EBAWA: Efficient Byzantine Agreement for Wide-Area Networks

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*—The popularity of wide-area computer services has generated a compelling need for efficient algorithms that provide high reliability. Byzantine fault-tolerant (BFT) algorithms can be used with this purpose because they allow replicated systems to continue to provide a correct service even when some of their replicas fail arbitrarily, either accidentally or due to malicious faults. Current BFT algorithms perform well on LANs but when the replicas are distributed geographically their performance is affected by the lower bandwidth and the higher and more heterogeneous network latencies. This paper proposes and evaluates a novel BFT algorithm for WANs that requires fewer communication steps, fewer replicas and has better throughput and latency than others in the literature. The paper presents an extensive evaluation of the algorithm's performance in several settings and conditions: in a LAN; in real and emulated WANs; with clients close to servers and dispersed geographically; with similar and different communication latencies between clients and servers.**

I. Introduction

The growing reliance of our society on wide-area services demands highly reliable systems that provide correct and uninterrupted services. One of the most popular techniques for building fault-tolerant distributed services is the state machine approach [1]. The main idea is to replicate the service in a set of servers and make clients issue requests to the service running an algorithm that ensures that all correct replicas execute the same sequence of requests.

Several practical algorithms to implement efficient *Byzantine Fault-Tolerant* (BFT) state machine replication have been proposed. PBFT [2] was the first of them shown to be efficient in practice, but several others derived from it [3], [4], [5], [6], [7]. For these two reasons, PBFT is often considered to be a baseline for performance of BFT algorithms in local area networks (LANs), which other algorithms improve under certain conditions.

The main motivation for this paper is that for a replicated service to be fault-tolerant, common mode failures have to be avoided. More specifically, the paper is concerned with common mode failures caused by natural disasters, power outages and physical attacks, which have to be prevented by scattering replicas geographically. This requires the sites where the replicas reside to be connected by a wide-area network (WAN) like the Internet.

Unfortunately, when the replicas are distributed geographically the performance of current BFT algorithms is affected by the lower bandwidth, and the higher and more heterogeneous network latencies. Furthermore, these algorithms usually rely on a primary replica for defining the order in which requests are executed, and in a WAN it is difficult to detect that the primary is faulty or that the latency and/or bandwidth of its communication with the other replicas has become a bottleneck for the service performance.

In order to cope with these challenges, this paper presents the Efficient Byzantine Agreement for Wide-Area networks (EBAWA), a new BFT state machine replication algorithm designed for WANs that has a reduced number of communication steps, requires only $2f+1$ replicas, and introduces a set of mechanisms focused in providing a better performance in large-scale environments. The communication pattern of EBAWA is based on a BFT algorithm that we designed previously, *Spinning* [7]. Like Spinning, EBAWA rotates the primary constantly, i.e., changes the primary after every batch of pending requests is accepted for execution. The rotation of the primary makes EBAWA mostly unaffected by certain performance degradation attacks [3] and enables the use of mechanisms for WANs.

Spinning executes agreements on client requests in sequence, one agreement per view. In WANs, this can lead to undesirable delays due to the high latency and low bandwidth of the communication links. In this paper we introduce a new mechanism called *asynchronous views*, one of the key features that make EBAWA efficient in WANs. The asynchronous views mechanism consists in parallelizing several agreements, i.e., several views. This mechanism allows not only to reduce the delay of a server waiting for its turn for becoming the primary, but also allows servers to batch messages of different agreements/views in a single message, reducing the number of sent messages and the overhead of cryptographic operations. This mechanism reduces the latency of the algorithm and improves the throughput because there are fewer messages being processed.

The use of a rotating primary and asynchronous views allow another interesting feature: clients can send requests only to the nearest server. In WANs this tends to reduce the end-to-end latency experienced by the client. However, in Spinning it would have the opposite effect, because each server would have to wait to become the primary to start an agreement for the execution of the requests sent by the clients. EBAWA uses the *skip* mechanism borrowed from Mencius to avoid

executing an agreement when a server has no pending requests [8].

Finally, EBAWA uses a *trusted component* on the servers to reduce the number of replicas and the number of communication steps required for agreement. This component implements a simple service called Unique Sequential Identifier Generator (USIG) [9], which comprises a counter plus a cryptographic authentication algorithm, thus being simple to implement in a trustworthy way. The service provided by this trusted component makes EBAWA match lower bounds only comparable with crash fault-tolerant algorithms: $2f+1$ replicas to tolerate f faulty and 2 communication steps for agreement [10]. Moreover, minimizing the number of communication steps is fundamental to improve the performance of a distributed algorithm for WANs, since link latencies tend to dominate the end-to-end latency experienced by clients accessing a replicated service.

In summary the main contributions of this paper are the following:

- it presents EBAWA, a new BFT algorithm suitable for supporting the execution of wide-area replicated services. EBAWA has a reduced number of communication steps, is the first BFT algorithm for WANs that requires only $2f+1$ replicas, and combines a set of mechanisms focused in providing better performance in large-scale environments.
- it presents a thorough evaluation of the performance of EBAWA in several settings and conditions: in a LAN; in real and emulated WANs; with clients close to servers and dispersed geographically; with similar and different communication latencies between clients and servers. In particular, it presents, for the first time, a performance evaluation of BFT state machine algorithms on a *real wide-area network* (PlanetLab). The results of these experiments show that the techniques employed by EBAWA make it an efficient wide-area replication algorithm.

II. Related Work

There are two BFT algorithms in the literature that were designed for WANs [11], [12]. Steward is a hierarchical state machine replication architecture [11]. Each site has a group of servers that play the role of a single participant in a wide-area algorithm. Steward's main protocol is a lightweight, crash fault-tolerant consensus algorithm executed between sites, while within each site it runs PBFT. EBAWA has a more flexible architecture than Steward because it does not have a hierarchical architecture and clients and servers can be localized in the same site or dispersed geographically. EBAWA has only two sub-algorithms that deal with the normal case operation and merge operation, while Steward has over ten specialized algorithms that run within and among sites, most of which are associated with global view changes. Besides, EBAWA tolerates f malicious servers that can be localized in different sites, while Steward can withstand f out of $3f+1$ Byzantine failures within each site but cannot survive even a single site compromise.

