Efficient Byzantine Fault Tolerance

Giuliana Santos Veronese, Miguel Correia *Member, IEEE,*
Alysson Neves Bessani, Lau Cheuk Lung and Paulo Verissimo, *Fellow, IEEE*

Abstract—We present two asynchronous Byzantine fault-tolerant state machine replication (BFT) algorithms, which improve previous algorithms in terms of several metrics. First, they require only $2f+1$ replicas, instead of the usual $3f+1$. Second, the trusted service in which this reduction of replicas is based is quite simple, making a verified implementation straightforward (and even feasible using commercial trusted hardware). Third, in nice executions the two algorithms run in the minimum number of communication steps for non-speculative and speculative algorithms, respectively 4 and 3 steps. Besides the obvious benefits in terms of cost, resilience and management complexity — fewer replicas to tolerate a certain number of faults — our algorithms are simpler than previous ones, being closer to crash fault-tolerant replication algorithms. The performance evaluation shows that, even with the trusted component access overhead, they can have better throughput than Castro and Liskov's PBFT, and better latency in networks with non-negligible communication delays.

Index Terms—Byzantine fault tolerance, intrusion tolerance, state machine replication, distributed systems, trusted components

1 Introduction

The complexity and extensibility of current computer systems have been causing a plague of exploitable software bugs and configuration mistakes. Accordingly, the number of cyber-attacks has been growing, confirming computer security as an increasingly important research challenge. As an approach to meet this challenge, several *asynchronous Byzantine fault-tolerant algorithms* have been proposed[1]. The main idea of these algorithms is to allow a system to automatically continue operating correctly, even if some of its components exhibit arbitrary, possibly malicious behavior [4], [12], [13], [15], [17], [19], [26], [29], [30], [39], [49]. These algorithms have already been used to design *intrusion-tolerant* services such as network file systems [12], [14], [49], cooperative backup [3], large scale storage [2], DNS [11], coordination services [8], [14], certification authorities [50], databases [21], and key management systems [40].

Byzantine fault-tolerant systems are often built using replication. The *state machine approach* is a generic replication technique to implement deterministic fault-tolerant services. It has been used as a means to tolerate both crash and Byzantine/arbitrary faults [42], [12], [39]. The algorithms used in the second case are usually called simply BFT. There are, however, other algorithms in the literature that are Byzantine fault-tolerant, but provide weaker semantics, e.g., registers implemented with quorum systems [31]. When we speak about BFT in the paper, we do not include these.

• *Giuliana Santos Veronese, Miguel Correia, Alysson Bessani, Paulo Verissimo are with the Departmento de Informática, Faculdade de Ciências, Universidade de Lisboa, Campo Grande, C6, Lisboa, Portugal. Email: giuliana@lasige.di.fc.ul.pt, {bessani,mpc,pjv}@di.fc.ul.pt*
• *Lau Cheuk Lung is with Departamento de Informática e Estatística, Centro Tecnológico, Universidade Federal de Santa Catarina, Brazil. Email: lau.lung@inf.ufsc.br*

1. Although the term asynchronous is often used in this context, most of these algorithms are not strictly asynchronous, but more precisely do not rely on synchrony for correctness. Most of them make weak time assumptions like the ones we make in Section 3.

This paper presents two novel BFT algorithms. Their novelty comes from these algorithms improving previous ones in terms of three metrics: *number of replicas*, *trusted service simplicity* and *number of communication steps*. We say that these algorithms are *efficient*, in this sense that they are equal to or better than previous algorithms in terms of these metrics. Efficiency in this case comes from the use of a simple trusted component. This means that the system model we consider is slightly different from those of most previous works, but the difference is worthwhile, since the latter present algorithms that are already optimal in the system model they consider. We explain this efficiency in the following paragraphs.

Number of replicas. BFT algorithms typically require $3f+1$ *servers* (or *replicas*[2]) to tolerate f Byzantine (or *faulty*) servers [12], [17], [26], [39]. They are based on the idea that correct replicas can overcome faulty replicas in a sequence of votes and, for this to happen, they would need at least $f+1$ correct replicas, in a total of $2f+1$ servers. However, these algorithms require f more servers than this minimum. The only two algorithms that require just $2f+1$ servers are those proposed by Chun et al. [13] and Correia et al. [15].

Reducing the number of replicas has an important impact in the cost of intrusion-tolerant systems as one replica is far more costly than its hardware. For tolerating attacks and intrusions, the replicas can not be identical and share the same vulnerabilities, otherwise causing intrusions in all the replicas would cost almost the same as in a single one. Therefore, there has to be *diversity* among the replicas, i.e., replicas shall have different operating systems, different application software, etc. [20], [34]. This involves considerable additional costs per-replica, in terms not only of hardware but especially of software development, acquisition, and management. There are also complementary solutions that can create some levels of diversity automatically, like address space layout randomization [48] and

2. We use the two words interchangeably, since servers are used exclusively as replicas of the service they run.

instruction set randomization [6].

