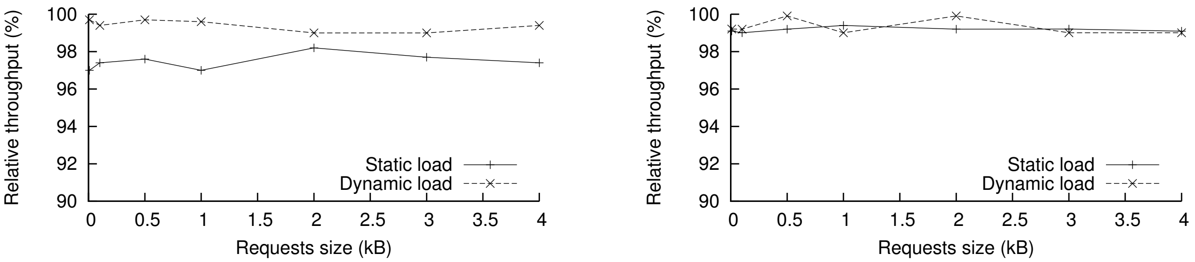
In this section, we first present the two worst attacks that can be done to RBFT. These attacks show the maximal damage an attacker can make on the protocol in two different situations: (1) when the primary of the master instance is correct (*worst-attack-1*), and (2) when the primary of the master instance is malicious (*worst-attack-2*). We consider these attacks as worst attacks in their respective context, because in each case, all the $f$ malicious nodes and any number of malicious clients collude to reduce the system performance. We run the experiments in two configurations: $f = 1$ and $f = 2$. We show that the maximal throughput degradation is 3%. Finally, we show that the monitoring of the latency prevents a faulty primary at the master instance not to be fair with a subset of the clients when constructing an ordering message.



(a) $f = 1$

(b) $f = 2$

Fig. 9: RBFT throughput under *worst-attack-1* relative to the throughput in the fault-free case, for both a static and a dynamic load.



(a) $f = 1$

(b) $f = 2$

Fig. 11: RBFT throughput under *worst-attack-2* relative to the throughput in the fault-free case, for both a static and a dynamic load.

*1) Worst-attack-1:* In this attack, there are $f$ faulty nodes and all clients are faulty. The primary of the master protocol instance is correct (i.e. it runs on a correct node). The goal of the attack is to decrease as much as possible the performance of the master instance, in order to induce a protocol instance change. If this attack is succesful, then it is possible for malicious nodes to put a malicious replica as the primary of the master instance. A faulty node can cause the following damages: first, it can flood other nodes; second, the replicas it hosts are also faulty and they can take actions to reduce the throughput of the protocol. Let $p$ be the node on which the primary of the master protocol instance runs. The attack we are performing is the following: (i) the clients (that are all faulty) send requests that can be verified by all nodes, but the one on which the primary of the master protocol instance runs; (ii) the $f$ faulty nodes flood $p$ with invalid PROPAGATE messages of the maximal size; (iii) the faulty replicas of the master protocol instance flood the correct ones with invalid messages of the maximal size; (iv) the faulty replicas of the master protocol instance do not take part in the protocol.

We run this experiment with a request size ranging from 8B to 4kB. Figure 9 presents the throughput under attack relative to the throughput in the fault-free case, under both the static and dynamic load described earlier. We observe that RBFT is robust: the throughput loss is below 2.2%, under a static load and that it is null with the dynamic load, when $f = 1$. When at most two faults are tolerated, i.e., $f = 2$, we observe that the throughput loss is even lower: at most 0.4%.

In order to illustrate the behavior of RBFT, we registered the throughput measures performed by the monitoring mechanism of the different nodes. Results are depicted in Figure 10 for the static workload, with a request size of 4kB, and when at most one fault is tolerated (we observed similar results in other configurations). This figure first shows that each node measures the same throughput. Moreover, it shows that the throughput measured for the master protocol instance is very close to the throughput measured for the backup protocol instance (2% difference). Note that we do not represent the values reported by the monitoring module of node 3 as this is the faulty node in this experiment (it can thus report arbitrary values).

*2) Worst-attack-2:* In this attack, there are $f$ faulty nodes and all clients are faulty. The primary of the master protocol instance is faulty (i.e. it runs on a faulty node). Faulty clients and nodes aim at decreasing the performance of the backup protocol instances in order to give a margin for the faulty primary of the master protocol instance to delay requests without being detected. Towards this purpose, the faulty nodes collude with the faulty clients, and take the following actions: (i) the (faulty) clients send invalid requests to the correct nodes; (ii) the $f$ faulty nodes flood the correct nodes with invalid messages of the maximal size and do not participate in the PROPAGATE phase; (iii) the replicas of the backup protocol instances executing on the $f$ faulty nodes flood the correct ones with invalid messages of the maximal size, and do not take part in the protocol.

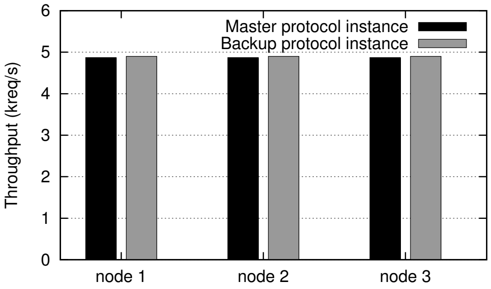


Fig. 12: Throughput measured by the different nodes under worst attack 2 ($f = 1$, static workload, 4kB requests).

