Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary

Giuliana Santos Veronese, Miguel Correia,
Alysson Neves Bessani, Lau Cheuk Lung

DI–FCUL TR–2009–16

June 2009

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Spin One's Wheels?
Byzantine Fault Tolerance with a Spinning Primary

Giuliana Santos Veronese[1], Miguel Correia[1,2], Alysson Neves Bessani[1], Lau Cheuk Lung[3]
[1]Universidade de Lisboa, Faculdade de Ciências, LASIGE – Portugal
[2]Carnegie Mellon University, Information Networking Institute – USA
[3]Departamento de Informática e Estatística, Centro Tecnológico, Universidade Federal de Santa Catarina – Brazil

***Abstract*—Most Byzantine fault-tolerant state machine replication (BFT) algorithms have a primary replica that is in charge of ordering the clients requests. Recently it was shown that this dependence allows a faulty primary to degrade the performance of the system to a small fraction of what the environment allows. In this paper we present Spinning, a novel BFT algorithm that mitigates such performance attacks by changing the primary after every batch of pending requests is accepted for execution. This novel mode of operation deals with those attacks at a much lower cost than previous solutions, maintaining a throughput equal or better to the algorithm that is usually considered to be the baseline in the area, Castro and Liskov's PBFT.**

I. Introduction

Many applications with high security and fault tolerance requirements can benefit from Byzantine fault-tolerant algorithms. These algorithms allow systems to continue to provide a correct service even when some of their components fail, either accidentally (e.g., by crashing) or due to malicious faults (arbitrarily). Systems based on those algorithms are often said to be *intrusion-tolerant* and several have already been presented in the literature: network file systems [1], cooperative backup [2], coordination services [3], certification authorities [4].

Intrusion-tolerant systems are usually built using replication techniques. The idea is that there is a service that is replicated in a set of servers that execute requests from the clients. State machine replication (SMR) is one of these techniques, which allows making any deterministic distributed service fault- or intrusion-tolerant [5]. In this form of replication all (non-faulty) servers have to execute all clients' requests in the same order. Several *leader-based* Byzantine fault-tolerant state machine replication algorithms – that we call *BFT* in the paper – have been presented [1], [2], [6], [7], [8] (SMR algorithms that are not leader-based have also been presented [9]). Among these algorithms, Castro and Liskov's PBFT [1] is often considered to be a baseline in terms of performance, probably because it was the first efficient algorithm in the area and many others derive from it (e.g., [6], [7], [8], [10]).

In BFT algorithms, the servers move through a succession of configurations called views. Each view has a primary server (or leader) that is in charge of defining the order in which the requests are executed by all servers. In all previous BFT algorithms, including PBFT, the primary remains the same as long as no faults are detected. When a subset of the servers suspect that the primary is faulty, they choose another server to be the primary.

This basic scheme has been shown to be vulnerable to *performance attacks* by Amir et al. [8]. More precisely, these authors have shown two attacks that can degrade the performance of PBFT to let it so slow that it is barely usable. In the first attack, *pre-prepare delay*, a faulty server delays the ordering of requests from some of the clients, causing a considerable increase of the latency of those requests and a great reduction of the throughput. PBFT imposes a maximum delay on the execution of requests, but only on the first request of a queue of pending requests, so a faulty primary can process one request at a time, strongly delaying most requests. In the second attack, *timeout manipulation*, faulty servers manage to increase the timeouts used in PBFT, seriously degrading the performance of the system. These attacks also apply to some of the algorithms that derive from PBFT (e.g., [7]).

Amir et al. [8] presented an algorithm, Prime, that tolerates these attacks by adding a pre-ordering phase of 3 communication steps to PBFT. Clement et al. also presented a system, Aardvark [10], that tolerates these attacks by monitoring the performance of the primary and changing the view in case it seems to be performing slowly. Aardvark is also based on PBFT.

This paper presents *Spinning*, an algorithm that modifies the usual form of operation of BFT algorithms: instead of changing the primary when it is suspected of being faulty, it *changes the primary whenever it defines the order of a single batch of requests*. Putting it more simply: in each view the primary orders only one batch. The name of the algorithm comes precisely from the fact that the primary is always changing, i.e., spinning.

Spinning was also designed as a modification of PBFT, just like Prime and Aardvark. Its *normal* operation is similar to PBFT's, with its three communication steps. It has no view change operation, since views are always changing, but it has a *merge* operation, which is in charge of putting together the information from different servers to decide if requests in views that "went wrong" are to be executed or not. A view can go wrong essentially for the same reasons

as a view change can be needed in PBFT: the primary is faulty and does not send (some of) the messages it should, or the network becomes slow and timeouts expire.

This basic operation of *Spinning* has an obvious problem: the primary is always changing so faulty servers go on being the primary, becoming able to impair the average performance of the service. To avoid this problem, we use a mechanism that punishes misbehavior, basically by putting in a blacklist the servers that were the primaries when something went wrong. When a server is in the blacklist, it does not become the primary.

Spinning has two main benefits. The first is that it avoids the above-mentioned performance attacks made by faulty primaries in a very simple and efficient way: by always changing the primary. View changes in *Spinning* do not incur in the cost of running a distributed algorithm with several communication steps; the view is changed automatically after the 3 communication steps executed by the servers, which are essentially the same as in PBFT. Theoretically, rotating the primary after each request can make the system lose less than $\frac{1}{3}$ of its throughput when under attack. This leads to better performance than Prime and Aardvark.

The second benefit is that *Spinning* manages to improve the throughput of PBFT when there are no faulty servers by balancing the load of ordering requests among all (correct) servers. Although ordering requests requires that all servers exchange messages, thus causes load in all of them, most load is in the primary so changing the primary improves the throughput of the algorithm by a factor of 20%. Something similar was recently explored by Mao et al. in the Mencius algorithm [11], but they consider only crash faults, so manage to avoid communication between the servers other than the primary, thus obtaining a much higher improvement of the throughput.

The main contributions of the paper are the following: (1) it presents a novel style of leader-based Byzantine fault-tolerant state machine replication algorithm that changes the primary whenever the current primary defines the order of a batch of requests, instead of only when it is suspected of being faulty; (2) it presents *Spinning*, a BFT algorithm that is less vulnerable to performance degradation attacks caused by a faulty primary, attaining a throughput similar to the baseline algorithm in the area (better in the fault-free case) and better than other algorithms that tolerate these attacks.

II. System Model

The system is composed by a set of n *servers* $\Pi = \{s_0, ..., s_{n-1}\}$ that provide a Byzantine fault-tolerant service to a set of *clients* $C = \{c_0, c_1, ...\}$. Clients and servers are interconnected by a network and communicate only by message passing. We assume that the communication is done using reliable authenticated point-to-point channels, but that these channels may be disconnected and reconnected later, causing sequences of messages to be lost[1]. This communication model is equivalent to PBFT's [1] and other BFT systems. All servers are equipped with a local clock used to compute message timeouts. These clocks are not synchronized so their values can drift.

We assume a partial synchrony system model [12]: in all executions of the system, there is a bound Δ and an instant GST (Global Stabilization Time), so that every message sent by a correct server to another correct server at instant $u >$ GST is received before $u + \Delta$, with Δ and GST unknown. The intuition behind this model is that the system can work asynchronously (with no bounds on delays) most of the time but there are stable periods in which the communication and processing delays are bounded[2]. This assumption is required to ensure the liveness of BFT algorithms (e.g., [1], [6], [7]).

Servers and clients can be *correct* or *faulty*. Correct servers and clients follow the algorithm that they are supposed to execute. We assume that any number of clients can be faulty, but the number of servers that can be faulty is limited to $f = \lfloor (n-1)/3 \rfloor$, thus $n \geq 3f+1$. For simplicity we present the algorithm for the tight case, i.e., with $n = 3f+1$. The failures can be Byzantine or arbitrary, meaning that the processes can deviate arbitrarily from their algorithm, even by colluding with some malicious purpose. For the server to be intrusion-tolerant, they can not share the same vulnerabilities, so they have to be diverse [13].

The authenticity of the messages exchanged by the algorithm is protected with signatures based on public-key cryptography or message authentication codes (MACs) produced with collision-resistant hash functions. Each server and client has a public/private key pair. Each server knows the public keys of all servers and clients, and each client knows the public keys of all servers. The servers use their key pairs to establish shared keys among themselves and use the latter to create and verify MACs. A signature σ_c is said to be *valid* iff it was created with its sender's (c) private key and corresponds to the message. A message digest is said to be *valid* iff it was created with the key shared between its sender and its recipient. We also use hash functions to obtain digests of requests and other data. We make the standard assumptions about cryptography, i.e., that hash functions are collision-resistant and that signatures can not be forged. We assume all public and private keys are distributed before the algorithm is executed.