The first sense in which the BFT algorithms presented in the paper are said to be efficient is that they require only $2f + 1$ replicas, which is clearly the minimum for BFT algorithms, since a majority of the replicas must be non-faulty for majority voting to be applied [42]. It also matches the lower bound for non-synchronous crash fault tolerance [18].

Trusted service simplicity. A few years ago, a BFT algorithm that needs only $2f + 1$ replicas was published [15]. This algorithm requires that the system is enhanced with a tamperproof distributed component called Trusted Timely Computing Base (TTCB). The TTCB provides an *ordering service* used to implement an atomic multicast protocol that is the core of the replication scheme. Recently, another BFT algorithm with only $2f + 1$ replicas was presented, A2M-PBFT-EA [13]. It is based on an *Attested Append-Only Memory* (A2M) abstraction that has to be tamperproof, like the TTCB, but local to each computer, instead of distributed.

These two works have shown that one can reduce the number of replicas from $3f + 1$ to $2f + 1$ by extending the servers with tamperproof components, i.e., with components that provide a correct service even if the server where they are installed becomes faulty. Therefore, an important aspect of the design of $2f+1$ BFT algorithms is the design of these components so that they can be trusted to be tamperproof. This problem is not novel for it is similar to the problem of designing a Trusted Computing Base or a reference monitor. A fundamental goal is to design the component in such a way that it is *verifiable*, which requires simplicity (see for example [23]). Despite this need of simplicity, the TTCB is a distributed component that provides several services and A2M provides a log that can grow considerably and an interface with functions to append, lookup and truncate messages in the log.

The second sense in which the algorithms presented in this paper are said to be efficient is that the trusted/tamperproof service in which they are based is simpler that the previous two in the literature (TTCB, A2M) and is the simplest that we can conceive. Let us explain this last statement. Some papers prove that it is not possible to design specific Byzantine fault-tolerant agreement algorithms with fewer than $3f + 1$ servers [9], [28]. Looking into these proofs it can be seen that the main problem is that a malicious server can lie to the correct ones. In case of state machine replication, the main agreement problem is to make all correct replicas execute the same sequence of operations. In this sense, a trusted service has to provide at least something that gives to replicas the notion of a sequence of operations, in such a way that a malicious replica would not be able to make different correct replicas execute different operations as their i-th operation. It is not difficult to see that nothing is simpler than a trusted monotonic counter, used to associate sequence numbers to each operation (the state is only one natural number). On the other hand, the values generated by this counter should be unforgeable, so some kind of authentication based on cryptographic primitives must be employed. The trusted service presented in this paper (USIG) provides an interface with operations only to increment a counter and to verify if other counter values (incremented by other replicas) are correctly authenticated.

An important side effect of the simplicity of our trusted component is that it can be implemented even on COTS trusted hardware, such as the *Trusted Platform Module* (TPM) [36]. This secure co-processor is currently available in the mainboard of many commodity PCs.

Number of communication steps. The number of communication steps is an important metric for distributed algorithms, for the delay of communication tends to have a major impact in the latency of the algorithm. This is specially important in WANs, where the communication delay can be as much as a thousand times higher than in LANs. Moreover, to tolerate disasters and large-scale attacks like DDoS, replicas have to be deployed in different sites, which increases the message communication delays.

The first algorithm we propose – MinBFT – follows a message exchange pattern similar to PBFT's [12]. The replicas move through a succession of configurations called views. Each view has a primary replica and the others are backups. When a quorum of replicas suspects that the primary replica is faulty, a new primary is chosen, allowing the system to make progress. In each view there are communication steps in which the primary sends messages to all backups, and steps in which all replicas send messages to all others. The fundamental idea of MinBFT is that the primary uses the trusted counters to assign sequence numbers to client requests. However, more than assigning a number, the tamperproof component produces a signed certificate that proves unequivocally that the number is assigned to that message (and not another) and that the counter was incremented (so the same number can not be used twice). This is used to guarantee that all non-faulty replicas consider that all messages with a certain identifier are the same and, ultimately, agree on the same order for the execution of the requests.

The second algorithm we propose – MinZyzzyva– is based on *speculation*, i.e., on the tentative execution of the clients' requests without previous agreement on the order of their execution. MinZyzzyva is a modified version of Zyzzyva, the first speculative BFT algorithm [26]. The Zyzzyva communication pattern is similar to PBFT's except for speculation: when the backups receive a request from the primary, they speculatively execute the request and send a reply to the client.

