# Highly-Resilient Services for Critical Infrastructures

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*—Modern society depends on several critical infrastructures like power, water, oil and gas generation and distribution. These infrastructures have evolved to become largely controlled by computers and interconnected by computer networks, which lets them exposed to the same types of threats as Internet systems. Therefore, research about mechanisms to improve the protection of these infrastructures is extremely important. Byzantine fault-tolerant (BFT) replication algorithms tackle this problem by allowing critical services, like storage and processing of monitoring data, to continue to operate correctly even if some of their components are compromised by malicious attackers. This paper proposes a novel BFT algorithm that requires fewer replicas, fewer communication steps and analytically seems to have better throughput and latency than others in literature. The main idea is to provide an efficient BFT algorithm suitable to WANs, to be applied in the construction of highly-resilient services for critical infrastructures, tolerating even the physical destruction of some servers.**

## I. Introduction

Critical infrastructures consist of physical and information technology facilities, networks and services whose disruption would have a serious impact on the health, safety or economic well-being of populations. Often, these infrastructures are indirectly connected to the Internet, thus exposed to the same types of threats as home-banking, online shops or personal computers [9]. However, they have a much higher socio-economic value than most Internet systems. Some of them, for instance nuclear plants, are even safety-critical because their failure may cause human deaths and serious harm to the environment. Therefore, using mechanisms that raise the protection of these infrastructures to a new level is a fundamental issue.

Critical infrastructures usually rely on data for several important purposes. For instance, both real-time and stored monitoring data is critical for the right commands to be done by human operators at the right moment. If this data is unavailable or modified with the malicious intent of making the operators do wrong operations, the physical infrastructure may not function correctly or even be physically damaged. Therefore, critical infrastructures require highly-resilient fault-, intrusion- and disaster-tolerant services that function correctly even under harsh cyber-attacks that manage to corrupt some of the computers involved.

A mechanism that allows attaining this objective are Byzantine fault-tolerant algorithms. The basic idea underlying these algorithms is to allow a system to continue to operate correctly even if some of its components exhibit arbitrary behavior, e.g., because there are crashes or intrusions by malicious attackers [1], [2], [3], [4], [5], [7], [10], [11]. Byzantine fault-tolerant systems are usually built using replication techniques. The *state machine approach* is a generic replication technique to implement fault-tolerant services. It was first defined as a means to tolerate crash faults [8] and later extended for Byzantine/arbitrary faults [2]. The algorithms of the latter category are usually called simply BFT. These algorithms typically require $3f+1$ replicas to tolerate $f$ Byzantine servers, e.g., 4 replicas to tolerate one faulty [2].

For tolerating attacks and intrusions, these replicas can not be identical and share the same vulnerabilities. Otherwise, causing intrusions in all the replicas would be almost the same as in a single one. There has to be diversity among the replicas, i.e., replicas shall have different operating systems, different application software, etc. In order to tolerate natural disasters and large-scale attacks like distributed denial-of-service, there should be also diversity in terms of geographical location, i.e., replicas have to be deployed in different sites connected by a wide-area network (WAN).

Current BFT algorithms perform well on local area networks (LANs) but their time complexity limits their ability to scale to WANs, which typically have lower bandwidth, higher and heterogeneous latencies, and exhibit partitions. Furthermore, these algorithms usually rely on a primary replica that is in charge of defining the order in which requests are executed. In large scale environments, the primary replica of these leader-based algorithms becomes a bottleneck that limits the system throughput [6].

In order to deal with these limitations, this paper introduces the *Efficient Byzantine Algorithm for Wide Area networks* (EBAWA), a BFT state machine replication algorithm for large-scale environments. EBAWA is based on a previous work, *Spinning*, a BFT algorithm that constantly rotates the primary [11]. Spinning changes the primary after every batch of pending requests is accepted for execution. Recently Amir et al. described two attacks against the performance of BFT leader-based algorithms that can degrade their performance to let them barely usable [1]. The rotation of the primary lets Spinning mostly unaffected by this kind of performance attacks, which makes it ideal to run in WANs where detecting a faulty primary takes much longer than in LANs (communication delays are larger). EBAWA also uses a rotating primary with the purpose of avoiding these attacks and for load balancing.