III. Spinning

In state machine replication, each server in Π is modeled as a functionally identical state machine [5]. Each server maintains a set of *state variables* that are modified by a set

[1]These channels can easily be implemented in practice assuming fair links and using retransmissions, or in a more practical view, using TCP over IPsec or SSL/TLS.

[2]In practice this stable period has to be long enough for the algorithm to terminate, but does not need to be forever.

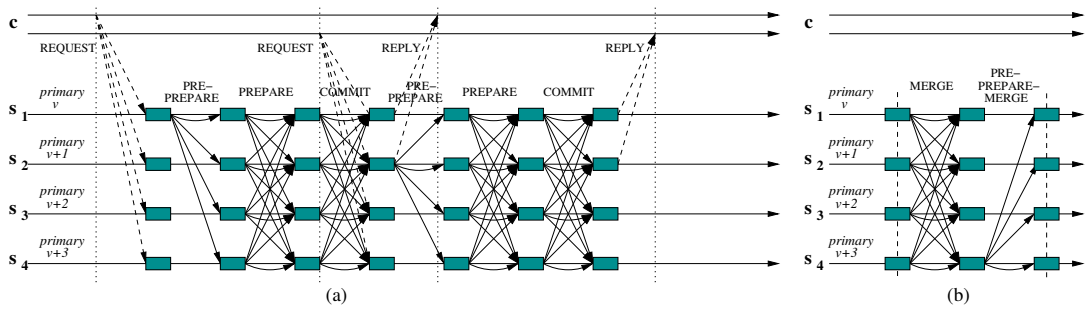

Figure 1. (a) Normal operation, communication pattern between the servers, with the communication with a client for one request superimposed. (b) Merge operation.

of *operations*. Clients of the service issue *requests* with operations through a replication algorithm which ensures that, despite concurrency and failures, servers perform requests in the same order. The properties that the algorithm has to enforce are: (1) all correct servers execute the same requests in the same order (*safety*); (2) all correct clients' requests are eventually executed (*liveness*). This section presents the *Spinning* algorithm.

A. Algorithm overview

In the initial presentation of the algorithm we do several simplifications that we remove later in Section III-F, e.g., that requests are not processed in batches but one at a time. We also do not consider the blacklist mechanism, which is described later in Section III-D.

The servers move through successive configurations called views. Each view has a primary server that changes in a round-robin fashion. The primary is the server $s_p \triangleq v \bmod n$, where v is the current view number. The purpose of the algorithm is to force all correct servers to *execute* the requests in the same order, but only one request is executed per view. Notice that *there are no sequence numbers* in the algorithm. The number of sequence for each request is simply the number of the view in which it is accepted for execution. The primary has the task of defining which request is the one to be executed in the current view.

The normal operation of the algorithm follows essentially the communication pattern of PBFT's normal operation. The primary sends a PRE-PREPARE message with a pending request to all servers. When a correct server receives this PRE-PREPARE message, it validates the message and sends to all servers a PREPARE message. When a correct server receives $2f+1$ PREPARE messages, it sends a COMMIT message. Each correct server that receives $2f+1$ COMMIT messages accepts and executes the request, then increments its view number. Notice that the 3 steps have the same names as the messages sent in each of them: pre-prepare, prepare, commit.

A faulty primary may not send the PRE-PREPARE message, or send it only to some of the servers (but not all). Therefore, if a server has client requests in its buffer to be ordered, it waits a maximum time interval T_{acc} to accept the request of that view. If a server does not receive enough COMMIT messages to accept the request during T_{acc}, then it sends a MERGE message to all servers.

The merge operation plays a role somewhat similar, but different, to the view change in PBFT. The generic objective is the same: to ensure liveness if the primary is faulty. The specific objective is different: not to change of view, but to agree on which requests of the previous views were accepted and have to be executed by all (correct) servers. The main problem is when some of the messages are lost or not sent, and some of the correct servers accept the requests, but other correct servers do not. In that case, the algorithm has to *merge* the information from the different servers to agree on the requests that were accepted and go to the next view.

When a server s_i receives at least $f+1$ MERGE messages, it also sends its MERGE message and its state is changed to merge. When the server that will be primary in the next view receives $2f+1$ MERGE messages, it sends a PRE-PREPARE-MERGE message that carries information enough to make all servers agree about the order of requests in the previous views. When another correct server receives and validates the PRE-PREPARE-MERGE message, it sends a PREPARE message and follows the normal operation of the algorithm.

Details about how a server processes each message of the algorithm are given in the next section and the algorithm is formalized in Appendix A that also gives a proof of correctness. Figure 1 shows the algorithm communication pattern.

B. Algorithm

This section provides a more in depth description of the algorithm. It presents the algorithm in terms of the sequence of operations executed in a view, first in normal operation and later in the merge operation. Each server is always in one of two states that correspond to these two operations: `normal` and `merge`.

Table I serves as a reference to the fields of the messages exchanged.

Normal Operation: In normal operation a request is processed the following way:

1. Client sends a request to all servers. A client c issues a request for the execution of an operation *op* by sending a message $\langle \text{REQUEST}, c, seq, op \rangle_{\sigma_c}$ to all servers. The *seq* field is the request identifier that is used to ensure exactly-once semantics: the servers do not execute a request for a client with a *seq* lower than the last executed of that client to avoid executing the same request twice.

2. Upon server s_i becoming the primary of view v. When the view changes and a server s_i becomes the primary of v it verifies if: (i) it accepted the request from view $v-1$; (ii) it is in the `normal` state; and (iii) if it has at least one client request pending to be ordered. If these conditions are satisfied the primary sends a $\langle \text{PRE-PREPARE}, s_i, v, dm \rangle_{\sigma_{s_i}}$ message to all servers, where s_i is the server identifier, v the view number and *dm* a digest of the request sent by a client.

3. Upon server s_j receiving a PRE-PREPARE *message.* When a server s_j receives a $\langle \text{PRE-PREPARE}, v, s_i, dm \rangle_{\sigma_{s_i}}$ message from s_i, it evaluates if: (i) the signature is valid; (ii) the view number v is equal to the current view number on s_j and the sender is the primary of v; and (iii) it is in the `normal` state. If these conditions are satisfied the message is said to be *valid* and s_j sends a $\langle \text{PREPARE}, v, s_j, dm \rangle_{\sigma_{s_j}}$ message to all the other servers. After s_j sends its PREPARE message, it discards any other PRE-PREPARE messages for view v.

A server can also receive a PRE-PREPARE message with view $v' > v$. In this case, it buffers the message and waits to accept all messages with view less than v'. After that, if the message with view v' is valid, the server sends the PREPARE message of that view. This ensures that, even if the network mixes the order of messages, the server always accepts messages following the order defined by the view number.

4. Upon server s_j receiving PREPARE *messages from at least $2f+1$ servers.* When a server receives $2f+1$ PREPARE messages from different servers (possibly including itself), with valid signatures, the same view number v and the same *dm*, it sends a $\langle \text{COMMIT}, v, s_j \rangle_{\sigma_{s_j}}$ message to all servers.

5. Upon server s_j receiving COMMIT *messages from at least $2f+1$ servers.* When a server receives $2f+1$ COMMIT messages from different servers (possibly including itself), with valid signatures and the same view number v, it accepts and executes the request. After executing the request, the server sends $\langle \text{REPLY}, s, seq, res \rangle_{\sigma_{s_j}}$ to the client that issued the request, where *res* is the result of the operation. The server increments its view number and if it is the primary sends a PRE-PREPARE message. Otherwise, it sets a timer waiting for the request of the new view to be executed.

6. Upon a client receiving matching replies from $f+1$ servers. When the client receives $f+1$ replies $\langle \text{REPLY}, s, seq, res \rangle_{\sigma_s}$ from different servers s with matching results *res*, it accepts the result.

Merge Operation: Servers can not wait indefinitely for messages from the primary because it may be faulty. Therefore, whenever a new view begins and the servers have requests pending to be ordered, every correct server starts a timer, in order to wait at most T_{acc} for the request of that view to be accepted. If that timer expires, the merge operation starts. More precisely, when the timer expires at a server or it receives $f+1$ MERGE messages from $f+1$ other servers (at least one of which correct), it changes its state from `normal` to `merge`. When it receives a PRE-PREPARE-MERGE from the new primary, it sets the state back to `normal`. The objective is to change the primary until one that is not suspected of being faulty is selected.

The main difficulty of this operation is that in a view v the timer may expire in some correct servers but not in others. This may lead some correct servers to accept and execute the request of that view, while others do not. To prevent this situation from breaking the safety property, the messages exchanged in the merge operation take information about the requests accepted in the last views.

If the network is slow, timers can go on expiring successively and merge operations can be executed one after another. However, the partial synchrony model (Section II) together with the timeout configuration (Section III-E) ensure that eventually the system becomes stable enough for the algorithm to change to normal operation.