Mencius is an algorithm for efficient state machine replication in WANs that also changes the primary for each agreement [8] but that has many important differences to EBAWA. Mencius only tolerates replica crashes, while EBAWA tolerates Byzantine replicas. In Mencius servers other than the primary do not exchange messages among themselves, while EBAWA has an all-to-all communication pattern. In EBAWA clients can be arbitrarily scattered geographically, while Mencius uses a fate sharing model in which the service is used only in a set of data centers and clients can only communicate with the service through the replica in their data center (clients may be unable to use the service if the local server is faulty). EBAWA uses several mechanisms that do not exist in Mencius but are important for tolerating malicious behavior and increasing performance: asynchronous views, batches of messages, merge operation, timeout reset, and blacklisting of faulty servers.

Mao et. al [12] describe the design principles of a BFT algorithm for WANs, the RAM algorithm, by exploring the use of the A2M trusted service [13] and a rotating primary. When compared to RAM, EBAWA requires fewer replicas ($2f+1$ instead of $3f+1$) and uses a trusted service that is simpler, easier to implement and make trustworthy. The USIG service provides an interface with operations only to increment a counter and to verify if counter values are correctly signed, while A2M provides a log that can grow considerably and an interface with functions to append, lookup and truncate messages in the log. Besides, EBAWA uses a flexible model, while RAM follows the Mencius' fate sharing model.

There are three previous BFT state machine replication algorithms that require only $2f+1$ replicas. However, they were not designed specifically for WANs and do not include the mechanisms that we use in EBAWA with the purpose of obtaining efficiency in those environments, with the exception of a trusted component. The first algorithm of the kind is based on a distributed trusted component called TTCB [14]. The second is based on the above mentioned A2M local trusted component [13]. Finally, MinBFT uses USIG itself, but does not include any of the other mechanisms of EBAWA [9]. USIG [9] is very similar to TrInc [15], which was proposed independently in the literature at the same time.

III. System Model

We model the system as a set of n servers (or replicas) $\{s_0,...,s_{n-1}\}$ that provide a Byzantine fault-tolerant service to a set of clients $\{c_0,c_1,...\}$. Clients and servers are interconnected by a LAN or a WAN and communicate only by message-passing. The bandwidth between pairs of servers can be asymmetric and variable. 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 when necessary. We assume a partial synchronous system model [16], i.e, the system can be asynchronous but there are (unknown) communication and processing delay bounds that are respected after some (unknown) instant of time. An attacker may have access to the network and be able to modify messages, so messages take

digital signatures or message authentication codes (HMACs). Servers and clients know the keys they need to verify these signatures/HMACs. We make the standard assumptions about cryptography that hash functions are collision-resistant and that signatures and HMACs cannot be forged.

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 with $n = 2f+1$. The fault model considered is *hybrid* [17]: although any number of clients and up to f servers can be subject to Byzantine faults, the modules that implement the USIG service on servers are trusted/trustworthy, i.e., they always satisfy the specification of the USIG service.

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 [9]. More precisely, the confidentiality of its key and the integrity of its operation are assumed to be held even if an adversary has access to the server. There is no communication among the modules in different servers, so each correct server s_i has exclusive access to module w_i of a set of modules $\{w_0, ..., w_{n-1}\}$.

The service is used to assign identifiers to messages ensuring the following properties: (1) it will never assign the same identifier to two different messages (*uniqueness*), and (2) it will never assign an identifier that is not the successor of the previous one (*sequentiality*). The interface of the service has two functions: (1) `createUI`(m) returns a *unique identifier* UI, which is a data structure containing a counter value and a *USIG-certificate* that certifies that UI was created by this module w_i for message m; (2) `verifyUI`(UI, m) verifies if the UI is *valid* for message m, i.e., if the USIG certificate matches the message and the rest of the data in UI.

The main components of a module w_i are a *counter* and cryptographic mechanisms. The unique identifier is a reading of the counter, which is incremented whenever `createUI` is called. When this function is called in module w_i, it returns a USIG certificate containing a HMAC that is obtained using the message m and a secret key K_i. This key is shared with all modules w_j, which use it to validate the certificate. The service properties are based on the secretness of the shared key, then both functions must be implemented inside a tamperproof component. This tamperproofness can be obtained even on COTS trusted hardware, such as the *Trusted Platform Module* (TPM) [18]. More details about the USIG implementation can be found in [9].

IV. EBAWA MECHANISMS

The state machine approach consists of replicating a service in a group of servers. Each server maintains a set of *state variables*, which are modified by a set of *operations*. Clients of the service issue *requests* with operations through a replication algorithm which ensures *safety* (all correct servers execute the same requests in the same order) and *liveness* (all correct clients' requests are eventually executed).

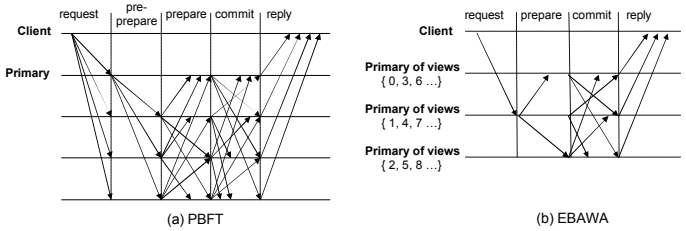


Figure 1. Communication patterns of a *single agreement* (a) in PBFT and Spinning and (b) in EBAWA.

This section presents the mechanisms that EBAWA combined in order to be efficient in WANs.

Trusted/trustworthy component. EBAWA explores the use of a trusted component that provides the USIG service to constrain the power of faulty processes to have certain behaviors. This allows the reduction of the number of communication steps (from the previous minimum[1] of 5 [2], [4] to 4) and of the number of replicas (from $3f+1$ [2], [3], [4], [5], [6] to $2f+1$). Figure 1 illustrates these improvements by showing the communication patterns of PBFT and EBAWA. The number of communication steps is an important metric for distributed algorithms, as discussed before. Besides, the already mentioned need of avoiding common mode failures requires diversity of the replicas [19], which involves additional considerable costs per replica, in terms not only of hardware but especially of software development and management. Then, reducing the number of replicas has a significant impact in the system cost.