On another note, recently some algorithms that require only $2f+1$ replicas instead of the above mentioned $3f+1$ were published [5], [3]. This reduction from $3f+1$ to $2f+1$, e.g., from 4 to 3 replicas to tolerate a faulty one, is possible with a hybrid system model, i.e., by extending the system model with a trusted/trustworthy component that constrains the power of faulty processes to have certain behaviors. The above mentioned need of diversity involves additional considerable costs per replica, in terms not only of hardware but especially of software development, acquisition and management. Then, reducing the number of replicas has a significant impact in the system cost. We have proposed a minimal trusted/trustworthy service to be used by BFT algorithms to support this reduction of the number of replicas [10]. This Unique Sequential Identifier Generator service (USIG) contains a monotonic counter plus a few cryptographic functions that are used to associate sequence numbers to certain operations done by the replicas.

The EBAWA algorithm exploits the USIG service in order to constrain the behavior of faulty replicas and use only $2f+1$ replicas. EBAWA can be seen as a modification of the Spinning algorithm that requires only $2f+1$ replicas, unlike Spinning that requires $3f+1$. The main idea of this paper is to combine the best attributes of Spinning with the USIG service and other mechanisms in order to obtain an efficient BFT algorithm suitable for WANs. Mao et. al [7] explore the use of a trusted service [3] and a rotating primary in order to reduce latency in BFT algorithms. This work goes one step further by proposing a novel BFT algorithm that requires fewer replicas, fewer communication steps and analytically seems to have better throughput and latency than others in literature.

As mentioned before, critical infrastructures depend on services as storage of critical information from the monitoring systems, which if modified by cyber-criminals may impair the normal operation of the infrastructure. EBAWA allows the implementation of highly-resilient services in WANs, like storage of critical monitoring data, DNS or PKI services, even if there are cyber-attacks, intrusions or the physical destruction of some of the service replicas.

## II. System Model

We model the system as a set of $n$ servers $P = \{s_0, ..., s_{n-1}\}$ interconnected by a network that together provide a Byzantine fault-tolerant service to a set of clients. The network can drop, reorder and duplicate messages, but these faults are masked using common techniques like packet retransmissions. Messages are kept in a message log for being retransmitted. We do not make assumptions about processing or communication delays, except that these delays do not grow indefinitely. This weak assumption has to be satisfied only to ensure the liveness of the system, not its safety. An attacker may have access to the network and be able to modify messages, so messages contain digital signatures or message authentication codes (HMACs). Servers and clients know the keys they need to check these signatures/HMACs.

*Correct* servers/clients always follow their algorithm. *Faulty* servers/clients can deviate arbitrarily from their algorithm, even by colluding with some malicious purpose. This class of unconstrained faults is usually called *Byzantine* or *arbitrary*. We assume that at most $f$ out of $n$ servers can be faulty for $n = 2f+1$. For simplicity of presentation, we consider the tight case ($n = 2f+1$) and not the generic case ($n \geq 2f+1$), but the generalization is trivial. Each server contains a local trusted/tamperproof component that provides the USIG service. Therefore, the fault model we consider is *hybrid*. The Byzantine model states that any number of clients and any $f$ servers can be faulty. However, the USIG service is tamperproof, i.e., always satisfies its specification, even if in a faulty server.