1. Upon the timer expiring on server s_i. When the timer expires, the server sends a $\langle \text{MERGE}, s_i, v, P \rangle_{\sigma_{s_i}}$ message to all servers. If the server has at least $f+1$ PREPARE messages from different servers with the current view number, then it sets the v field in the message to this view number. Otherwise, the field is set to v_{last} that is the last view for which s_i accepted a request. This mechanism ensures that if the timer expires on other servers, at least $f+1$ servers will initiate the merge with the same view number. The field P contains information about requests from the current and

Label	Meaning
Server messages	
s	server identifier
v	view number
op	request
dm	digest of request
P	set of prepare certificates (each one contains $2f+1$ valid PREPARE messages)
VP	vector with digests of requests ordered by view number (the digests are taken from the P field of MERGE messages)
M	set of $2f+1$ MERGE messages (merge certificate)
Client messages	
c	client identifier
seq	request identifier
op	operation requested to be executed

Table I
LABELS GIVEN TO FIELDS IN MESSAGES

previous views for which s_i received a valid PRE-PREPARE message and at least $2f+1$ valid PREPARE messages. More precisely, P contains one *prepare certificate* for each of these requests. Such a certificate is composed by $2f+1$ valid PREPARE messages corresponding to a certain PRE-PREPARE message.

Correct servers always keep the last n requests accepted and their prepare certificates. P contains all certificates that s_i has for any views greater or equal to $v_{last}-n$, where v_{last} is the last view for which s_i accepted the request. Note that it is necessary to include only the last n accepted requests because a correct server only accepts a request in a view v if s_i has accepted the request ordered in the view $v-1$. After sending a MERGE message, the server changes its state to `merge` and increments the view number.

2. Upon server s_j receiving $f+1$ MERGE messages from different servers. When s_j receives at least $f+1$ MERGE messages for a view v, if v is higher or equal to its current view and it has not sent a MERGE message for this view, it sends $\langle \text{MERGE}, s_j, v, P \rangle_{\sigma_{s_j}}$ to all servers. It changes the state to `merge` and increments the view number. The verification that v is higher or equal to its current view is needed to prevent faulty servers from doing a merge operation of a past view.

3. Upon server s_i becoming the primary of a new view v. When s_i receives $2f+1$ MERGE messages for view $v-1$ and becomes the primary of the new view v (i.e., $s_i = v \bmod n$), it sends a $\langle \text{PRE-PREPARE-MERGE}, s_i, v, VP, M \rangle_{\sigma_{s_i}}$ message to all servers. VP is a vector of digests of requests taken from the P field of the MERGE messages, ordered by view number. Only digests from views v_{min} to v_{max} are included: v_{max} is the highest view number in the prepare certificates from the P fields received in the MERGE messages; $v_{min} = v_{max} - n$. M is a *merge certificate* composed by the $2f+1$ MERGE messages received. This certificate is used by the recipients of the message to verify if the primary computed VP correctly, i.e., if it is *valid*.

4. Upon server s_j receiving a PRE-PREPARE-MERGE *message.* When a server s_j receives a valid message $\langle \text{PRE-PREPARE-MERGE}, s_i, v, VP, M \rangle_{\sigma_{s_i}}$ from s_i it evaluates if: (i) the signature of the message is valid; (ii) the sender is the primary of v; and (iii) VP is valid (using the merge certificate M to do the same computation as the primary).

Consider that the last request accepted by s_j has the view number v_{last} and that the lowest view number in VP is v_{min}. If $v_{last}+1 \geq v_{min}$, the server state is changed to `normal` and s_j sends a $\langle \text{PREPARE}, v, s_j, dm \rangle_{\sigma_{s_j}}$ message to all servers, where dm is the digest of VP. After that, the server follows the steps 4 and 5 of normal operation. Servers use the information about previously executed requests to avoid re-executing clients' requests. Otherwise, if the server have missed some messages before the view v_{min}, it must obtain missing information from another server. The mechanism to bring a server up to date is discussed below.

C. Garbage collection

If a server becomes unable to communicate with the rest for some time because its channels are disconnected, it may miss some of the messages exchanged. Therefore, each server has to store messages in a buffer and retransmit them when necessary.

Each correct server keeps the messages for the last l requests accepted in its buffer. The algorithm does not allow accepting requests out of order, therefore we can use the view number to limit the value of l. l is a system parameter that can be set to n, which represents a full cycle of the algorithm (each server is primary once). Even when there are subsequent merge operations, at least $2f+1$ servers keep in their buffers the last l requests accepted (and all related algorithm messages). These messages are only discarded when $2f+1$ servers accept the next l messages.

If a server s_i missed some messages but all servers discarded them already, the correct servers send to s_i the most recent prepare certificate proving that the system made progress and that it is no longer possible to recover the messages. The solution to this situation is to do a state transfer from the correct servers to s_i.

Messages received for a view higher than the current one (v) are buffered. In order to avoid buffer exhaustion when malicious servers send messages with high view numbers, correct servers discard messages with view higher than $v+n$ when the buffer free space drops below some low water mark L (a system parameter).

The garbage collection does not need PBFT's *checkpoint* mechanism to advance the low and high water marks of the buffer [1]. Servers do not exchange *checkpoint* messages. However state transfers may be needed, as already mentioned. Therefore, servers compute checkpoints to be used for this single purpose. The checkpoints are generated periodically, when a view number is divisible by some constant (e.g. 10). After a server generates a checkpoint it discards all PRE-PREPARE, PREPARE and COMMIT messages with view number less than v from its buffer. It also discards all earlier checkpoints as every server has to store a single checkpoint. When a server that missed some messages needs a state transfer, it requests the checkpoint from other servers. The server accepts a checkpoint state when it receives the same checkpoint reply from at least $f+1$ different servers.

D. Punishing misbehavior

In the basic version of the algorithm described, every server becomes the primary periodically. This is an opportunity for faulty servers to periodically impair the performance of the algorithm, although only during the window of time in which they are the primary. Therefore, in order to punish the misbehavior of faulty primaries we defined the *blacklist* mechanism that is presented in this section.

Normally the algorithm works in *cycles* of n primaries. Each server keeps a blacklist of servers. When a server is included in the blacklist it does not become the primary. When a correct server s accepts a request and increments the view number, it verifies if the primary of the next view is not in its blacklist. If it is, the server increments the view number again until it finds a new primary that is not in the list. Notice that a server can be wrongly suspected of being faulty due to long network delays. Therefore, servers in the blacklist are not excluded from the algorithm in any other way than not being the primary.

The blacklist has to be updated by all correct servers in a coordinated way, so all servers have to apply the same criteria in the same order to insert and remove servers from the list. The basic mechanism is the following. When a server receives a valid PRE-PREPARE-MERGE message in view v, this means that $2f+1$ servers agreed that some problem occurred in view $v-1$. All correct servers that receive the PRE-PREPARE-MERGE message include the primary of $v-1$ in the blacklist. The size of the blacklist is f, since it is the maximum number of faulty servers. If the list is full and a server has to be inserted, the oldest one in the list is removed (i.e., FIFO policy).

The basic scheme has to be modified in the following way. If there is a merge operation and the new primary is faulty, it may not send valid PRE-PREPARE-MERGE messages to all correct servers, leading to different blacklists. However, if the new primary does this attack, a new merge operation must be executed. Therefore, the servers only keep in the blacklist the primary's identifier that caused the last merge operation. When a server understands that a merge operation started after another one, it *replaces* –not adds– the last server inserted in the blacklist by the server that caused the new merge operation. The idea is that all merge operations that occurred before the last one are ignored, simply because it is not possible to know if all correct servers received all PRE-PREPARE-MERGE messages. Notice that there is no risk of confusion about which is the next primary because the server that is removed from the blacklist is always the one before the last one, which can never be the next primary.

E. Timeout configuration

T_{acc} is the maximum time interval for a message to be accepted in a view. This value is not constant. It starts with an initial value defined by the system parameter T_{start} and is multiplied by two whenever there is a merge operation. Just like in PBFT, the objective is to ensure the liveness of the system when there are long communication delays, i.e., to ensure that eventually $T_{start} > \Delta$.

To avoid that a malicious primary forces this timeout to be high and the progress of the system slow, each server divides by two the value of T_{acc} whenever it detects that the system is stable (only if $T_{acc} > T_{start}$). In order to detect if the system is stable, all correct servers calculate the time T_{avg} that a request takes to be accepted. If after r cycles a server verifies that T_{avg} is lower than $T_{acc}/2$, the server resets the value of T_{acc} to $T_{acc}/2$ (r is a system parameter). After r cycles if all requests have been accepted it is possible to infer that the system is stable. Due to delays in the network, some servers can reduce their timeouts while others do not. However, this is not a problem because the partial synchrony assumption guarantees that the system stabilizes and all correct servers eventually reset T_{acc}.

F. Optimizations

This section presents several optimizations to the basic *Spinning* algorithm.