We run this experiment with a request size ranging from 8B to 4kB. Figure 11 presents the throughput under attack relative to the throughput in the fault-free case, for both the static and dynamic load described earlier. The malicious primary of the master instance decreases its throughput by delaying the requests, down to the limit value such that the throughput ratio observed at the correct nodes is be greater or equal than $\Delta$. Indeed, a lower ratio observed at the correct nodes implies a protocol instance change. We again observe that RBFT is robust: the maximum throughput loss is below 3% when $f = 1$. It is even below when $f = 2$: less than 1%.

As for the first worst attack, we present in Figure 12 the throughput measures that are performed by the monitoring mechanism running on the different nodes. Similarly to the previous attack, we observe that all nodes measure the same throughput. Moreover, we observe that the throughput measured for the master protocol instance is almost similar to the one measured for the backup instance. This explains why RBFT is robust.

*3) Unfair primary:* In this attack, the primary of the master instance is not fair and proposes an ordering on the requests of a given client less frequently than for the other clients. We show that the malicious primary cannot increase the latency of the requests of a single client beyond a small limit. We run an experiment with $f = 1$, 2 clients and requests of 4kB. The maximal acceptable latency, $\Lambda$, is 1.5ms. We also set a high value for $\Omega$, the maximal acceptable difference between the average latency of a client on the different protocol instances. Consequently, we easily observe that the primary of the master instance cannot increase the latency of a client beyond $\Lambda$.

Figure 13 presents the latency of each request for both clients. At the beginning and for 500 requests, the malicious primary, of the master protocol instance, acts normally. The average latency is 0.8ms. Then it proposes an ordering for the requests of the first client less frequently, so that the average latency observed for this client is 1.3ms, during 500 more requests. At request 1000, it increases even more the latency observed for this client. This single request has a latency of 1.6ms. As it is higher than the maximal acceptable latency, the different nodes vote a Protocol Instance Change. As a result, the malicious primary of the master instance is evicted and replaced by a correct replica. This correct replica is fair and provides the same latency for the requests of both clients. Note that in this experiment the throughput monitoring mechanism did not triggered a protocol instance change because the malicious primary of the master instance provided a similar throughput than the throughput of the backup protocol instances.

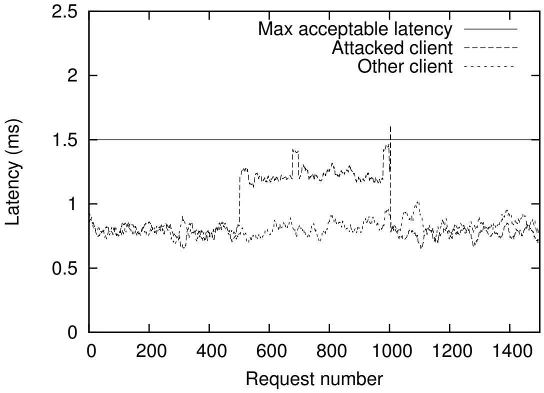


Fig. 13: Ordering latencies for the requests of two clients on the master protocol instance with an unfair primary, which starts to delay the request of one of the clients after 500 requests.

## VIII. Conclusion

In this paper we have demonstrated that the state-of-the-art robust BFT protocols are not effectively robust. We have shown that a malicious primary replica can drastically degrade performance. To tackle the robustness problem, we have proposed a new approach: RBFT, for Redundant Byzantine Fault Tolerance. The key idea in RBFT is to run several instances of a BFT protocol in parallel, and to monitor their performance in order to detect a malicious primary. We have shown that the performance of RBFT in the fault-free case is equivalent to the performance of the state-of-the-art robust BFT protocols. Moreover, we have shown that RBFT is more robust than existing protocols as it bounds the throughput degradation caused by a malicious primary to 3%, even in drastic conditions where an unlimited number of malicious clients collude with the $f$ malicious nodes to harm the system. Not only we found a small performance loss for the case $f = 1$, but this loss is even smaller for $f = 2$. In our future work, we plan to study how RBFT can be extended to deal with closed-loop systems, i.e. systems in which clients wait for replies to their previous requests before issuing new requests.

## Acknowledgments

We are grateful to the anonymous reviewers for the insightful comments that helped improve the paper. We would also like to thank F. Bongiovanni, R. Lachaize, B. Lepers and A. Pace for their helpful feedback on this work. This work was partially funded by the French ANR SocEDA project.

# References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*. ACM, 2005.

[2] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *DSN*, pages 105–114, 2008.

[3] Fatemeh Borran and André Schiper. Brief announcement: a leader-free byzantine consensus algorithm. DISC'09, 2009.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

[5] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *SOSP*, pages 277–290, New York, NY, USA, 2009. ACM.

[6] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[7] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[8] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient middleware for byzantine fault tolerant database replication. EuroSys, 2011.

[9] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *EuroSys*, pages 363–376, 2010.

[10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. USENIX ATC, 2010.

[11] M. Kaptritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. Eve: Execute-verify replication for multi-core servers. In *OSDI 2012*, Oct 2012.

[12] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):1–39, 2009.

[13] Leslie Lamport. Lower bounds for asynchronous consensus, 2004.

[14] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. OSDI, pages 8–8, 2004.

[15] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. NSDI, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[16] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distrib. Syst.*, 21(4):452–465, April 2010.

[17] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS*, pages 135–144, 2009.

[18] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. EuroSys, 2011.