For BFT algorithms, the metric considered for latency is usually the number of communication steps in nice executions, i.e., when there are no failures and the system is synchronous enough for the primary not to be changed. MinBFT and MinZyzzyva are very efficient in terms of this metric, because in nice executions the two algorithms run in the minimum known number of communication steps of non-speculative and speculative algorithms, respectively 4 [30] and 3 steps [26]. Notice that the gain of one step in speculative algorithms comes at a price: in certain situations Zyzzyva and MinZyzzyva may have to

		PBFT [12] (+[49])	Zyzzyva [26]	TTCB [15]	A2M-PBFT-EA [13]	MinBFT this paper	MinZyzzyva this paper
Model	Tamperproof component	no	no	TTCB	A2M	USIG	USIG
Speculative		no	yes	no	no	no	yes
Cost	Total replicas	$3f+1$	$3f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$
	Replicas with application state	$2f+1$ [49]	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$
Throughput	HMAC ops at bottleneck server	$2+(8f+1)/b$	$2+3f/b$	3	$2+(2f+4)/b$	$2+(f+3)/b$	$2+sign/b$ (†)
Latency	Communication steps	5 / 4	3	5	5	4	3

Table 1
Comparison of BFT algorithms, expanding Table 1 in [26]. The throughput and latency metrics are for each request. f is the maximum number of faulty servers and b the size of the batch of requests used. (†) MinZyzzyva does 2 HMAC operations and one signature.

rollback some executions, which makes the programming model more complicated.

Benefits and drawbacks. Table 1 presents a summary of the characteristics of the algorithms presented in the paper and the state-of-the-art algorithms in the literature with which they are compared. It summarizes the benefits in terms of the above mentioned metrics.

The benefits in terms of number of replicas and communication steps come from the use of the trusted service. A core problem in Byzantine fault-tolerant algorithms is duplicity, i.e., the possibility of a faulty server sending inconsistent messages to different servers [28]. More precisely, a faulty server may send two messages with different content but the same identifier to two different subsets of servers. Tolerating this behavior requires, e.g., that every subset of $n-f$ servers (because f may be crashed) contains a majority of correct servers, which leads to a minimum of $3f+1$ servers. Our trusted service, USIG, can prevent this duplicity because it is used to assign identifiers to messages and never assigns the same identifier to two different messages. When it assigns an identifier, it provides also a certificate that is appended to the message, proving that the identifier was actually given by the service.

It is important to understand that that these improvements have a downside, there are no free lunches. In relation to BFT algorithms that do not use a trusted component, our algorithms (as the previous two [13], [15]) have an additional point of failure: the assumption of the tamperproofness of the component. With this regard, we stress again the simplicity of the USIG trusted service used in this paper, making a verified implementation or the use of commercial trusted hardware (e.g., TPM) straightforward. In consequence, and given a robust design of those components, this is only a disadvantage in settings where the potential attacker has physical access to a replica, since protecting even hardware components from physical attacks is at best complicated.

Contributions. The contributions of the paper can be summarized as follows:

- it presents two BFT algorithms that match or improve previous algorithms in terms of number of replicas (only $2f+1$), complexity of the trusted service used, and number of communication steps (4 and 3 respectively without/with speculation); it also shows that, even with the trusted component access overhead, these algorithms can have better throughput than Castro and Liskov's PBFT, and better latency in networks with non-negligible communication delays;
- it presents the first implementations with some level of isolation for a trusted component used to improve BFT algorithms. We implemented several versions of the USIG service with different cryptography mechanisms that are isolated both in separate virtual machines and trusted hardware.

The characteristics of the presented BFT algorithms in terms of number of replicas, trusted service simplicity and number of steps, leads to a simplicity that we believe makes them practical to a level only comparable with crash fault-tolerant algorithms.

2 USIG Service

The *Unique Sequential Identifier Generator (USIG)* is a *local service* that exists in every server. It assigns to messages (i.e., arrays of bytes) the value of a counter and signs it. Identifiers are unique, monotonic and sequential for that server. These three properties imply that the USIG (1) will never assign the same identifier to two different messages (uniqueness), (2) will never assign an identifier that is lower than a previous one (monotonicity), and (3) will never assign an identifier that is not the successor of the previous one (sequentiality). These properties are guaranteed even if the server is compromised, so the service has to be implemented in a tamperproof module.

The interface of the service has two functions:

- `createUI`(m) – returns a *USIG certificate* that contains a *unique identifier UI* and certifies that this UI was created by this tamperproof component for message m. The unique identifier is essentially a reading of the monotonic counter, which is incremented whenever `createUI` is called.
- `verifyUI`(PK, 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.

There are two basic options to implement the service, depending on the certificate being based on cryptographic hashes or public-key cryptography:

- *USIG-Hmac*: a certificate contains a Hash-based Message Authentication Code (HMAC) [27] 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.

- *USIG-Sign*: the certificate contains a signature obtained using the message and the private key of this USIG.

In USIG-Hmac the properties of the service (e.g., uniqueness) are based on the secretness of the shared keys, while in USIG-Sign the properties are based on the secretness of the private keys. Therefore, while for USIG-Hmac both functions `createUI` and `verifyUI` must be implemented inside the tamperproof component, for USIG-Sign the verification requires only the public-key of the USIG that created the certificate, so this operation can be done outside of the component. In both cases, keys have to be shared for the verification to be done in servers other than the one where `createUI` was called.

The implementation of the service is based on an isolated, tamperproof, component that we assume can not be corrupted. This component contains essentially a counter and either an HMAC primitive (for USIG-Hmac) or a digital signature primitive (for USIG-Sign). More details about the USIG implementation can be found in Section 6.