Batches of requests: The basic algorithm only "orders" one request per view or, more precisely, only agrees on the execution of one request per view. The algorithm can be trivially modified to agree on the execution of a batch (i.e., a set) of requests per view. The difference is that the primary has to send in the PRE-PREPARE message the digests of the pending requests, instead of only one. After being accepted, the requests are executed in some deterministic order.

Piggybacking PRE-PREPARE *messages:* The PRE-PREPARE messages can be sent together with COMMIT messages, reducing the cycle of the communication among the servers from 3 communication steps to 2. When a server sends a COMMIT message in view v, if it is the primary of the next view ($v+1$), it appends a PRE-PREPARE message to the COMMIT messages. Therefore, the next primary sends a $\langle\langle \text{COMMIT}, v, s_j\rangle, \langle \text{PRE-PREPARE}, v+1, s_j, dm'\rangle\rangle_{\sigma_{s_j}}$ message where dm' is the digest of a client's request that has not yet been accepted. When a correct server receives such a message, it stores the PRE-PREPARE and only sends the corresponding PREPARE message when the view changes, i.e., when the request from view v is accepted.

Using MAC vectors: Signatures based on public-key cryptography are known to be much slower to create and verify than MACs. PBFT uses vectors of MACs instead of signatures to improve the performance of BFT algorithms [1]. The idea is to authenticate a message with a vector of MACs, called an *authenticator*, each one obtained with a secret key shared between the sender and each of the recipients. However, authenticators are less powerful than signatures because a faulty sender can falsify a subset of the MACs, while signatures are all or nothing: either valid or invalid.

In order to improve the performance of the *Spinning* algorithm, the messages sent in the algorithm can be authenticated using authenticators instead of signatures, like in PBFT. In order to prevent a request with a partially valid vector of MACs to be accepted only by some of the correct servers, we use a mechanism similar to the one described in [1]: a server s_i authenticates a message if either the MAC for s_i is correct or s_i has $2f$ PREPARE messages with the same request's digest.

Parallel executions: The basic *Spinning* algorithm accepts and executes a single batch of requests per view. However, this basic functioning can be generalized to run a pre-configured maximum number of parallel agreements, which we call the *window size* (w). When a request is received during a view, if the primary still did not start w agreements in the current view (i.e., sent PRE-PREPARE messages), it starts a new one. Otherwise, the request is kept for the next view. The messages exchanged between the servers take an order number local to the view (from 1 to w). This mechanism works similarly to the way PBFT does batching of requests.

IV. Evaluation

In this section we assess the advantages of *Spinning* comparing it with related protocols both experimentally and analytically. Our evaluation makes three points. First, we show experimentally that rotating the primary does not impair the performance of the system, but instead brings some throughput gains in fault-free executions (Section IV-A1). Second, we show that rotating the primary is a simple and effective strategy to make a BFT replication algorithm tolerate performance attacks from malicious servers (Section IV-A2). Finally, we show analytically that *Spinning* presents important advantages when compared with other solutions to deal with performance attacks (Section IV-B).

A. Experimental Evaluation

Protocol Implementations: We implemented our prototype of *Spinning* in Java. We chose Java because it is considered a much more secure language than C/C++ due to the existence of features like memory protection, strong typing and access control. These features can make a BFT implementation much more dependable. Java has also the advantage of improving the system portability (hardware and operating system) making it easier to deploy in different environments, a crutial requirement for BFT systems due to the need of diversity to justify the fault independence assumption.

PBFT [1] is often considered to be the baseline for BFT algorithms, so we were interested in comparing *Spinning* with the C++ implementation available at http://www.pmg.lcs.mit.edu/bft/. We also implemented a version of PBFT's normal case operation in Java (JPBFT) in order to better compare *Spinning* with a "fixed-leader" algorithm based on the same codebase.

The prototypes (*Spinning* and JPBFT) were implemented for scalability, i.e., for delivering a throughput as high as possible when receiving requests from a large number of clients. To achieve this goal, we built a scalable event-driven I/O architecture and implemented an adaptive batching algorithm and window congestion control similar to the one used in PBFT (the algorithm can run a pre-configured maximum number of parallel agreements, the *window size*; requests received when there are no slots for running agreements are batched in the next agreement possible). Additionally, we used recent Java features such as non-blocking I/O and the concurrent API (packages `java.nio` and `java.util.concurrent`). Finally, we used TCP sockets for communication and message authentication is done using HMACs based on SHA1.

Setup and methodology: Unless where noted, we consider a setup that can tolerate one faulty server ($f = 1$), with $n = 4$ servers. We executed from 0 to 120 logical clients distributed through 6 machines. The servers and clients machines were 2.8 GHz Pentium-4 PCs with 2 GBs RAM running Sun JDK 1.6 on top of Linux 2.6.18 connected by a Dell gigabit switch. In all experiments in which Java implementations were used, we enabled the Just-In-Time (JIT) compiler and run a warm-up phase to load and verify all classes, transforming the bytecodes into native code.

We measured the *latency* of the algorithms using a simple service with no state that executes null operations, by varying the requests size between 0 and 4Kbytes. The latency was measured at the client by reading the local clock immediately before the request was sent, then immediately after the response was accepted and subtracting the former from the latter. *Throughput* results were obtained calling also null operations using requests and responses with 0 bytes. These requests were sent by a variable number of logical clients in each experiment (1-120). Each client sent operations periodically (without waiting for replies), in order to obtain the maximum possible throughput. Each experiment ran for 100,000 client operations to allow performance to stabilize, before recording data for the following 100,000 operations.

1) Fault-free Executions: The first part of our experiments aims to compare the performance of *Spinning* with PBFT when there are no faults and asynchrony on the system. Table II present the results for latency.

Req/Res	PBFT	JPBFT	Spinning
0/0	0.4	1.8	1.3
4K/0	0.6	2.2	1.7
0/4K	0.8	2.5	2.1

Table II
Latency results (in ms) varying request and response size for PBFT, JPBFT and Spinning.

In this experiment, PBFT has shown the best performance of all algorithms/implementations, followed by *Spinning* and JPBFT. This experiment shows clearly that our Java implementation runs an agreement much slower than PBFT, although they run the same number of communication steps. One of the possible reasons for this can be the overhead of our event-driven socket management layer that maintains several queues and event listeners to deal smoothly with a high number of connections. *Spinning* is faster than JPBFT since it implements the *tentative execution* optimization

originally proposed and implemented in PBFT [1], which executes an operation after receiving $2f+1$ PREPARE messages, i.e., one step earlier.

The second part of the our experiments had the objective of measuring the peak throughput of the algorithms with different loads. The results are presented in Figure 2.

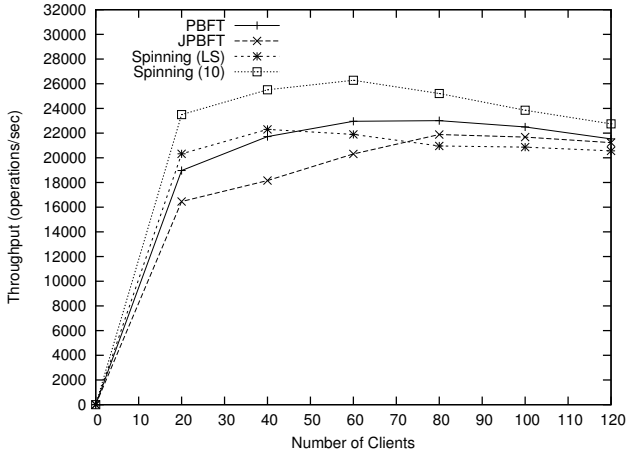

Figure 2. Peak throughput for 0/0 operations for PBFT, JPBFT, Spinning lock-step and Spinning with window size equals 10.

For our experiments, we defined the window size of PBFT as 1, which is the optimal value for fast networks, and the window size of JPBFT as 10. These values are the optimal ones found on our network. We also executed two versions of *Spinning*, one in which at most one consensus is initiated by each leader (Lock Step - LS) and another in which each server initiates at most 10 consensuses (possibly in parallel), which is equivalent to PBFT/JPBFT with window size of 10. The figure shows that PBFT, JPBFT and *Spinning* (LS) have approximately the same peak throughput (22000 to 23000 op/sec). *Spinning* (10), on the other hand, has a throughput 14% better than PBFT and 20% better than JPBFT.

This improvement can be explained by the better load balancing between the servers provided by the *Spinning* algorithm. In PBFT, the throughput of the system is constrained by the amount of messages per batch processed by the leader, which is $5n$. Other servers process only $4n+1$ messages per batch. If we consider this asymmetry as $\frac{5n}{4n+1} \approx 1.2$, this means that the leader executes 20% more work than other servers. This value corresponds to the throughput improvement we observed from JPBFT to *Spinning* (10).