**The USIG Service.** The Unique Sequential Identifier Generator (USIG) is a service provided locally in each server by a module that has to be built to be trusted/trustworthy (or secure) [10]. There is no communication among the modules in different servers. The service assigns to messages (i.e., arrays of bytes) identifiers with the guarantee that (1) it will never assign the same identifier to two different messages (uniqueness), (2) it will never assign an identifier that is lower than a previous one (monotonicity), and (3) it will never assign an identifier that is not the successor of the previous one (sequentiality). The main components of the service are a *counter* and cryptographic mechanisms. The interface of the service has two functions:

`createUI`$(m)$ – returns a *USIG certificate* that certifies that the *unique identifier UI* contained in the certificate was created by this tamperproof component for message $m$. The unique identifier includes the value of the monotonic counter, which is incremented whenever `createUI` is called.

`verifyUI`$(UI, m)$ – verifies if the unique identifier $UI$ is *valid* for message $m$, i.e., if the USIG certificate matches the message and the rest of the data in $UI$.

The USIG certificate contains a HMAC obtained using the message and a secret key owned by this USIG but known by all the others, for them to be able to verify the certificates generated. The service properties (e.g., uniqueness) are based on the secretness of the shared keys. Therefore both functions `createUI` and `verifyUI` must be implemented inside the tamperproof component. The implementation of the service is based on an isolated, tamperproof component that we assume can not be corrupted. More details about the USIG implementation can be found in [10].

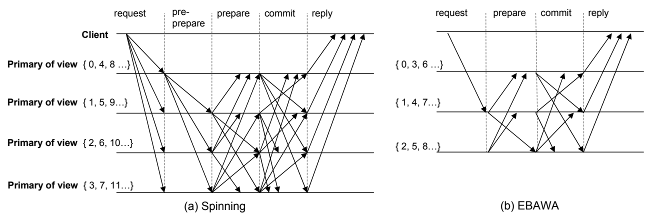

Figure 1. Normal operation, communication pattern between the servers in the (a) Spinning and (b) EBAWA algorithms.

## III. The EBAWA Algorithm

In the state machine approach, each server maintains a set of state variables that are modified by a set of operations. Clients of the service issue *requests* with operations through a replication algorithm which ensures that (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 EBAWA in terms of the modifications carried out in the Spinning algorithm in order to reduce the number of replicas from $3f+1$ to $2f+1$ with the assistance of the USIG service.

EBAWA follows a message exchange pattern similar to Spinning (see Figure 1). In comparison with Spinning, EBAWA has one less communication step and needs fewer replicas. Like Spinning, in EBAWA the servers/replicas move through successive configurations called views. Each view has a primary that changes in a round-robin fashion. The primary is the server $s_p \triangleq v \ mod \ n$, where $v$ is the current view number. Notice that in the EBAWA algorithm only one request is executed per view (a simplification of presentation removed later) and each message exchanged by the servers has a unique identifier assigned by the USIG service.

Clients and servers can be scattered geographically. To tolerate natural disasters, servers have to be in different sites , while clients can be in the same sites of servers or somewhere else. Nevertheless, there is a notion of proximity: a client is *nearer* to certain servers than others. This proximity is not necessarily physical but in terms of communication latency. Clients typically send requests only to the nearest server. Servers have to communicate among themselves.

We now explain the algorithm. The servers run in two modes: normal operation and merge operation.

**Clients.** Clients have a list of servers that contains at least the $f+1$ nearest servers, ordered from the nearest to the farthest. A client $c$ issues a request $m$ for the execution of an operation $m$ by sending a message $\langle \text{REQUEST}, c, seq, m \rangle_{\sigma_c}$ to the first server of the list.

The *seq* field is the request identifier that is used to ensure exactly-once semantics: the servers do not execute a request of the client with a *seq* lower than the last executed in order to avoid executing the same request twice. If the client does not receive enough replies during a time interval, it resends the client request to the next in the server list. In case the request has already been processed, the server resends the reply. $\sigma_c$ is the signature of the message.