Rotating Primary. EBAWA follows the idea of Spinning of changing the primary whenever a batch of requests is accepted for execution. An individual agreement in Spinning follows essentially the communication pattern of PBFT (Figure 1(a)), but in Spinning the primary changes automatically in a round-robin fashion whenever it defines the order of a single batch of requests. Spinning 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.

The use of rotating primary has two main benefits. The first is that it avoids performance attacks made by faulty servers to which many BFT algorithms are vulnerable [3] in a very simple and efficient way: by always changing the primary. The second is that it improves the throughput of PBFT when there are no faulty servers by balancing the load of ordering requests among all (correct) servers.

Clients scattered geographically. In order to tolerate natural disasters, power outages and physical attacks, servers have to be in different locations. Previous works on wide-area BFT replication consider that clients are in the same site than

[1]There is an optimization called *tentative execution* that allow these algorithms to run in 4 steps, but it only works in synchronous and fault-free environments [2].

servers (e.g., in the same data center) [11], [12], and their fate is shared with the site's server. The main problem with this fate sharing model is that clients of a site are unable to access the service if their local server is compromised. Moreover, this model makes it unclear how to deal with clients that are not located on any of the replica's sites.

In EBAWA clients can be placed anywhere. We use the notion of proximity to associate clients with servers: a client is *nearer* to certain servers than others. This proximity is in terms of communication round-trip time, so a client sends requests only to the server that replies faster (unless it is faulty). In order to reduce the cost of communication, the nearest server is also in charge of sending the complete result of the requested operation to the client, while others send only a digest (hash). This optimization is important when results are large. Clients verify if a result is valid by matching f digests with the complete result.

Asynchronous views. In Spinning a server s only becomes the primary of view v when the batch of requests of view $v-1$ is accepted, so only then it can send a PRE-PREPARE message defining the batch of requests to be executed in view v. In other words, *agreements* about batches of messages (pairs of steps prepare-commit) are done sequentially, following the sequence of views. In EBAWA, the order of the requests is defined by PREPARE messages. Differently of Spinning, in EBAWA a server starts an agreement as soon as it receives a client request by sending a PREPARE message, allowing servers to deal with many agreements (in some sense, with many views) in parallel, or asynchronously. Therefore, when a server receives a sequence of PREPARE messages it can batch the COMMIT messages in a single one, reducing the number of communication steps among servers and the overhead of cryptographic operations (one unique identifier per message).

As mentioned before, reducing the number of steps among servers has a significant impact in the latency of wide-area algorithms. Reducing the number of cryptographic operations increases the system throughput. Therefore, this new feature impacts both the latency and the throughput of the algorithm.

Skip messages. In order to reduce the communication steps and computation overhead, we borrow from Mencius [8] the idea of SKIP messages. Servers without pending client requests can skip their turn, i.e., their view, by sending a SKIP message. A SKIP message is equivalent to an empty PREPARE message but avoids that each server sends its COMMIT message, reducing the number of communication steps of the agreement. To prevent that a faulty primary sends in the same view a SKIP message and a PREPARE message to different subsets of replicas, the SKIP message is always signed with an unique identifier, just like a PREPARE message. The performance evaluation shows that this modification is especially important when the load of client requests shifts from one server to another.

V. The EBAWA Replication Algorithm

This section provides a more detailed description of the EBAWA algorithm. The detailed pseudocode and proof of correctness are in the extended version of the paper [20].

Each server is always in one of two states: *normal* or *merge*. In normal operation each server orders only one batch of requests or skips its turn. Servers agree on requests and execute their operations. When a server is faulty a merge operation is executed to put together the information from different servers and decide which requests in the previous views were executed. We use two mechanisms to avoid that a faulty replica periodically impairs the performance of the system: *blacklisting* and *timeout resetting*. Table I serves as a reference to the fields of the messages of the algorithm.

Client Messages - Request	
c	client identifier
seq	request identifier
op	operation requested to be executed
Server Messages - Normal operation	
s	server identifier
v	view number
m	request message
UI	unique identifier of a message signed by the USIG of its sender
Server Messages - Checkpoint	
v_{last}	the view number of the last request accepted by s
UI_{last}	is the unique identifier of the last executed request
d	digest of the replica state
Server Messages - Merge operation	
C_{last}	latest stable checkpoint certificate
P	a set of valid PREPARE messages received by a server since C_{last}
O	a set of all messages signed by the local USIG service since C_{last}
VP	vector with requests taken from the P field of MERGE messages (ordered by view number)
M	set of $f+1$ MERGE messages (merge certificate)

Table I
LABELS GIVEN TO FIELDS IN MESSAGES

Normal Case Operation. The sequence of events is (cf. Figure 1(b)): (1) a client sends a request to the nearest server; (2) the server assigns an execution order number to the request (the unique identifier) and sends it to all servers in a PREPARE message; (3) when a server receives a valid PREPARE from the primary it sends a COMMIT message to the other replicas; (4) when a server receives $f+1$ valid COMMIT messages it accepts a request, executes the operation, and returns a reply to the client (5) the client waits for $f+1$ matching replies for the request and completes the operation. Next we describe these events in more detail.

Clients keep a list with the nearest servers. A client c issues a request for the execution of an operation op by sending a message $m = \langle \text{REQUEST}, c, seq, op \rangle_{\sigma_c}$ to the nearest server (see the meaning of fields in Table I). The seq field is used to ensure at-most-once semantics: servers do not execute a request with a seq lower or equal than the last executed from that client, in order to avoid executing the same request twice (if a malicious client does not follow the sequence, its requests are discarded). If the client does not receive enough replies during a time interval, it resends the request to the next in the server list. In case the request has already been processed, the server resends the reply.