3 System Model and Properties

The system is composed by a set of n servers $P = \{s_0, ..., s_{n-1}\}$ that provides a Byzantine fault-tolerant service to a set of clients. Clients and servers are interconnected by a network and communicate only by message-passing.

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 in order to be retransmitted. An attacker may have access to the network and be able to modify messages, so messages contain digital signatures or HMACs. Servers and clients know which keys they need, in order to check these signatures/HMACs. We make the standard assumptions about cryptography, i.e., that hash functions are collision-resistant and that signatures can not be forged.

Servers and clients are said to be either *correct* or *faulty*. Correct servers/clients always follow their algorithm. On the contrary, 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 $n = 2f + 1$. In practice, and in order to prevent early resource exhaustion by common-mode attacks, this requires that the servers be diverse [20], [34]. Notice that we are not considering the generic case ($n \geq 2f + 1$) but the tight case in which the number of servers n is the minimum for a value of f, i.e., $n = 2f + 1$. This restriction is known to greatly simplify the presentation of the algorithms, which are simple to modify to the generic case.

Each server contains a local trusted/tamperproof component that provides the USIG service. Therefore, the fault model we consider is *hybrid* [46], with the hybrid behavior enforced by the architecture. In fact, whilst we state that any number of clients and any f servers can be faulty in a Byzantine manner, there is a set of components of the system architecture which are tamperproof: each of these components implements the USIG service, which always satisfies its specification, even if hosted in a faulty server. For instance, a faulty server may decide not to send a message or send it corrupted, but it can not send two different messages with the same value of the USIG's counter and a correct certificate.

We do not make assumptions about processing or communication delays, except that these delays do not grow indefinitely (like in [12]). This rather weak assumption has to be satisfied only to ensure the liveness of the system, not its safety. This can be confirmed by following the proofs in the supplemental material.

Properties. The algorithms presented in the following two sections implement the state machine approach, which consists of replicating a service in a group of servers and maintaining a strong consistency amongst them. Each server maintains a set of *state variables*, which are modified by a set of *operations*. These operations have to be atomic (they can not interfere with other operations) and deterministic (the same operation executed in the same initial state generates the same final state), and the initial state must be the same in all servers. The properties that the algorithm has to enforce are: *safety* – all correct servers execute the same requests in the same order; *liveness* – all correct clients' requests are eventually executed.

4 MinBFT

This section presents MinBFT, the non-speculative $2f + 1$ BFT algorithm. MinBFT follows a message exchange pattern similar to PBFT's (see Figure 1). The servers move through successive configurations called *views*. Each view has a *primary* replica and the rest are *backups*. The primary is the server s_p with $p = v \ mod \ n$, where v is the current view number. Clients issue *requests* with operations.

In the *normal case operation* the sequence of events is the following: (1) a client sends a request to all servers; (2) the primary assigns a *sequence number* (execution order number) to the request and sends it to all servers in a PREPARE message; (3) each server multicasts a COMMIT message to other services when it receives a PREPARE from the primary; (4) when a server *accepts* a request, it executes the corresponding operation and returns a reply to the client; (5) the client waits for $f + 1$ matching replies for the request and *completes* the operation.

When $f + 1$ backups suspect that the primary is faulty, a *view change operation* is executed, and a new server s'_p with $p = v' \ mod \ n$ becomes the primary ($v' > v$ is the new view number). This mechanism provides liveness by allowing the system to make progress when the primary is faulty.

Clients. A client c requests the execution of an operation op by sending a message $\langle \text{REQUEST}, c, seq, op \rangle_{\sigma_c}$ to all servers. seq is the request identifier that is used to ensure exactly-once semantics: (1) the servers store in a vector V_{req} the seq of the latest request they executed for each client; (2) the servers discard requests with seq lower than

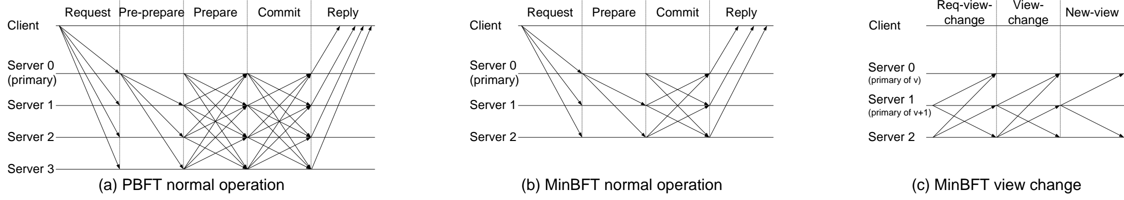

Figure 1. Message patterns of PBFT and MinBFT.

the latest executed (to avoid executing the same request twice), and any requests received while the previous one is being processed. Requests are signed with the private key of the client. Requests with an invalid signature σ_c are simply discarded. After sending a request, the client waits for $f+1$ replies $\langle \text{REPLY}, s, seq, res\rangle$ from different servers s with matching results res, which ensures that at least one reply comes from a correct server. If the client does not receive enough replies during a time interval read in its local clock, it resends the request. In case the request has already been processed, the servers resend the cached reply.