**Servers – normal operation.** A request $m$ is sent by the primary $s_i$ to all servers in a message $\langle \text{PREPARE}, s_i, v, UI_i, m \rangle$ where $UI_i$ is obtained by calling function *createUI*. Each server $s_j$ resends the request to all others in a message $\langle \text{COMMIT}, v, s_j, UI_i, UI_j \rangle$. Each message sent, either a PREPARE or a COMMIT has a unique identifier $UI$ obtained by calling the *createUI* function, so no two messages can have the same identifier. Servers check if the identifier of the messages they receive are valid for these messages using the *verifyUI* function. A request $m$ is accepted by a server if the server receives $f+1$ valid COMMIT messages from different servers for $m$.

A correct server $s_j$ multicasts a COMMIT message in response to a PREPARE message received from $s_i$ only if $s_j$ already accepted request $m'$ sent by $s_i$ with counter value $cv' = cv - 1$, where $cv$ is the counter value in $UI_i$ (to prevent a faulty primary from creating "holes" in the sequence of messages) and $s_i$ is the primary of view $v$. This message ordering mechanism imposes a FIFO order: no correct server processes a message $\langle ..., s_i, ..., UI_i, ... \rangle$ sent by any server $s_i$ with counter value $cv$ in $UI_i$ before it has processed message $\langle ..., s_i, ... UI_i', ... \rangle$ sent by $s_i$ with counter value $cv - 1$.

In order to reduce the communication steps and computation overhead, we borrow from Mencius [6] the idea of SKIP messages. Servers without pending client requests can skip their turns by sending a SKIP message, with the format $\langle \text{SKIP}, s_i, v, UI_i \rangle$. To avoid that a faulty primary sends to the same view a SKIP message and a PREPARE message to different subgroups of backups, the SKIP message has also an unique identifier.

**Servers – merge operation.** The objective of this operation is to force servers 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 were lost or not sent, and some of the correct servers accept the requests, but other correct servers do not.

The USIG service strongly constrains what a faulty primary can do. However, a faulty primary can still prevent progress by not assigning sequence numbers to some requests. Therefore, a server waits a maximum time interval $T_{acc}$ to accept the request or *skip* that view. If in a view $v$ a server does not receive enough COMMIT messages to accept the request neither a SKIP message during $T_{acc}$, then it sends a MERGE message for view $v$ to all 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, it sends $\langle \text{MERGE}, s_j, v, C_{last}, P, O, UI_j \rangle$ to all servers, where $C_{last}$ is a commit certificate of the last accepted request. The field $P$ contains valid PREPARE messages with view number greater than $C_{last}.v$ and less or equal to $v$. $O$ contains all signed messages sent by the server since the last accepted request.

Correct servers only consider MERGE messages that are consistent with the system state: (1) the commit certificate $C_{last}$ contains $f+1$ valid $UI$ identifiers; (2) the counter value ($cv_i$) in $UI_j$ is $cv_j = cv+1$, where $cv$ is the highest counter value of the $UI$s signed by the replica in $O$; if $O$ is empty the highest counter value will be the $UI$ in $C_{last}$ signed by the replica when it accepted the request; and (3) there are no holes in the sequence number of messages in $O$.

If the server did not send a MERGE message before, 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. When $s_i$ receives $f+1$ MERGE messages from view $v-1$ and it is the primary of the view $v$, it sends $\langle \text{PREPARE-MERGE}, s_i, v, VP, M, UI_i \rangle$ to all servers. $VP$ is a vector of digests of prepared requests taken from the $P$ field of the MERGE messages, ordered by view number. To compute $VP$, the primary starts by selecting the most recent (valid) commit certificate received in MERGE messages. Next, it picks in MERGE messages the PREPARE messages in $P$ sets with $v$ greater than the view number in the commit certificate and $v$ lower than $v$. If the primary have been sent before its PREPARE or SKIP message to view $v$ it adds the message to $VP$. $M$ is a *merge certificate* composed by the $f+1$ 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*.