2) BFT Under Attack: To assess the benefits of *Spinning* under performance attacks when compared with PBFT, we run some experiments in which we designated one of the servers as the faulty one (the primary for (J)PBFT) and this server waits an *attack delay* d_{attack} before sending PRE-PREPARE messages. We evaluated the latency and throughput of the algorithms with d_{attack} ranging from 0 to 100 ms.

Table III reports the latency values for the algorithms with one faulty process executing the performance attack, without expiring any timeout.

d_{attack}	PBFT	JPBFT	Spinning (LS)	Spinning (10)
0	0.4	1.8	1.3	1.3
1	1.1	3	3.4	4.2
10	16	13	4.2	5.8
100	103	103	19	22

Table III
LATENCY OF 0/0 OPERATIONS (IN MS) FOR PBFT, JPBFT, SPINNING (LS) AND SPINNING (10) UNDER DIFFERENT LEVELS OF ATTACKS (d_{attack} OF 1, 10 AND 100 MS).

In this table it can be seen that the operation latencies of both PBFT and JPBFT are directly proportional to the attack delay injected by the malicious leader. However, this is not the case for *Spinning*. Assuming that the delay of a BFT algorithm execution with $d_{attack} = 0$ is c, the average delay of PBFT/JPBFT under attack would be approximately $d_{attack} + c$, while for *Spinning* it is the mean between the latency of the operation for all n processes, which would be approximately $\frac{fd_{attack}+(3f+1)c}{3f+1} = \frac{f}{3f+1}d_{attack} + c$. This means that the attack delay will always be diluted by a factor of $\frac{f}{3f+1}$, which is $\frac{1}{4}$ for $f = 1$ (our setup). This corresponds approximately to the ratio we observed between the latency of *Spinning* and JPBFT.

Our final experiment evaluates the peak throughput of the algorithms with different d_{attack} values. Figure 3 reports the observed values.

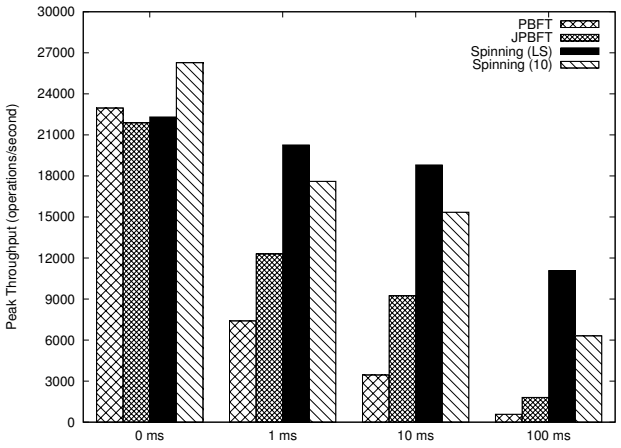

Figure 3. Peak throughput for 0/0 operations for PBFT, JPBFT, Spinning (LS) and Spinning (10) under different levels of attacks (d_{attack} of 1, 10 and 100 ms).

Based on the results of this figure we can make three observations regarding the behavior of the algorithms in face of performance attacks: *(1.)* PBFT suffers more than JPBFT; *(2.) Spinning* (LS) is more resilient than *Spinning* (10); *(3.) Spinning* is more resilient than PBFT.

Observations *(1.)* and *(2.)* show that algorithms that execute more consensus instances are more affected by these attacks when compared with algorithms that execute less instances, which take more time to execute and order more messages in their batches. This is not a surprise since the attack affects each consensus initiated by faulty servers, so more consensuses lead to more attacks. The conclusion here is that, while *Spinning* (10) provides better throughput when there are no attacks in the system, *Spinning* (LS) is more resilient to performance attacks, and thus there is a tradeoff here relating the window size of the algorithms and performance under this type of attack.

Observation *(3.)* shows that our main motivation for developing *Spinning* actually holds in practice: changing the leader periodically makes a BFT algorithm more resilient to performance attacks. Theoretically, the ratio between the throughput of PBFT and *Spinning* when under attack should be approximately $\frac{f}{3f+1}$, which would be $\frac{1}{4}$ for $f = 1$. In practice, as seen in Figure 3, this does not happen since the adaptive batching algorithm employed in the algorithms dilutes the throughput loss when the attack is not so severe. Take for example, JPBFT and *Spinning* (LS), when $d_{attack} = 1$ and $d_{attack} = 10$, the throughput of JPBFT is 40% and 50% worse than *Spinning*, respectively, instead of the expected 75%. However, when $d_{attack} = 100$ ms, JPBFT is 83% worse than *Spinning* (instead of 75%), which means that after some point, batching does not help.

B. Spinning vs. Related Solutions

In this section we compare *Spinning* with two very recent algorithms designed to be resilient in face of performance attacks: Prime [8] and Aardvark [10]. Since the implementations of these algorithms are not available, we do an analytical comparison based on their theoretical properties.

The Prime replication algorithm introduces a pre-order phase with three communication steps before the global-order (which is based on PBFT) that, together with the constant monitoring of the performance of the primary, make the system able to detect several performance attacks and change the leader when it degrades the performance of the system. One important advantage of *Spinning* when compared with Prime is that in our algorithm the primary of a view always processes less messages than Prime, which means less network I/O and, more important, less cryptographic operations.

Considering b as the number of messages in a batch, in *Spinning* the primary processes $2 + \frac{8f}{b}$ messages per client request while in Prime it processes $2 + 9f + \frac{14f}{b}$. Figure 4 ilustrastes these costs for different values of f and b. The figure shows that even with batching, the processing costs of Prime are much higher than the costs of *Spinning*, and thus, the expected throughput of the former should be much lower than the later. Notice also that this evaluation does not consider that Prime uses public-key signatures while *Spinning* does not.

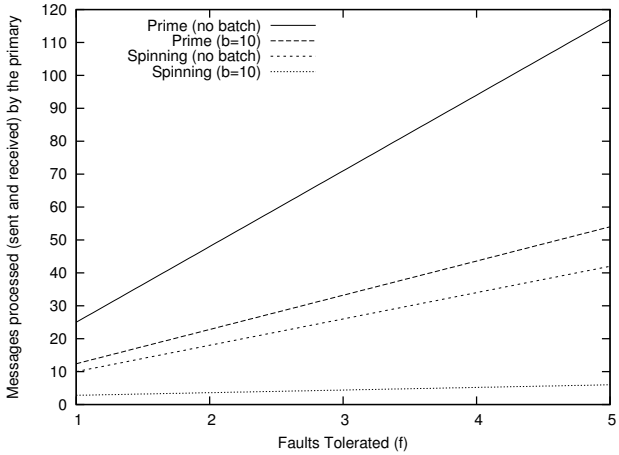

Figure 4. Number of messages processed by the primary per client request in *Spinning* and Prime for several values of f and b.

Aardvark is a BFT library developed concurrently with this work in which a set of engineering principles are applied to PBFT to make it more resilient against several kinds of attacks from clients and servers. One of the attacks addressed by Aardvark is the pre-prepare delay injected by a malicious primary. This attack is mitigated through constant monitoring of the throughput sustained during a view plus the periodic change of primary through the execution of PBFT's view change operation.

Theorem 3 of the Aardvark paper [10] states that under certain conditions (e.g., timely network, correct clients), the ratio between the throughput of the system with a malicious primary and the fault-free throughput is bounded by $\frac{t_{grace}}{2f\,timeout+t_{grace}} \frac{2f+1}{3f+1}$, being t_{grace} the minimum amount of time that a correct server stays as a primary and *timeout* the timeout used to trigger view changes (e.g., in case of a malicious primary).

For *Spinning*, under the same conditions, this ratio can be calculated in the following way: if in a fault-free execution, a stable *Spinning* execution sustains a throughput of k operations per second, when there are f faulty servers executing performance attacks (delaying messages less than *timeout*), the same k operations would take at most $1 + f\,timeout$ seconds. Consequently, the ratio between the throughput of the system on fault-free and faulty executions is $\frac{1}{1+f\,timeout}$.

Figure 5 shows the ratio for both *Spinning* and Aardvark considering timeout values of 40 and 100 ms (which ultimately define the maximum delay for a performance attack). For Aardvark we consider $t_{grace} = 5$ s, as described in [10]. The figure shows that *Spinning* is *potentially* more resilient to performance attacks than Aardvark, and the difference between the ratios for the two protocols only increases as f and *timeout* increase.