Servers: normal case operation. The core of the algorithm executed by the servers is the PREPARE and COMMIT message processing (see Figure 1). MinBFT has only two communication steps, not three like PBFT or A2M-PBFT-EA. When the primary receives a client request, it uses a PREPARE message to multicast the request to all servers. The main role of the primary is to assign a *sequence number* to each request. This number is the counter value returned by the USIG service in the unique identifier UI. These numbers are sequential while the primary does not change, but not when there is a view change, an issue that we discuss later.

The basic idea is that a request m is sent by the primary s_i to all servers in a message $\langle \text{PREPARE}, v, s_i, m, UI_i\rangle$, and each server s_j resends it to all others in a message $\langle \text{COMMIT}, v, s_j, s_i, m, UI_i, UI_j\rangle$, where UI_j is obtained by calling `createUI`. Each message sent, either a PREPARE or a COMMIT, has thus a unique identifier UI obtained by calling the `createUI` function, so no two messages can have the same identifier. Servers check if the identifiers of the messages they receive are valid for these messages, using the `verifyUI` function.

If a server s_k did not receive a PREPARE message but received a COMMIT message with a valid identifier generated by the sender, then it sends its COMMIT message. This can happen if the sender is faulty and does not send the PREPARE message to server s_k (but sends it to other servers), or if the PREPARE message is simply late and is received after the COMMIT messages. A request m is *accepted* by a server following the algorithm if the server receives $f+1$ valid COMMIT messages from different servers for m.

This core algorithm has to be enhanced to deal with certain cases. A correct server s_j multicasts a COMMIT message in response to a message $\langle \text{PREPARE}, v, s_i, m, UI_i\rangle$ only if three additional conditions are satisfied: (1) v is the current view number on s_j and the sender of the PREPARE message is the primary of v (only the primary can send PREPARE messages); (2) the request m contains a valid signature produced by the requesting client (to prevent a faulty primary from forging requests); and (3) s_j already accepted request m' 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). This last condition ensures that not only the requests are *executed* in the order defined by the counter of the primary, but also that they are *accepted* in that same order. Therefore, when a request is accepted it can be executed immediately (there is never the necessity of waiting for requests with lower numbers). The only exception is that if the server is faulty it can "order" the same request twice. So, when a server accepts a request, it first checks in V_{req} if the request was already executed and executes it only if not.

This message ordering mechanism imposes a FIFO order that is also enforced to other messages (in the view change operation) that also take a unique identifier UI: 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 $cv-1$. To enforce this property, each server keeps a vector V_{acc} with the highest counter value cv it received from each of the other servers in PREPARE, COMMIT, CHECKPOINT or VIEW-CHANGE messages. The FIFO order does not impose that the algorithm works in lockstep, i.e, the primary can send many PREPARE messages but all servers will accept the corresponding requests following the sequence order assigned by the primary.

FIFO order is needed to complement the use of unique identifiers in preventing duplicity. These identifiers alone do not prevent duplicity because a faulty server can send inconsistent messages to different servers, even if with different identifiers. By processing messages in the FIFO order indicated by the counters, all correct servers process the messages sent by the faulty server in the same order, or stop processing them if they do not receive one of them, thus preventing duplicity.

Servers: garbage collection and checkpoints. Messages sent by a server are kept in a message log in case they have to be resent. To discard messages from this log, MinBFT uses a garbage collection mechanism based on checkpoints, similar to PBFT's.

Servers generate checkpoints periodically when a request sequence number (the counter value in the UI produced by the primary) is divisible by the constant cp (checkpoint period). After the server s_j produces the checkpoint it multicasts $\langle \text{CHECKPOINT}, s_j, UI_{latest}, d, UI_j\rangle$

where UI_{latest} is the unique identifier of the latest executed request, d is the hash of the server's state and UI_j is obtained by calling `createUI` for the checkpoint message itself. A server considers that a checkpoint is *stable* when it receives $f+1$ CHECKPOINT messages signed by different replicas with the same UI_{latest} and d. We call this set of messages a *checkpoint certificate*, which proves that the server's state was correct until that request execution. Therefore, the replica can discard all entries in its log with sequence number less than the counter value of UI_{latest}.

The checkpoint is used to limit the number of messages in the log. We use two limiters: the low water mark (h) and the high water mark (H). The low water mark is the sequence number of the latest stable checkpoint. Replicas discard received messages with the counter value less than h. The high water mark is $H = h + L$ where L is the maximum size of the log. Replicas discard received messages with the counter value greater than H. This mechanism is for a single view. When there is a view change, a new checkpoint is generated and the log is cleaned.