When a server $s_j$ receives a valid message $\langle \text{PREPARE-MERGE}, s_i, v, VP, M, UI_i \rangle$ from $s_i$ it evaluates if: (i) the sender is the primary of $v$; (ii) $VP$ is valid (using the merge certificate $M$ to do the same computation as the primary). (iii) the $UI_i$ of the message is valid. If the PREPARE-MERGE message is valid a replica changes its state to normal and sends a COMMIT message. Server increments the view $v'$ after all prepared requests in $VP$ that have not been executed before, are executed. If a replica detects that there is a hole in the sequence number of the last request that it executed and the first request in $VP$, it requests to other replicas the commit certificates of the missing requests to update its state. If due to the garbage collection the other replicas have deleted these messages, there is a state transfer.

**Other issues.** As Spinning, EBAWA uses checkpoints only for state transfer purposes. Also, if a server is faulty it can periodically impair the performance of the service by delaying sending some of the messages it has to send. To circumvent this problem, we use Spinning's blacklist mechanism. Servers that cause merge operations are put in a blacklist and do not become the primary. Also like Spinning, the timeouts of the merge operation have to be increased and decreased depending on the network delays.

There are several optimizations to the algorithm that we do not have space to discuss in detail. Servers can batch all pending requests in a single PREPARE message, instead of sending that message for a single request. Servers can send PREPARE or SKIP messages together with COMMIT messages, doing the two communication steps together.

## IV. Evaluation

This section provides an analytical comparison of EBAWA with other BFT algorithms in the literature. Table I presents a summary of the characteristics of the algorithms. EBAWA and A2M-PBFT-EA require only $2f+1$, instead of the usual $3f+1$ replicas (lines "Model" and "Total replicas" in the table).

The number of communication steps is an important metric for distributed algorithms, for the delay of the communication tends to have a major impact in the latency of the algorithm (line "Latency"). This is specially important in WANs, where the communication delay can be as much as a thousand times higher than in LANs. When compared with other BFT algorithms, EBAWA and MinBFT run in the minimum known number of communication steps. However, MinBFT is vulnerable to performance degradation attacks, something that can be disastrous in WANs. The reason is that the timeouts used to detect if the primary is misbehaving have to be higher than the round trip time plus a margin to compensate delay variations that are common in WANs.

EBAWA, Spinning, Prime and Aardvark mitigate performance degradation attacks (line "Resilience"). Prime introduces a pre-order phase with three communication steps before the global order (based on PBFT) that, together with the constant monitoring of the performance of the primary, make the system able to detect performance attacks. Aardvark follows the same communication pattern than PBFT and proposes a constant monitoring of the throughput sustained during a view plus the periodic change of primary through the execution of a view change operation. As in Spinning, EBAWA rotates the primary after each request, which has shown to be more efficient and resilient than the solutions of Prime and Aardvark.

In EBAWA, servers process less messages than the servers of other algorithms, which means less network I/O and less cryptographic operations (line "Throughput"). Figure 2 illustrates these costs for different values of $f$ and $b$ (size of batch), comparing the number of cryptographic operations done by the primary per client request for different BFT algorithms. Signatures based on public-key cryptography are known to be much slower to create and verify than HMACs, so it is important to notice that Prime and Aardvark use public-key signatures, while EBAWA does not. EBAWA potentially has the best latency and throughput of all BFT algorithms in the table due to the reduced number of replicas, communication steps and the load balancing between the servers provided by the rotating primary.

| | | PBFT [2] | Aardvark [4] | Spinning [11] | Prime [1] | A2M-PBFT-EA[3] | MinBFT [10] | EBAWA (this paper) |
|---|---|---|---|---|---|---|---|---|
| Model | Tamperproof component | no | no | no | no | A2M | USIG | USIG |
| Cost | Total replicas | $3f+1$ | $3f+1$ | $3f+1$ | $3f+1$ | $2f+1$ | $2f+1$ | $2f+1$ |
| Throughput | HMAC ops at bottleneck server | $2+\frac{8f+1}{b}$ | $2+\frac{8f+1}{b}$ (*) | $\frac{2+\frac{8f+1}{b}+3f(2+\frac{5f+1}{b})}{3f+1}$ | $2+8f+\frac{13f}{b}$ (*) | $2+\frac{2f+4}{b}$ | $2+\frac{f+3}{b}$ | $\frac{2+\frac{1+(f+2)}{b}+2f(1+\frac{1+(f+1)}{b})}{2f+1}$ |
| Latency | Communication steps | 5 / 4 | 5 | 5 | 8 | 5 | 4 | 4 |
| Resilience | Performance degr. attacks | vulnerable | tolerant | tolerant | tolerant | vulnerable | vulnerable | tolerant |