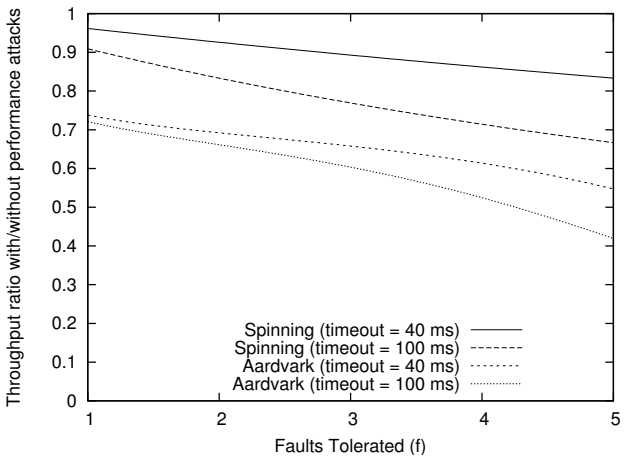

Figure 5. Calculated ratio between the throughput with and without performance attacks for *Spinning* and Aardvark for different different values of f and *timeout*.

The analysis presented in this section shows some evidence that systematically rotating the primary after each request makes *Spinning* more efficient and resilient than other recent BFT systems that use complex mechanisms to ensure acceptable throughput when the primary is faulty.

V. Related Work

The idea of intrusion-tolerant (or Byzantine fault-tolerant) systems appeared in seminal works by Pease et al. [14] and Fraga and Powell [15]. However, the notion started raising more interest much later with works such as PBFT [1], and with the OASIS program and the MAFTIA project.

Many Byzantine fault-tolerant algorithms rely on a primary/leader server that tries to impose a decision; when this primary fails a new one substitutes it. Some examples of leader-based Byzantine fault-tolerant algorithms can be found in [1], [2], [6], [7]. Atomic multicast algorithms that can be used as the main building block of leader-based BFT algorithms have also been proposed [16], [17].

These current leader-based BFT algorithms share the same vulnerability: a faulty primary can slow the system down undetected, until its performance is a fraction of what the conditions allow. Amir et al. [8] brought to light this vulnerability and proposed a bounded delay property to complement the liveness property, requiring the primary to impose a timely decision in order not to be replaced. They also proposed the Prime algorithm to solve the problem, but as shown in Section IV-B *Spinning* solves it more efficiently.

Even more recently, Clement et al. proposed the Aardvark algorithm that modifies PBFT in order to protect it from attacks against performance [10]. Aardvark includes a set of mechanisms that also solve other problems, but the solution to prevent faulty servers from delaying the service is less efficient than *Spinning*, as shown in Section IV-B. Aardvark also changes of primary whenever the primary seems to be performing slowly, but it does this change by running a view change operation, while *Spinning* is constantly changing the primary without the need of executing a distributed algorithm to do it.

There are several consensus algorithms that are based on a *rotating coordinator*, i.e., that change of process that imposes the decision until the algorithm manages to reach a final decision (e.g., [12]). However, these algorithms do one consensus, while *Spinning* does a sequence of consensuses. Furthermore, these algorithms rotate the coordinator with the single purpose of skipping faulty coordinators, while *Spinning* rotates the primary with the purpose of tolerating performance degradation attacks made by faulty primaries and for load balancing.

To the best of our knowledge the only Byzantine fault-tolerant algorithm that rotates the leader without a timeout expiring or the primary misbehaving is BAR-B [2]. However, BAR-B does it differently and with other purposes. BAR-B is an algorithm to implement cooperative systems that considers three kinds of nodes: Byzantine (what we call faulty), altruistic (what we call correct) and rational. Rational nodes participate in the system to gain some benefit and can depart from the algorithm to increase their benefits. BAR-B rotates the leader to guarantee that every node has the opportunity to submit proposals to the system. Each primary starts a sequence of 6 communication steps. The nodes follow a pattern of communication similar to PBFT but have additional steps because the primary does terminating reliable broadcast instead of consensus. Besides needing more steps, no steps are run in parallel, unlike *Spinning*.

Very recently, Mao et al. presented Mencius, an algorithm for efficient state machine replication in WANs [11]. Mencius also changes the primary for each consensus instance, just like *Spinning*, but has several important differences. The main one is that Mencius only tolerates crash faults, not Byzantine faults, so it has a completely different purpose, as it assumes that the primary can not be malicious. It balances the load among the servers and reduces the communication delays by making each client communicate only with the server that is closest to it. Mencius achieves great performance benefits because with crash faults servers other than the primary do not have to communicate directly among them, something that is not true with Byzantine faults.

VI. Conclusion

The paper presents *Spinning*, a novel Byzantine fault-tolerant state machine replication algorithm that tolerates performance attacks by changing the primary whenever a batch of pending requests is accepted for execution. This way of tolerating these attacks is much simpler and more efficient that other solutions in the literature (Prime, Aardvark). This novel mode of operation also does some load balancing among the servers, allowing an improvement of PBFT's throughput in the fault-free case.

Acknowledgments: We warmly thank Yair Amir, Jonathan Kirsch and Hans Reiser for discussions on the paper that greatly assisted us in improving it. This work was partially supported by the Alban scholarship E05D057126BR and by the FCT through the Multiannual and the CMU-Portugal Programmes.

REFERENCES

[1] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, Nov. 2002.

[2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth, "BAR fault tolerance for cooperative services," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Oct. 2005.

[3] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "DepSpace: a Byzantine fault-tolerant coordination service," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference*, Apr. 2008.

[4] L. Zhou, F. Schneider, and R. van Renesse, "COCA: A secure distributed on-line certification authority," *ACM Transactions on Computer Systems*, vol. 20, no. 4, Nov. 2002.

[5] F. B. Schneider, "Implementing faul-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990.

[6] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementations*, Nov. 2006.

[7] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[8] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, Jun. 2008.

[9] M. Correia, N. F. Neves, and P. Verissimo, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, Oct. 2004.

[10] 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 & Implementation*, Apr. 2009.

[11] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proceedings of the 8th USENIX Symposium on Operating systems Design and Implementation*, Dec. 2008.

[12] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, Apr. 1988.

[13] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia, "How practical are intrusion-tolerant distributed systems?" Department of Informatics, University of Lisbon, DI-FCUL TR 06–15, Sep. 2006.

[14] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[15] J. S. Fraga and D. Powell, "A fault- and intrusion-tolerant file system," in *Proceedings of the 3rd International Conference on Computer Security*, Aug. 1985, pp. 203–218.

[16] H. Ramasamy and C. Cachin, "Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast," in *Proceedings of the 9th International Conference on Principles of Distributed Systems*, ser. Lecture Notes in Computer Science. Springer-Verlag, Dec. 2006, vol. 3974, pp. 88–102.

[17] J. P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

APPENDIX

This appendix sketches a proof of correctness of the *Spinning* algorithm. We have to prove that the *safety* property is always satisfied (i.e., that all correct servers execute the same requests in the same order) and the same for *liveness* (i.e., that all clients' requests are eventually executed).

Before the proofs, we present pseudo-code of the algorithm (Algorithm 1). For simplicity we omit details about garbage collection and timeout configuration. We also make the simplifications mentioned in Section III-A: requests are processed one at a time (no batches); $n = 3f + 1$; signatures are used instead of authenticators.

The format of the pseudo-code is a common one so we do not introduce the notation in detail. We use an operator '.' (dot) to refer to elements of messages and rather informally from sets (e.g., in line 25 where the vector *VP* is filled with digests from the *P* field of MERGE messages). Table IV presents the meaning of structures and variables, and Table V shows the auxiliary functions used by the algorithm. Recall that Table I summarized the labels of fields of the messages.

Name	Meaning
n	the number of servers
f	the number of servers that can be faulty
my_v	the view number on server s
v_{min}	the lowest view number in M
v_{max}	the highest view number in M
v_{last}	the view number of the last request accepted by s
m_{last}	the view number of the last valid PRE-PREPARE-MERGE message received by s
blacklist	a set of f faulty primaries
unordered	a set of clients' requests
pending	a set of PRE-PREPARE messages with view number $> my_v$
processing	a set of PRE-PREPARE/PRE-PREPARE-MERGE messages with view number $\leq my_v$
my_P	a set of PREPARE certificates
state	the server state (`normal` or `merge`)
isPrePrepared	indicates if s has sent its PRE-PREPARE message (boolean)
isPrepared	indicates if s has sent its PREPARE message (boolean)

Table IV
VARIABLES AND STRUCTURES AT A SERVER s

Safety. The proof that *Spinning* satisfies the safety property is the following.