Servers: view change operation. In normal case operation, the primary assigns sequence numbers to the requests it receives and multicasts these numbers to the backups using PREPARE messages. This algorithm strongly constrains what a faulty primary can do: it can not repeat or assign arbitrarily higher sequence numbers. However, a faulty primary can still prevent progress by not assigning sequence numbers to some requests, or even to any requests at all.

When the primary is faulty, a *view change* has to be executed and a new primary chosen. View changes are triggered by timeout. When a backup receives a request from a client, it starts a timer that expires after T_{exec}. When the request is accepted, the timer is stopped. If the timer expires, the backup suspects that the primary is faulty and starts a view change.

The view change operation is represented in Figure 1. When a timer in backup s_i times-out, s_i sends a message $\langle \text{REQ-VIEW-CHANGE}, s_i, v, v' \rangle$ to all servers, where v is the current view number and $v' = v + 1$ the new view number[3].

When a server s_i receives $f + 1$ REQ-VIEW-CHANGE messages, it moves to view $v+1$ and multicasts $\langle \text{VIEW-CHANGE}, s_i, v', C_{latest}, O, UI_i \rangle$, where C_{latest} is the latest stable checkpoint certificate and O is a set of all messages sent by the replica since the latest checkpoint was generated (PREPARE, COMMIT, VIEW-CHANGE and NEW-VIEW messages). At this point, that replica stops accepting messages for v.

The VIEW-CHANGE message takes a unique identifier UI_i obtained by calling `createUI`. The objective is to prevent faulty servers from sending different VIEW-CHANGE messages with different C_{latest} and O to different subsets of the servers, leading to different decisions on which was the last request of the previous view. Faulty servers still can do it, but they have to assign different UI identifiers to these different messages, which will be processed in order by the correct servers, so all will take the same decision on the last request of the previous view. Correct servers only consider $\langle \text{VIEW-CHANGE}, s_i, v', C_{latest}, O, UI_i \rangle$ messages that are consistent with the system state: (1) the checkpoint certificate C_{latest} contains at least $f+1$ valid UI identifiers; (2) the counter value (cv_i) in UI_i is $cv_i = cv+1$, 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_{latest} signed by the replica when it generated the checkpoint; and (3) there are no holes in the sequence number of messages in O.

When the new primary for view v' receives $f + 1$ VIEW-CHANGE messages from different servers, it stores them in a set V_{vc}, which is the *new-view certificate*. V_{vc} must contain all requests accepted since the previous checkpoint, and can also include requests that only were prepared. In order to define the initial state for v', the primary of this view uses the information in the C_{latest} and O fields in the VIEW-CHANGE messages to define S, which is the set of requests that were prepared/accepted since the checkpoint. To compute S, the primary starts by selecting the most recent (valid) checkpoint certificate received in VIEW-CHANGE messages. Next, it picks in VIEW-CHANGE messages the requests in O sets with UI counter values greater than the UI counter value in the checkpoint certificate.

After this computation, the primary multicasts a message $\langle \text{NEW-VIEW}, s_i, v', V_{vc}, S, UI_i \rangle$. When a replica receives a NEW-VIEW message it verifies if the *new-view certificate* is valid. All replicas also verify if S was computed properly doing the same computation as the primary. A replica begins the new view v' after all requests in S that have not been executed before are executed. If a replica detects that there is a hole between the sequence numbers of the latest request that it executed and of the first request in S, 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 (using the same protocol of PBFT).

In previous BFT algorithms, requests are assigned with sequential execution order numbers even when there are view changes. This is not the case in MinBFT as the sequence numbers are provided by a different tamperproof component (or USIG service) for each view. Therefore, when there is a view change the first sequence number for the new view has to be defined. This value is the counter value in the unique identifier UI_i in the NEW-VIEW message plus one. The next PREPARE message sent by the new primary must follow the UI_i in the NEW-VIEW message.

When a server sends a VIEW-CHANGE message, it starts a timer that expires after T_{vc} units of time. If the timer expires before the server receives a valid NEW-VIEW message, it starts another view change for view $v + 2$ [4]. If

3. It seems superfluous to send v and $v' = v + 1$ but in some cases the next view can be for instance $v' = v + 2$.

4. But the previous view is still v. Recall the previous footnote about REQ-VIEW-CHANGE messages.

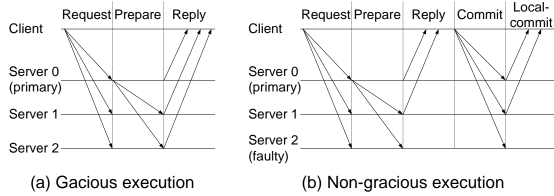

Figure 2. MinZyzzyva basic operation.

additional view changes are needed, the timer is multiplied by two each time, increasing exponentially until a new primary responds. The objective is to avoid timer expirations forever due to long communication delays.

The complete proof of correctness of the algorithm can be found in the supplemental material.

5 MINZYZZYVA

This section presents the second BFT algorithm of the paper, MinZyzzyva. This algorithm has characteristics similar to the previous one, but needs one communication step less in nice executions because it is speculative. MinZyzzyva is a modified version of Zyzzyva, the first speculative BFT algorithm [26].