A request m is sent by the primary s_i in a message $\langle \text{PREPARE}, s_i, v, UI_i, m \rangle$ where UI_i is obtained by calling `createUI`. When a server s_j receives a PREPARE message from s_i, it evaluates if: (i) UI_i is valid (by calling `verifyUI`);

(ii) the client signature is valid; (iii) the view number v is equal to the current view number on s_j and the sender is the primary of v; and (iv) it is in the normal state. If these conditions are satisfied the message is said to be valid and s_j sends a $\langle \text{COMMIT}, s_j, v, UI_i, UI_j \rangle$ message to all others.

Each PREPARE or COMMIT has a unique identifier UI produced by a USIG module, so *no two messages can have the same identifier* from the same server. Servers check if the identifier of the messages that they receive are valid calling 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, i.e., $f+1$ COMMIT messages that contain valid UIs with the same primary's UI.

Servers process messages following the primaries UIs and the view number. Each server keeps a vector $V_{acc}[n]$ with the highest counter value cv that it received from each of the other servers in all signed messages of the algorithm. Using V_{acc} a replica easily detects holes in the sequence numbers of messages of potentially faulty servers. This message ordering mechanism imposes a FIFO order: no correct server processes a message $\langle ..., s_i, ..., UI_i, ... \rangle$ sent by a 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 algorithm overhead, servers assign a single UI to a batch of requests and start a single agreement (i.e., send one PREPARE message). A (correct) server only starts an agreement if the number of unfinished agreements that it started is lower than W, so this parameter bounds the number of *pending* agreements created by each server. This allows each server to deal with $n \times W$ agreements in parallel or asynchronously. Therefore, the COMMITs for a sequence of PREPARE messages can be batched in a single message, reducing the number of messages.

With the asynchronous views mechanism, servers usually receive PREPARE messages for views greater than the current one (v). If a PREPARE mp for view $v' > v$ is valid, the server buffers mp. When it finishes accepting all messages with $v'' < v'$, it sends the COMMIT message of view v'. In order to cause buffer exhaustion malicious servers might send messages with high view numbers. To prevent this, correct servers discard messages with view number higher than $v + n \times W$, when the buffer free space drops below some low water mark L (a system parameter). L should be big enough so that replicas do not stall waiting for a stable checkpoint to remove messages from the log (see checkpoint mechanism in the next section) and greater or equal to $n \times W$ to allow servers to deal with the maximum number of parallel agreements.

Servers without pending client requests can skip their turns by sending a message $\langle \text{SKIP}, s_i, v, UI_i \rangle$. In order to reduce the computation overhead, SKIP messages can be also piggybacked with COMMIT messages.

Checkpointing. As in PBFT, replicas generate and multicast periodically checkpoints of the format $\langle \text{CHECKPOINT}, s, v_{last}, UI_{last}, d, UI \rangle$. UI is obtained for the checkpoint message itself. A replica considers that a checkpoint is *stable* when it receives $f+1$ CHECKPOINT messages from different replicas with the same v_{last}, d and UI_{last}. This set of messages forms a *checkpoint certificate* that proves that the replica's state was correct until that request execution. Therefore, the replica can discard all entries in its log with view number $v' < v_{last}$.

Merge operation. The USIG service strongly constrains what a faulty primary can do, e.g., it cannot repeat or assign arbitrary sequence numbers to batches of messages. However, a faulty primary can still prevent progress by sending a PREPARE or a SKIP message to less than $f+1$ replicas. When $f+1$ replicas suspect that the primary is faulty, they execute a merge operation in order to make correct servers agree on the requests that were accepted and go to the next view. When a server accepts a batch of requests it increments the view number and starts a timer that expires after T_{acc}. If in the view v a server does not receive enough COMMIT messages to accept a request, and neither a SKIP message during T_{acc}, then it changes its state to merge and sends a MERGE message for view v to all servers.

A MERGE message has the format $\langle \text{MERGE}, s_j, v, C_{last}, P, O, UI_j \rangle$. Correct servers only consider MERGE messages that satisfy the following requirements: (1) the checkpoint certificate C_{last} contains $f+1$ valid UI identifiers; (2) the counter value in UI_j is $cv_j = cv+1$, where cv is the highest counter value of the UIs signed by the replica in O (if O is empty the highest counter value will be the UI in C_{last}); and (3) there are no holes in the sequence number of messages in O. The O field in the merge message prevents that a faulty server that committed one request in a view does not include that request in its merge message during the merge operation.

When the primary of the view v, s_i, receives $f+1$ MERGE messages from view $v-1$, it sends $\langle \text{PREPARE-MERGE}, s_i, v, VP, M, UI_i \rangle$ to all servers. M is used by the recipients of the message to verify if the primary computed VP correctly. When a server s_j receives a valid $\langle \text{PREPARE-MERGE}, s_i, v, VP, M, UI_i \rangle$ from s_i it changes its state to normal and sends a COMMIT message. The server increments the view v' after receiving $f+1$ valid COMMIT messages for all prepared requests VP that have not been executed before and executes them. If a replica detects that there is a gap between the sequence number of its last executed request and the first request to be executed in VP, it fetches other replicas for commit certificates of the missing requests. If, due to garbage collection, the other replicas have deleted these messages, there is a state transfer (using a protocol similar to PBFT's [2]).

Punishing Misbehavior. EBAWA avoids that faulty servers impact the performance by blacklisting them. When a server is included in the blacklist it stops becoming the primary, but it can continue participating of the algorithm. When a server receives a valid PREPARE-MERGE message in view v, this means that $f+1$ servers agreed that some problem occurred in view $v-1$ therefore they include the primary of $v-1$

in the blacklist. The size of the blacklist is bounded to f and the servers are inserted and removed following a FIFO policy. 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 servers in the list. Therefore, when a merge operation starts immediately after another one, the servers replace the last server that was inserted in their blacklist, instead of adding. That server is replaced by the server that caused the new merge operation. All servers enter normal mode in the same view, so this guarantees that all the blacklists are always consistent.

Timeout reset. The maximum time interval for a message to be accepted in a view, T_{acc}, starts with an initial value T_{start} and is multiplied by two whenever there is a merge operation. To avoid that a malicious primary forces this timeout to be high, each server halves the value of T_{acc} whenever it detects that the system is stable (if $T_{acc} > T_{start}$). The system is considered stable if after r views a server verifies that the average time to accept a request in a view (or skip it) is lower than $T_{acc}/2$ (r is a system parameter).