Name	Functionality
exec(*req*)	executes a request
oldestRequest(*unordered*)	returns the oldest request in the *unordered* set, if *unordered* is empty, null is returned
size(*VP*)	returns the size of the vector *P*
stateTransfer()	brings the server's state up to date
startTimer()	starts the timer
stopTimer()	stops the timer
restartTimer()	restarts the timer
D(m)	calculates the digest of a message *m*
insertHead(*s*)	inserts the server *s* in the first position of the blacklist. If the blacklist's size is equal *f* removes the element of the last position before to insert *s*
replaceHead(*s*)	replace the first element of the blacklist for *s*

Table V
AUXILIARY FUNCTIONS USED IN THE ALGORITHM

Lemma 1 *In normal operation, if a correct server executes an operation o in view v, no correct server will execute $o' \neq o$ in view v.*

Proof: The proof is by contradiction. Assume the contrary: two correct servers s and s' execute respectively o and o' in view v in normal operation (or `normal` state), i.e., in line 46. A correct server only executes that line if it receives $2f+1$ COMMIT messages from different servers (line 41) and a correct server only sends a COMMIT message if it receives $2f+1$ PREPARE messages with the same message digest dm (lines 32-37). Therefore, for s and s' to execute o and o' there must be $2f+1$ PREPARE messages carrying $\langle v, D(o)\rangle$ and another $2f+1$ carrying $\langle v, D(o')\rangle$, which gives a total of $2\times(2f+1) = 4f+2$. This is clearly impossible because there are only $3f+1$ servers and at most f of them are faulty and may send two different PREPARE messages ($3f+1+f < 4f+2$). A contradiction. ■*Lemma 1*

Lemma 2 *If the timers expire in $f+1$ correct servers, eventually all correct servers enter the* `merge` *state.*

Proof: When the timer expires in a server s_j in view v, it enters the `merge` state and sends a message $\langle \text{MERGE}, s_j, v, P\rangle_{\sigma_{s_j}}$ to all servers (lines 89-100). If that happens in $f+1$ correct servers, eventually all correct servers receive the $f+1$ MERGE messages and enter the `merge` state in line 83, if they did not enter that state before in lines 89 or 100. ■*Lemma 2*

Lemma 3 *If all correct servers are in normal operation in view v, a correct server s executes an operation o but the timers expire in other $f+1$ correct servers, all correct servers eventually execute o.*

Proof: For s to execute o it must have received $2f+1$ COMMIT messages (lines 41-46), at least $f+1$ sent by correct servers. For a correct server, say s', to send a COMMIT message it must have received $2f+1$ PREPARE messages (lines 32-37). If the timer of s' expires, s' inserts these $2f+1$ PREPARE messages in my_P (line 91). The timers expire in $f+1$ correct servers, therefore $f+1$ correct servers enter the `merge` state (lines 89-100), therefore all correct servers eventually enter the `merge` state (Lemma 2) and send MERGE messages to all servers (lines 82-88 or 89-100). There are two cases to consider:

1) *The primary of view $v+1$ sends the* PRE-PREPARE-MERGE *message following the algorithm (lines 15-26):* that message contains a vector *VP* that must contain the digest of o because: (a) that vector is filled using the data from $2f+1$ MERGE messages (lines 20-26); (b) at least $f+1$ correct servers insert $2f+1$ PREPARE messages with o in my_P (like s' above); and (c) the intersection of the quorum of $2f+1$ that sent the MERGE messages of (a) and the $f+1$ correct servers of (b) contains at least one correct server ($(2f+1)+(f+1) = n+1$). *VP* contains the digest of o, so o is executed when the request of view $v+1$ is accepted (lines 41-46, 49-52).
2) *The primary of view $v+1$ is faulty and does not send the* PRE-PREPARE-MERGE *message, or the communication is slow and the timers expire:* in this case the timers expire in all (correct) servers, they increment the view number and we have again to consider the same two cases, 1 and 2.

Eventually the communication will not be slow (partial synchrony model, Section II) and the primary will not be faulty (it changes for every view and only f are faulty), so the system will fall in case 1. ■*Lemma 3*

Lemma 4 *After a merge operation, eventually $2f+1$ servers have the same blacklist.*

Proof: When a correct server receives a valid PRE-PREPARE-MERGE message (line 65) it keeps the view number of this message in the m_{last} variable (line 79). When a correct server receives a valid PRE-PREPARE-MERGE message it verifies if this message succeeds a normal case operation or if this message is a result of successive merge operations. To effect this verification it compares the value kept in m_{last} with the view number of the last request executed kept in v_{last} (line 75). If $m_{last} < v_{last}$ then the received PRE-PREPARE-MERGE message succeeds a normal case operation and the server *inserts* the primary of $v-1$ in the *head* of the blacklist (line 76). If $m_{last} \geq v_{last}$ then the received PRE-PREPARE-MERGE message succeeds other merge operation and the server *replaces* the *head* of the blacklist for $v-1$ (line 78).

Assuming two subsets of servers Q and R, where $|Q| < 2f+1$ and $|R| > f$ servers in order to proof the algorithm correctness we have to consider two cases:

- Q servers accept the PRE-PREPARE-MERGE message for $v-1$ (lines 49-52), the timer expires in R servers (lines 89-100) and all correct servers enter the merge state. When the Q servers receive the next PRE-PREPARE-MERGE message with view number v they replace the first position in the blacklist for $v-1$ because the lowest possible value of m_{last} is $v-1$ and the value of v_{last} is $v-1$ (lines 75 and 78). All servers in R have received PRE-PREPARE-MERGE with $v-1$ that was accepted by Q therefore, when they received the next PRE-PREPARE-MERGE message they also replace the first position in the blacklist for $v-1$. Consequently, all correct servers in Q and R converge to the same blacklist.
- There were successive merge operations the Q servers have lost PRE-PREPARE-MERGE messages and R servers have received them but as Q servers did not send PREPARE messages the timer expires on R servers and there are no accepted PRE-PREPARE-MERGE messages. When a correct primary of view v sends a valid PRE-PREPARE-MERGE message that is received by Q and R servers, the Q servers insert the primary of $v-1$ in the head of their blacklists. Previously, the R servers have received PRE-PREPARE-MERGE messages therefore they replace the first position in the blacklist for $v-1$. Consequently, all correct servers in Q and R converge to the same blacklist.

■*Lemma* 4

Theorem 1 *Let s be the correct server that executed more operations of all correct servers up to a certain instant. If s executed the sequence of operations $S = \langle o_1, ...o_i \rangle$, then all other correct servers executed this same sequence of operations or a prefix of it (Safety).*

Proof: Let $prefix(S,k)$ be a function that gets the prefix of sequence S containing the first k operations, with $prefix(S,0)$ being the empty sequence. Let '.' be an operator that concatenates sequences.

Assume that the theorem is false, i.e., that there is a correct server s' that executed some sequence of operations S' that is not a prefix of S. More formally, assume that $prefix(S',j) = prefix(S,j-1).\langle o'_j \rangle$ and $prefix(S',j-1).\langle o_j \rangle = prefix(S,j)$, with $o'_j \neq o_j$. In this case o'_j is the j-th operation executed in s' and o_j is the j-th operation executed in s. Assume that o_j was executed in view v by s and o'_j was executed in view v' by s'. We have to consider two cases:

1) $v = v'$*:* in this case $o_j = o'_j$ due to Lemma 1.
2) $v' > v$*:* without lack of generality this case assumes that s executes o in v but the timers expire in other $f+1$ correct servers, including s'. This is the case considered in Lemma 3 that shows that o is eventually executed by s'. An inspection of the algorithm shows that: (1) s' does not execute any operation in view v; and (2) s' does not execute any operation between view v and v'. Therefore, the j-th operation executed in s' is $o' = o$.

■*Theorem* 1

Liveness. Next we present the proof of liveness of the *Spinning* algorithm. We say that a request issued by a client c *completes* when c receives the same response for the operation from at least $f+1$ different servers. We define a *stable view* as a view in which the primary is correct and no timeouts expire at correct servers. In this proof we consider the general case in which in each view a batch of requests is agreed upon and executed.

Lemma 5 *During a stable view, an operation requested by a correct client completes.*

Proof: The client is correct so it sends the REQUEST message with the operation o to all servers. The primary is correct (by definition of stable view) so it sends o in the PRE-PREPARE message (lines 15-19), all correct servers send PREPARE and COMMIT messages with o (respectively, lines 27-31 and 32-37), so all correct servers eventually execute o (lines 41-48). After executing o each correct server sends a REPLY message to the client (line 47). The client receives at least the REPLY messages from $2f+1$ correct servers, so the operation completes. ■*Lemma* 5

Theorem 2 *An operation requested by a correct client eventually completes (Liveness).*

Proof: The proof comes from the previous lemmas. We have to consider 4 cases:

1) In stable views operations requested by correct clients eventually complete (Lemma 5).
2) If all correct servers are in normal operation in a view, a correct server executes an operation o but the timers expire in other $f+1$ correct servers, all correct servers eventually execute o (Lemma 3), all correct servers send a REPLY message to the client (line 47) and o completes.
3) If all correct servers are in normal operation in a view, a correct server s executes an operation o but the timers expire in a quorum Q of *less than* $f+1$ correct servers, there are 4 possibilities:
   a) If the number of correct servers in Q plus the number of faulty servers are more than f and the faulty servers also send (valid) MERGE messages, we fall in case 2 above.
   b) If the number of correct servers not in Q are more than $2f$, we fall in case 1 above.
   c) If the number of correct servers not in Q plus the number of faulty servers are more than $2f$ and the faulty servers follow the algorithm, we fall in case 1 above.