The idea of *speculation* is that servers respond to clients' requests without first agreeing on the order in which the requests are executed. They optimistically adopt the order proposed by the primary server, execute the request, and respond immediately to the client. This execution is speculative because that may not be the real order in which the request should be executed. If some servers become inconsistent with the others, clients detect these inconsistencies and help (correct) servers converge on a single total order of requests, possibly having to rollback some of the executions. Clients only rely on responses that are consistent with this total order.

MinZyzzyva uses the USIG service to constrain the behavior of the primary, allowing a reduction of the number of replicas of Zyzzyva from $3f+1$ to $2f+1$, preserving the same safety and liveness properties.

Gracious execution. This is the optimistic mode of the algorithm. It works essentially as follows: (1) A client sends a request in a REQUEST message to the primary s_p. (2) The primary receives the request, calls `createUI` to assign it a unique identifier UI_p containing the sequence number (just like in MinBFT), and forwards the request and UI_p to other servers. (3) Servers receive the request, verify if UI_p is valid and if it comes in FIFO order, assign another unique identifier UI_s to the request, speculatively execute it, and send the response in a RESPONSE message to the client (with the two UI identifiers). (4) The client gathers the replies and only accepts messages with valid UI_p and UI_s. (5) If the client receives $2f+1$ matching responses, the request completes and the client delivers the response to the application.

Notice that $2f+1$ are all the servers. This is a requirement for MinZyzzyva to do gracious execution, just like it was for Zyzzyva. Clients and servers use request identifiers (seq) to ensure exactly-once semantics, just like in MinBFT (seq of the latest request executed of each client is stored in vector V_{req}). The client only accepts replies that satisfy the following conditions: (1) contain UI_p and UI_s that were generated for the client request; and (2) contain a UI_p that is valid and that is the same in all replies. Clients do not need to keep information about the servers' counter values.

A replica may only accept and speculatively execute requests following the primary sequence number order (FIFO order), but a faulty primary can introduce holes in the sequence number space. A replica detects a hole when it receives a request with the counter value cv in the primary's UI, where $cv > max_{cv}+1$ and max_{cv} is the counter value of the latest request received. In this situation, it sends to the primary a $\langle$FILL-HOLE$, s_i, v', max_{cv}+1, cv\rangle$ message and starts a timer. Upon receiving a FILL-HOLE message the primary sends all requests in the interval reported by the replica. The primary ignores FILL-HOLE messages of previous views. If the replica's timer expires without it having received a reply from the primary, it multicasts the FILL-HOLE message to the other replicas and also requests a view change by sending REQ-VIEW-CHANGE message (just like Zyzzyva).

Non-gracious execution. If the network is slow or one or more servers are faulty, the client may never receive matching responses from all $2f+1$ servers. When a client sends a request it sets a timer. If this timer expires and it has received between $f+1$ and $2f$ matching responses, then it sends a COMMIT message containing a *commit certificate* with these responses (with the UI_p and UI_s identifiers) to all servers. A commit certificate is thus composed by $f+1$ matching responses from $f+1$ different servers. These certificates can be (1) sent by a client in the non-gracious execution, (2) obtained when a view change occurs (below) or (3) obtained from a set of $f+1$ matching checkpoints (below).

When a correct server receives a valid commit certificate from a client, it acknowledges with a LOCAL-COMMIT message. Servers store in a vector V_{acc} the highest received counter value of the other servers (that come in the UI identifiers). With the UI_p and UI_s in the COMMIT message, the servers update their vector values[5]. The client resends the COMMIT message until it receives the corresponding LOCAL-COMMIT messages from $f+1$ servers. After that, the client considers the request completed and delivers the reply to the application. The system guarantees that even if there is a view change, all correct servers execute the request at this point.

If the client receives less than $f+1$ matching responses then it sets a second timer and resends the request to all servers. If a correct server receives a request that it has executed, it resends the cached response to the client. Otherwise, it sends the request to the primary and starts a timer. If the primary replies before the timeout, the server executes the request. If the timer expires before the primary sends a reply, the server initiates a view change.

5. Notice that the COMMIT messages do not contain the message history (as in Zyzzyva [26]), since the validity of UI_p requires that the operations were executed in order.

Using the USIG service, it is not possible to generate the same identifier for two different messages. A faulty primary can try to cause the re-execution of some requests by assigning it two different UI identifiers. However the servers detect this misbehavior using the clients' seq identifier in the request and do not do the second execution, just like in MinBFT[6].

Garbage collection and checkpoints. Like in Zyzzyva, the properties ensured by MinZyzzyva are defined in terms of histories. Each server in MinZyzzyva maintains an ordered *history* of the requests it has executed. Part of that history, up to some request, is said to be *committed*, while the rest is *speculative*. A prefix of the history is committed if the server has a commit certificate to prove that a certain request was executed with a certain sequence number.