VI. Performance Evaluation

In this section we assess the advantages of EBAWA and the proposed mechanisms by comparing this algorithm with previous ones in several environments and conditions. Our evaluation tries to answer the following questions: (1) Does the introduction of mechanisms to make EBAWA efficient on a WAN, make it worse than other BFT algorithms on a LAN? (2) What are the benefits of the EBAWA mechanisms on WANs when clients and servers are dispersed geographically, when they are located in the same data center, and when the number of tolerated faults increases? (3) How does EBAWA compare with other BFT algorithms when they are deployed on a real WAN?

A. Algorithm Implementation

We implemented our prototypes in Java. Due to features like memory protection, strong typing and access control, Java can make a BFT implementation more dependable than another written in languages like C or C++.

We compare EBAWA with three previous BFT replication algorithms: PBFT, Spinning and MinBFT. PBFT [2] is often considered to be the baseline for BFT algorithms, so we were interested in comparing EBAWA with PBFT's C++ implementation[2]. We also implemented a version of PBFT's normal case operation in Java (JPBFT) to be able to run it in a WAN, because the original PBFT prototype does not run adequately in a WAN (it uses UDP that loses too many messages forcing the leader to change frequently, and it requires IP Multicast that is usually not available[3]). MinBFT [9] is a previous BFT algorithm that we designed with the same number of communication steps and replicas as EBAWA, but that neither rotates the primary nor uses any of the mechanisms explained in Section IV, except the USIG service. Spinning is similar to PBFT but it rotates the primary whenever agreement about the execution of a batch of messages is done [7]. We were interested in comparing EBAWA with MinBFT and Spinning once they can be considered to be improvements of PBFT, thus showing that just reducing the number of communication steps (MinBFT) or having a rotating primary (Spinning) is not enough to provide an efficient BFT algorithm for WANs. We do not compare with RAM because it is neither completely specified nor implemented yet. We also do not compare with Steward because its hierarchical architecture with two levels of protocols is completely different and has a higher overhead than our simpler client/servers architecture. Finally, we do not compare with Zyzzyva [6], that has been shown to be the most efficient BFT algorithm in LANs in several conditions, since its performance depends strongly on the latency being very stable, something that is not true in a WAN.

The prototypes JPBFT, Spinning, MinBFT and EBAWA were implemented within the same codebase and optimized to have high throughput under heavy load. In the case of the first three algorithms, we implemented adaptive batching and window congestion control schemes similar to the ones used in PBFT [2]. In all our implementations, we used TCP sockets for communication and NTT ESIGN with 2048-bit for public-key signatures, as provided by the Crypto++ library accessed through the Java Native Interface. In the LAN's machines, it takes 1.035 ms to sign and 0.548 ms to verify a message with 20 bytes (size of a request's hash).

We used the Xen hypervisor [21] to isolate the USIG service from the replica process. We run the USIG service as a daemon in a virtual machine (VM) isolated from the one in which the normal system runs. The USIG service is not connected to the network and contains as little code as possible. The counter used by the USIG service has 64 bits, which is enough to prevent it from burning out before 2^{33} years if incremented twice per millisecond. To sign messages, the USIG service uses HMACs based on SHA1 resulting in 0.008 ms to execute the `createUI` function and 0.007 ms to execute `verifyUI` function.

B. Methodology

In all experiments with Java code 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. 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 by 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. Each client sent operations periodically (without waiting for replies), in order to obtain the maximum possible throughput. Unless where noted, in

[2]Available at http://www.pmg.lcs.mit.edu/bft/.

[3]The original PBFT prototype uses IP Multicast with two purposes: for communication among servers and for controlling the execution of experiments. The first can be switched off (IP used instead) but the latter can not.

experiments with EBAWA, clients evenly distributed their requests among servers and each server was able to start 10 agreements asynchronously ($W = 10$).

C. Local Area Network

Setup. In order to compare the performance of EBAWA with PBFT we conducted experiments in a LAN without faults or instability in the network. To measure the throughput, we executed from 0 to 120 logical clients distributed over 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 gigabit switch. To measure latency and throughput we implemented two versions of EBAWA. In EBAWA-S the clients send requests signed using public-key cryptography. In EBAWA-V clients sign requests with authenticators (vectors of MACs), as done in other algorithms [2], [9], [7]. We consider a setup that can tolerate one faulty server ($f = 1$), with $n = 4$ servers for PBFT, Spinning and JPBFT and $n = 3$ servers for MinBFT and EBAWA, unless where noted. Each experiment ran for 100000 client requests to allow performance to stabilize, before recording data for the following 100000 operations.

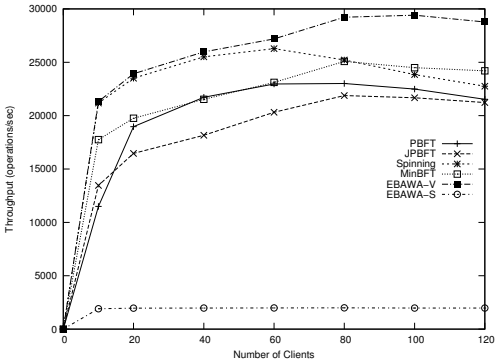
Latency. Figure 2(a) shows the latency of the algorithms for requests and responses of size 0 and 4K. PBFT has shown the best performance of all algorithms/implementations (0.4 ms with 0/0), followed by Spinning (1.3 ms with 0/0) and JPBFT (1.8 ms with 0/0). This experiment shows clearly that our Java implementation runs an agreement much slower than PBFT, although they run the same number of communication steps. One of the possible reasons for this is 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, which allows the replicas to send a reply before executing the commit step, as originally proposed in [2].

Even with fewer communication steps, the latencies of MinBFT and EBAWA in a LAN are higher than those of the other algorithms due to the overhead to access the trusted component in an isolated VM. EBAWA-V presented the best latency of the three (2.1 ms with 0/0), followed by MinBFT (2.3 ms with 0/0) and EBAWA-S (2.5 ms with 0/0). The worst latency of EBAWA-S is justified by the fact that the client signs requests and each server needs to verify if the client's signature is valid.