Table I
COMPARISON OF BFT ALGORITHMS. $b$ AND $c$ ARE RESPECTIVELY THE SIZE OF THE BATCH OF REQUESTS AND COMMITS. IN THE ALGORITHMS THAT USE A TAMPERPROOF COMPONENT, SOME HMACS ARE DONE INSIDE THIS COMPONENT. THE STAR (*) INDICATES SIGNATURES INSTEAD OF HMACS.

## V. CONCLUSION

Nowadays critical infrastructures are essentially physical processes controlled by computers connected by networks. They are usually as vulnerable as any other networked computer system, but their failure has a high socio-economic impact and some of them are even safety-critical. Therefore, critical infrastructures require highly-resilient services that function correctly even under harsh cyber-attacks that manage to corrupt some of the computers involved. We show a BFT algorithm that can be used to implement highly-resilient services in WANs, like storage of critical monitoring data, DNS or PKI services, that remains operational even in presence of intrusions or the physical destruction of some of the service replicas. EBAWA mitigates performance attacks by changing the primary after every batch of pending requests is accepted for execution. This mode of operation limits the number of messages exchanged between the servers, avoids that the primary replica becomes a bottleneck and consequently improves the system throughput. All these attributes make EBAWA ideal for highly-resilient services for critical infrastructures.

**Acknowledgments.** This work was partially supported by Alban scholarship E05D057126BR, and by the FCT through the Multi-annual and the CMU-Portugal Programmes.

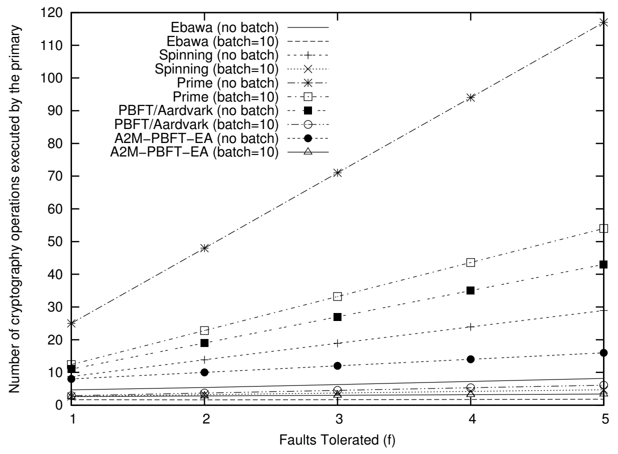

Figure 2. Comparison of the number of cryptography operations processed by the primary in BFT algorithms.

## REFERENCES

[1] 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*, pages 197–206, June 2008.

[2] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[3] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[4] 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.

[5] 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*, pages 174–183, Oct. 2004.

[6] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008.

[7] Y. Mao, F. P. Junqueira, and K. Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *Workshop on Hot Topics in System Dependability*, June 2009.

[8] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[9] P. Verissimo, N. F. Neves, and M. Correia. CRUTIAL: The blueprint of a reference critical information infrastructure architecture. In *Proceedings of the 1st International Workshop on Critical Infrastructures*, Aug. 2006.

[10] G. S. Veronese, M. Correia, A. Bessani, L. Chung, and P. Verissimo. Minimal Byzantine fault tolerance: Algorithm and evaluation. DI/FCUL TR 09–15, Department of Computer Science, University of Lisbon, June 2009.

[11] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems*, Sept. 2009.