In cases (b) and (c) the servers in Q will stop accepting and executing requests until they receive a PRE-PREPARE-MERGE message or execute a state transfer (lines 65-81).

4) If the correct servers are in merge operation when the request with the operation o is received, o is executed when the correct servers go back to the `normal` state, something that we have already shown to eventually happen in the proof of Lemma 3.

■*Theorem* 2

1: *// initialization*
2: $my_v = 1$
3: $m_{last} = 0$
4: $v_{last} = 0$
5: $blacklist = []$
6: $unordered = \{\}$
7: $pending = \{\}$
8: $processing = \{\}$
9: $state =$ `normal`
10: $isPrepared =$ `false`
11: $isPrePrepared =$ `false`

12: **when** s_i receives $\langle \text{REQUEST}, c, seq, op \rangle_{\sigma_c}$ from client c **do**
13: $unordered = unordered \cup \{\langle \text{REQUEST}, c, seq, op \rangle_{\sigma_c}\}$
14: startTimer()

15: **when** s_i becomes the primary of my_v **do**
16: **if** $my_v - 1 == v_{last} \wedge state ==$ `normal` $\wedge\, isPrePrepared ==$ `false` **then**
17: $dm = D(oldestRequest(unordered))$
18: send $\langle \text{PRE-PREPARE}, s_i, v, dm \rangle_{\sigma_{s_i}}$ to all servers
19: $isPrePrepared =$ `true`
20: **else if** (p received at least $2f+1$ MERGE messages sent by different servers in $v-1$) **then**
21: M = {the $2f+1$ MERGE messages}
22: v_{max} = highest view number received in P of MERGE messages
23: **for** each MERGE message $\in M$ **do**
24: **if** P in the MERGE message has view number $\geq v_{max} - n$ **then**
25: $VP[\text{MERGE}.P.v] = \text{MERGE}.P.dm$
26: send $\langle \text{PRE-PREPARE-MERGE}, s_i, v, VP, M \rangle_{\sigma_{s_i}}$ to all servers

27: **when** s_j receives $\langle \text{PRE-PREPARE}, s_i, v, dm \rangle_{\sigma_{s_i}}$ **do**
28: **if** $v == my_v \wedge \langle \text{PRE-PREPARE}, s_i, v, dm \rangle_{\sigma_{s_i}}$ is valid $\wedge my_v - 1 == v_{last} \wedge isPrepared ==$ `false` $\wedge\, state ==$ `normal` **then**
29: send $\langle \text{PREPARE}, s_j, v, dm \rangle_{\sigma_{s_j}}$ to all servers
30: $processing = processing \cup \{\langle \text{PRE-PREPARE}, s_j, v, dm \rangle_{\sigma_{s_j}}\}$
31: $isPrepared =$ `true`

32: **when** s_j receives $2f+1$ $\langle \text{PREPARE}, s_x, v, dm \rangle_{\sigma_{s_x}}$ from different servers **do**
33: **if** $v == my_v \wedge state ==$ `normal` **then**
34: **if** (s_j is the primary of $my_v + 1$) **then**
35: send $\langle \langle \text{COMMIT}, s_j, v \rangle \langle \text{PRE-PREPARE}, s_j, v+1, dm \rangle \rangle_{\sigma_{s_j}}$ to all servers
36: **else**
37: send $\langle \text{COMMIT}, s_j, v \rangle_{\sigma_{s_j}}$ to all servers

38: **when** s_j receives $\langle \langle \text{COMMIT}, s_j, v \rangle, \langle \text{PRE-PREPARE}, s_j, v', dm \rangle \rangle_{\sigma_{s_j}}$ **do**
39: **if** $v == my_v \wedge v' == my_v + 1 \wedge \langle \langle \text{COMMIT}, s_j, v \rangle \langle \text{PRE-PREPARE}, s_j, v', dm \rangle \rangle_{\sigma_{s_j}}$ is valid **then**
40: $pending = pending \cup \{\langle \text{PRE-PREPARE}, s_j, v', dm \rangle\}$

41: **when** s_j receives $2f+1 \langle \text{COMMIT}, s_x, v \rangle_{\sigma_{s_x}}$ from different servers **do**
42: **if** $v == my_v \wedge state ==$ `normal` **then**
43: stopTimer()
44: **if** $\langle \text{PRE-PREPARE}, s_j, v, dm \rangle_{\sigma_{s_j}} \in processing$ **then**
45: **if** $\exists req \in unordered : D(req) = dm$ **then**
46: $reply = exec(req)$ *// requests accepted*
47: send $\langle \text{REPLY}, s_j, reply \rangle_{\sigma_{s_j}}$ to $req.c$
48: $unordered = unordered - \{req\}$
49: **else if** $\langle \text{PRE-PREPARE-MERGE}, s_j, v, VP \rangle_{\sigma_{s_j}} \in processing$ **then**
50: **for** $x = 0$ to size$(VP) - 1$ **do**
51: **if** $\exists req \in unordered : D(req) = VP[x]$ **then**
52: $reply = exec(req)$ *// requests accepted*
53: send $\langle \text{REPLY}, s_j, reply \rangle_{\sigma_{s_j}}$ to $req.c$
54: $unordered = unordered - \{req\}$
55: $v_{last} = v$
56: **repeat**
57: $my_v = my_v + 1$
58: **until** $blacklist$ has my_v mod n
59: $isPrepared =$ `false`
60: $isPrePrepared =$ `false`
61: **if** $\langle \text{PRE-PREPARE}, s_j, v, dm \rangle \sigma_{s_j} \in pending \wedge my_v == v \wedge \langle \text{PRE-PREPARE}, s_j, v, dm \rangle \sigma_{s_j}$ is valid **then**
62: send $\langle \text{PREPARE}, s_j, v, dm \rangle_{\sigma_{s_j}}$ to all servers
63: $pending = pending - \{\langle \text{PRE-PREPARE}, s_j, v, dm \rangle_{\sigma_{s_j}}\}$
64: $isPrepared =$ `true`

65: **when** s_j receives $\langle \text{PRE-PREPARE-MERGE}, s_i, v, VP, M \rangle_{\sigma_{s_i}}$ **do**
66: **if** $v >= v_{last} \wedge \langle \text{PRE-PREPARE-MERGE}, s_i, v, VP, M \rangle_{\sigma_{s_i}}$ is valid $\wedge isPrepared ==$ `false` **then**
67: $processing = processing \cup \{\langle \text{PRE-PREPARE-MERGE}, s_i, v, VP, M \rangle_{\sigma_{s_i}}\}$
68: v_{min} = lowest view number in M
69: **if** $\forall dm \in VP, \exists \text{MERGE} \in M : dm \in M.P \wedge v_{last} + 1 \geq v_{min}$ **then**
70: $my_v = v$
71: $dm = D(VP)$
72: send $\langle \text{PREPARE}, s_j, v, dm \rangle_{\sigma_{s_j}}$ to all servers
73: $isPrepared =$ `true`
74: $state =$ `normal`
75: **if** $m_{last} < v_{last}$ **then**
76: insertHead($(v-1)$ mod n)
77: **else**
78: replaceHead($(v-1)$ mod n)
79: $m_{last} = v$
80: **else**
81: stateTransfer()

82: **when** s_j receives $f+1 \langle \text{MERGE}, s_x, v, P \rangle_{\sigma_{s_x}}$ from different servers $\wedge$ $\forall v >= v_{last}$ **do**
83: $state =$ `merge`
84: $isPrePrepared =$ `false`
85: $isPrepared =$ `false`
86: **if** $\exists req \in processing$ that has $2f+1$ PREPARE messages $\wedge$ has the view number $\geq v_{last} - n$ **then**
87: my_P = {the $2f+1$ PREPARE messages received}
88: send $\langle \text{MERGE}, s_j, v, my_P \rangle_{\sigma_{s_j}}$ to all servers;

89: **when** timer expires on a server s_j **do**
90: **if** $\exists req \in processing$ that has $2f+1$ PREPARE messages $\wedge$ has the view number $\geq v_{last} - n$ **then**
91: my_P = {the $2f+1$ PREPARE messages received}
92: **if** $state ==$ `normal` $\wedge\, \exists my_v \in processing$ that has $f+1$ PRE-PREPARE/PREPARE messages from different servers **then**
93: send $\langle \text{MERGE}, s_j, my_v + 1, my_P \rangle_{\sigma_{s_j}}$ to all servers
94: **else**
95: send $\langle \text{MERGE}, s_j, my_v, my_P \rangle_{\sigma_{s_j}}$ to all servers
96: $state =$ `merge`
97: $isPrepared =$ `false`
98: $isPrePrepared =$ `false`
99: $my_v = my_v + 1$
100: restartTimer()

Algorithm 1: Spinning algorithm