Throughput. The throughput results in a LAN are presented in Figure 2(b). The main conclusion is that EBAWA-V presents the best throughput in fault-free environments, followed by Spinning, MinBFT, PBFT and JPBFT. The peak throughput of EBAWA-V is 22% better than PBFT's peak throughput. These results can be explained by two factors: (1) the introduction of asynchronous views as explained in the Section IV, allowing piggybacking of PREPARE and SKIP messages on COMMIT messages, reducing the number of communication steps and the cryptographic operations executed by servers (only one unique identifier per message); (2) the rotation of the primary providing a better load balancing among the servers. In PBFT, the throughput of the system is constrained by the amount of messages per batch processed by the primary, which is $5n$, while 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 primary executes 20% more work than other servers.

Req/Resp size (Kb)	PBFT	JPBFT	Spinning	MinBFT	EBAWA-V	EBAWA-S
0/0	0.4	1.8	1.3	2.3	2.1	3.5
0/4	0.6	2.2	1.7	2.9	2.3	4.0
4/0	0.8	2.5	2.1	3.1	2.6	4.1

(a) Latency (ms)



(b) Throughput (ops/sec)

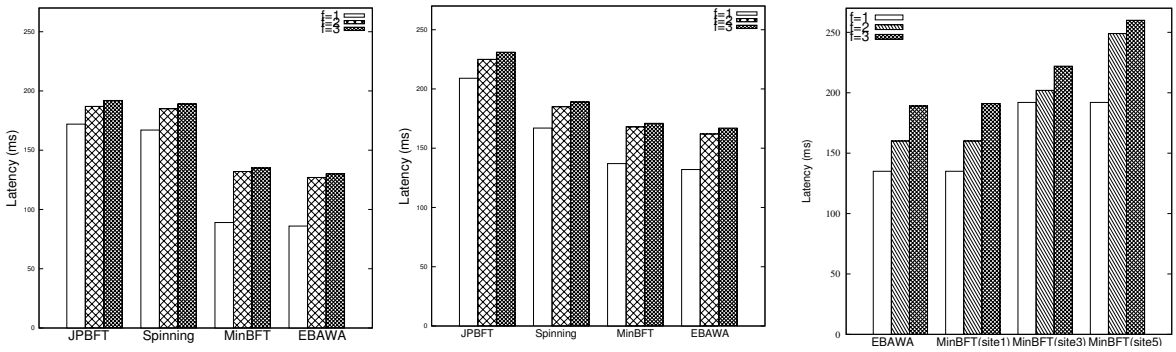
Figure 2. Latency and throughput of algorithms on fault-free LAN.

Figure 2(b) shows also that the verification of the clients' signatures limits the peak throughput of EBAWA-S to about 2000 operations per second, which is exactly what the machine processor is capable of, since it takes approximately 0.5 ms to verify a signature ($1/(0.5 \times 10^{-3}) = 2000$). However, the use of public-key signatures instead of authenticators provides non-repudiation, ensuring that all correct replicas take the same decisions about the validity of clients' requests [4]. The use of message authenticators can be faster than signatures but lets the algorithm vulnerable to malicious clients that can force primary changes in PBFT or merge operations in Spinning and EBAWA.

As a side note, this same signature verification takes 0.128 ms on a 64-bit 2.3GHz quadcore Xeon machine, which allows the execution of $7812 \times 4 = 31248$ verifications per second. This shows that algorithms based on public-key cryptography can be used successfully in high-end servers, especially if the power of multi-core architectures is exploited to verify signatures in parallel. This possibility and the greater robustness of EBAWA-S lead us to use this version of EBAWA in the WAN experiments (next sections).

D. Emulated WAN – Emulab

Setup. We conducted experiments on an emulated WAN to measure the latency of the algorithms when the replicas are scattered through different data centers connected by dedicated links. From now on, our focus is on the algorithms' latency, because low latency is intrinsically difficult to achieve in a WAN, while throughput can be increased with faster hardware and by using channels with higher bandwidth. The experiments



(a) Clients and servers in datacenters (b) Scattered clients (c) Primary location

Figure 3. Latency on a emulated fault-free WAN. (a) Clients in the same sites of the servers. (b) Clients and servers distributed on a WAN. (c) Heterogeneous latency among clients and servers.

were conducted on Emulab [22], using 20 Intel Pentium III machines with 600 Mhz processors, 256 Mb of RAM and Red-Hat Linux 9. The JVM was Sun JDK1.5.11. Each link was configured with 20Mbps of bandwidth and the server-to-server latency was configured to 40 ms. The latency between clients and servers is different on each experiment. Each experiment was executed during 30 minutes and we consider setups with different numbers of replicas (f=1, 2 and 3). Recall that in the WAN experiments we used our own PBFT prototype (JPBFT) instead of the original PBFT prototype since the latter uses UDP that loses too many messages (forcing the leader to change frequently) and requires IP Multicast (usually not available in WANs).

Clients located in the same site as servers. The first case we consider is when each site has a set of clients and one replica of the service. This setup mimics a common deployment in real data centers, which is considered by previous BFT algorithms for WANs [11], [12]. In EBAWA, clients send their requests to the local replica. In MinBFT and JPBFT we consider that the client and the primary server are in the same site (which is the *best possible case*). In Spinning, due to the rotation of the primary, the replica near the client becomes the primary periodically. Clients are in the same network than servers therefore their latency to access a server is about 0.2 ms.

Figure 3(a) presents the latency of the algorithms. The figure shows similar results for EBAWA and MinBFT, and these results are about 30% better than the latency presented by Spinning and JPBFT due to the lower number of communication steps of the former. With clients in the same data center, JPBFT presents a latency better than expected (REQUEST and PREPARE messages are received almost at the same time by replicas). Spinning presents the same average latency of JPBFT because in Spinning clients send requests to all servers and each server is constantly becoming the primary, so the fact that the client is placed with a replica has no impact.

Clients scattered geographically. Our second WAN experiment considers a client-replicas link latency of 40 ms to represent clients accessing replicas on different countries. Figure 3(b) shows the results of this experiment.

Under such homogeneous setting, the latency observed directly reflects the amount of communication steps required by each algorithm: EBAWA presents almost the same latency of MinBFT, which is better than Spinning's and JPBFT's. Spinning presents better latency than JPBFT due to the aforementioned tentative execution optimization.

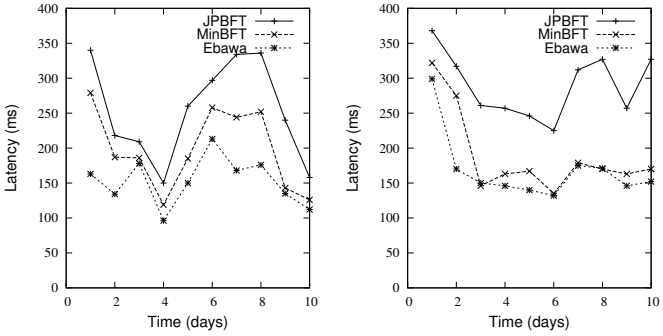
Missing communication step. When we conducted experiments with $f = 1$ we observed that, surprisingly, MinBFT and EBAWA showed a latency corresponding to 3 communication steps, which contradicts the theoretical 4 communication steps of these algorithms (their latency should be at least $40 \times 4 = 160$ ms). The explanation for this discrepancy highlights an interesting advantage of these algorithms. In a setup with $f = 1$ in which the network latency is stable, replicas receive the PREPARE and COMMIT messages from the primary almost together (the primary "sends" the PREPARE to itself and sends its COMMIT immediately). Since, the two algorithms need only $f + 1$ COMMIT messages to accept a request, with $f = 1$ these are only two. These two COMMITs would be received just after the PREPARE: one from the primary and another from the replica itself. Therefore, the client request is executed in a replica as soon as the PREPARE message from the primary arrives, making MinBFT and EBAWA have latencies 18% and 35% better than Spinning and JPBFT, respectively. In setups with $f > 1$ this nice feature disappears since the quorum to accept the request has to contain at least 3 replicas, making MinBFT and EBAWA present latencies 11% and 27% better than Spinning and JPBFT, respectively. It explains the significant latency increase on setups of $f = 1$ to $f = 2$ (+23%), which is not observed from $f = 2$ to $f = 3$ (+3%).

Primary location. In the previous experiments we observed that EBAWA and MinBFT present almost the same latency when the client is near to the primary and the environment is homogeneous. In this experiment we try to answer the following question: can this equivalence hold in environments where the client-replica latency is heterogeneous?

In these experiments we observed that the location of the primary replica affects significantly the service latency in WANs (when the primary is fixed). For MinBFT, we defined

WAN-Europe		WAN-America	
Portugal - France	41.03	Vancouver - Pennsylvania	94.57
France - Italy	27.3	Michigan - Pennsylvania	48.14
France - Germany	30.43	California - Pennsylvania	91.57
Italy - Germany	29.3	Vancouver - Michigan	74.48
Portugal - Italy	53.96	Michigan - California	80.76
Portugal - Germany	48.5	Vancouver - California	55.52
Spain - Portugal	46.87	New York - Pennsylvania	39.14
Spain - Germany	47.62	New York - Vancouver	72.57
Spain - Italy	45.75	New York - Michigan	48.57
Spain - France	28.87	New York - California	88.71
Average	39.96		69.40

(a) Network Latency



(b) WAN-Europe (c) WAN-America

Figure 4. Network latency on a real WAN in America/Europe (a) and algorithms' latencies in WAN-Europe (b) and WAN-America (c).

that the primary server can be located in three different sites ($site_1$,$site_3$ and $site_5$) and between each site and client we defined different latencies as presented in Table II. Figure 3(c) presents the results of the experiment. In EBAWA clients always send the request to the nearest server located in $site_1$. When the primary is located near the client ($site_1$), MinBFT presents the same latency as EBAWA, as we have seen in previous experiments. Otherwise, when we vary the location of the primary we have a significantly increase on MinBFT's latency. In MinBFT for $f = 1$, due to the case of the missing step, we have the same latency when the primary is in the $site_3$ or $site_5$ that is 30% greater than EBAWA's latency. For $f = 2$, MinBFT presents a latency 20% and 35% greater than EBAWA when the primary is located in $site_3$ and $site_5$, respectively. This difference in the algorithms' latencies is kept with $f = 3$. This experiment shows the benefit of implementing a BFT algorithm with a rotating primary and making clients choose the nearest server as their primary.

	$site_1$	$site_2$	$site_3$	$site_4$	$site_5$	$site_6$	$site_7$
latency	25 ms	40 ms	55 ms	75 ms	95 ms	75 ms	75 ms

Table II
LATENCY AMONG CLIENTS AND REPLICAS ON DIFFERENT SITES.

Load Balancing. When servers are replicated on a WAN, the load of requests generated by clients can change following an unpredictable pattern. We conducted an experiment with two client-server pairs, each pair located at a different site. Each client generates requests during 20 minutes and measures the latency. The client at data center *A* generates requests during the first 10 minutes obtaining an average latency of 86 ms. Then, the client of data center *B* generates requests in parallel with *A*'s client. Both sites registered an average latency of 89 ms. Finally, the experiment finish with *B*'s client generating requests during the last 10 minutes and registering an average latency of 84 ms. The latencies per request are presented in Figure 5. This experiment shows that when the load of client requests increases (duplicates in this case), EBAWA promptly adjusts to the new load assuring a minimum impact in the service latency. This happens because a replica sends *skip* messages (see Section IV) to give its turn to other replicas when it has no client requests to order; and because of the asynchronous views (see Section IV) that allow a replica to send a set of COMMIT messages for different pending PREPAREs in a single one, reducing communication overhead on the system.

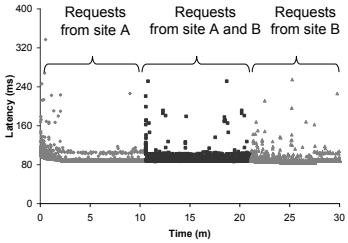


Figure 5. Service latency when the load changes from site A to B.

E. Real WAN – PlanetLab

Setup. In order to assess EBAWA's latency on real WANs, where both processing and network resources are unpredictable, we conducted a set of experiments on PlanetLab[4]. We used two setups called WAN-Europe and WAN-America, representing configurations with replicas scattered through different locations in Europe and America, respectively. Together with the protocols latency we also registered the communication latency between each pair of servers.The observed average latency varied between 27 ms to 94 ms as presented in Figure 4(a). We consider a setup with $f = 1$ with three (EBAWA and MinBFT) and four (JPBFT) servers and a client. In WAN-Europe, the client was located in Portugal and the servers in France, Italy, Germany and Spain (for JPBFT). The primary replicas were at Germany for JPBFT and MinBFT. In WAN-America, the client was located at Pennsylvania and the servers at Vancouver, California, Michigan and New York (for JPBFT). The primary replica was located in Michigan for JPBFT and MinBFT. We measured the latency of these algorithms during 10 days, 3 times per day, and the reported values are the average of the latency measurements made each day. In each experiment the client submitted 10000 requests.

Results. Figure 4(b) and (c) shows the latency of EBAWA, MinBFT and JPBFT in WAN-Europe and WAN-America. The figure shows that EBAWA consistently outperforms MinBFT and JPBFT on real networks with *heterogeneous* latency

[4]http://www.planet-lab.org/.

among servers. Not surprisingly, EBAWA and MinBFT present a better latency than JPBFT due to the reduced number of communication steps. The most significative result in WAN-Europe was when the average network latency was 50 ms, when EBAWA presented an average latency 31% lower than MinBFT and 48% lower than JPBFT. In the WAN-America experiments, when the average network latency was 60 ms, EBAWA presented an average latency 10% lower than MinBFT and 43% lower than JPBFT.

The closer results of MinBFT and EBAWA in WAN-America can be explained because in this scenario we located the primary replica of MinBFT in Michigan that is the site with the best average latency to the client, hosted in Pennsylvania.

The better latency of EBAWA in these experiments can be explained mainly by two factors: (1) the client in EBAWA always sends the requests to the *nearest* server (the server that present best latency to that client). In JPBFT and MinBFT, the client sends requests to all servers and the primary may not have the best latency to that client. Therefore this delays the ordering of the client requests and affects the end-to-end latency. We expect that the advantage of EBAWA would be even greater if we used larger request/responses. (2) The PREPARE message of EBAWA carries the REQUEST message, consequently, when a server accepts a request it can execute the request immediately (and return a reply to the client). In MinBFT and JPBFT, the PREPARE and PRE-PREPARE messages, respectively, sent by the primary server of these algorithms, contain only a *hash* of the original request. We observed that with the link heterogeneity of a real WAN, it is common that a server receives a PREPARE message from the primary before it has received the request from the client. Then, it cannot execute the request until it receives the message from the client or from another server, affecting the algorithm latency. Thus, our observation is that in a WAN, it is important to send requests in the PREPARE message.

VII. Conclusion

This paper presents and evaluates EBAWA, a novel BFT algorithm for WANs. When compared with other BFT algorithms for WANs, EBAWA reduces the numbers of communication steps and replicas by assuming a hybrid fault model with a minimal trusted service. This reduced number of replicas and communication steps among servers increases considerably the system's performance. EBAWA is more flexible in the sense that clients and servers can be located in the same site or dispersed geographically. A thorough performance evaluation of EBAWA has shown that it is competitive with other BFT algorithms in LANs, while outperforming all of them in several WAN settings, especially with the higher heterogeneity of a real WAN.

Acknowledgments. This work was partially supported by Alban scholarship E05D057126BR, by the EC through FP7-ICT project TCLOUDS, and by the FCT through project PTDC/EIA-IA/100581/2008 (REGENESYS), project PTDC/EIA-EIA/109044/2008 (RED) and the Multiannual and the CMU-Portugal Programmes.

References

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

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[3] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Proc. Intl. Conf. on Dependable Systems and Networks*, Jun. 2008, pp. 197–206.

[4] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *Proc. Symp. on Networked Systems Design & Implementation*, Apr. 2009.

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proc. Symp. on Oper. Systems Design and Implementations*, Nov. 2006, pp. 177–190.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *Proc. Symp. on Oper. Systems Principles*, Oct. 2007.

[7] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proc. Symp. on Reliable Distributed Systems*, Sep. 2009.

[8] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proc. Symp. on Operating Systems Design and Implementation*, Dec. 2008, pp. 369–384.

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

[10] L. Lamport, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.

[11] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Scaling Byzantine fault-tolerant replication to wide area networks," in *Proc. Intl. Conf. on Dependable Systems and Networks*, Jun. 2006.

[12] 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*, Jun. 2009.

[13] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in *Proc. Symposium on Operating Systems Principles*, Oct. 2007.

[14] M. Correia, N. F. Neves, and P. Verissimo, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *Proc. Symp. on Reliable Distributed Systems*, Oct. 2004, pp. 174–183.

[15] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: small trusted hardware for large distributed systems," in *Proc. Symp. on Networked Systems Design and Implementation*, 2009, pp. 1–14.

[16] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[17] P. Verissimo, "Travelling through wormholes: A new look at distributed systems models," *SIGACT News*, vol. 37, no. 1, pp. 66–81, 2006.

[18] Trusted Computing Group, "TPM Main, Part 1 Design Principles. Specification Version 1.2, Revision 62," 2003.

[19] B. Littlewood and L. Strigini, "Redundancy and diversity in security," in *Proc. European Symp. on Research Computer Security*. Springer, 2004.

[20] G. S. Veronese, M. Correia, A. Bessani, and L. Chung, "Efficient Byzantine agreement for wide area networks (extended version)," Tech. Rep., Dec. 2009, http://homepages.di.fc.ul.pt/∼mpc/ebawa-long.pdf.

[21] P. Barham, B. Dragovic, K. Fraiser, S. Hand, T. Harris, A. Ho, R. Neugebaurer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. Symp. on Operating Systems Principles*, Oct. 2003, pp. 164–177.

[22] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. Symp. on Oper. Systems Design and Implementation*, Dec. 2002, pp. 255–270.