Like in MinBFT, replicas generate checkpoints periodically, when the counter value in a UI generated by the primary is divisible by constant cp. After the replica s_j produces the checkpoint, it multicasts $\langle \text{CHECKPOINT}, s_j, UI_i, d, UI_j \rangle$ where UI_i is the unique identifier of the latest executed request, d is the digest of the current replica's state and UI_j is obtained by calling `createUI` for the checkpoint message itself. A replica considers that a checkpoint is stable when it receives $f+1$ CHECKPOINT messages with valid UI identifiers from different replicas with the same UI_i and d. Then all messages executed before UI_i are removed from the log.

View change. The view change operation works essentially as MinBFT's but MinZyzzyva weakens the condition under which a request appears in the new view message. When a server s_i suspects that the primary is faulty it sends a $\langle \text{REQ-VIEW-CHANGE}, s_i, v, v' \rangle$ message. When a server receives $f+1$ REQ-VIEW-CHANGE messages, it multicasts a $\langle \text{VIEW-CHANGE}, s_i, v', C_{latest}, O, UI_i \rangle$, where C_{latest} is the latest commit certificate collected by the replica and O is a set of ordered requests since C_{latest} that were executed by the replica. Each ordered request has UI_p signed by the primary and UI_i signed by the replica during the request execution. At this point, the replica stops accepting messages other than CHECKPOINT, VIEW-CHANGE and NEW-VIEW.

Correct servers evaluate VIEW-CHANGE messages and the new primary build NEW-VIEW messages exactly as in MinBFT. When a replica receives a NEW-VIEW message it verifies if the *new-view certificate* is valid. Replicas consider a valid NEW-VIEW message equivalent to a commit certificate.

This algorithm strongly constrains what a faulty primary can do since it can not repeat or assign arbitrarily high sequence numbers. However, due to the speculative nature of MinZyzzyva, in some cases servers may have to *rollback* some executions. This can happen after a view change when the new primary does not include in the NEW-VIEW message some operations that were executed by less than f servers. This can only happen for operations that do not have a commit certificate; therefore, the client also received neither $2f+1$ RESPONSE messages nor $f+1$ LOCAL-COMMIT messages, and thus the operations did not complete. Rollback is an internal server operation and does not involve the USIG service (there is no rollback of the counter). As mentioned before, the NEW-VIEW message is equivalent to a commit certificate, so operations that were rolled back by a replica will not appear in the next view changes or checkpoint messages.

The complete proof of correctness of the algorithm can be found in the supplemental material.

6 IMPLEMENTATION

We implemented the prototypes for both MinBFT and MinZyzzyva in Java. We chose Java for three reasons. First, we expect that avoiding bugs and vulnerabilities will be more important than performance in most BFT deployments, and Java offers features like sandboxing, type safety and memory safety that can make a BFT implementation more dependable. The second reason is to improve the system portability, making it easier to get deployed in different environments. Finally, we want to show that an optimized BFT Java prototype can have performance that is competitive with C implementations such as PBFT in terms of throughput.

The prototypes 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 (which can be seen as a simpler version of SEDA [47]) 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; messages received when there are no slots for running agreements are batched in the next agreement possible). Other common BFT optimizations [12] such as making agreements over the request hashes instead of the entire requests, and using authenticators were also employed in our prototypes. Additionally, we used recent Java features such as non-blocking I/O and the concurrent API (from packages java.nio and java.util.concurrent). Finally, we used TCP sockets.

Options for implementing the tamperproof component. Several options have been discussed in papers about the TTCB [16] and A2M [13]. The TTCB [16] can be seen as one of the earlier attempts at achieving isolation between general host services (including operating system) and a security kernel hosting trustworthy services. A2M [13] is another such example.

The main difficulty being the isolation of the service from the rest of the system, virtualization comes in as a very interesting solution, because a hypervisor provides isolation between a set of virtual machines with their own operating system. Examples include Xen [5] and other more security-related technologies like Terra [22], Nizza [43], and VM-FIT [38].

AMD's Secure Virtual Machine (SVM) architecture [1] and Intel's Trusted Execution Technology (TXT) [24] are

6. Therefore Zyzzyva's *proof of misbehavior* [26] is not needed in MinZyzzyva.

recent technologies that provide a hardware-based solution to launch software in a controlled way, allowing the enforcement of certain security properties. Flicker explores these technologies to provide a minimal trusted execution environment, i.e., to run a small software component in an isolated way [33]. Flicker and similar mechanisms can be used to implement the USIG service.

Implementing USIG-Sign with the TPM. As mentioned before, the simplicity of the USIG service allows it to be implemented with the Trusted Platform Module (TPM). More precisely, USIG-Sign can be implemented using TPMs compliant with the Trusted Computing Group (TCG) 1.2 specifications [36], [37].

TPMs have the ability to sign data using the private key of an attestation identity key pair (private AIK) that never exits the TPM, thus it can serve as the private key of the USIG service (the corresponding public keys have to be distributed by all servers). Furthermore, TPMs of version 1.2 provide a 32-bit *monotonic counter* on which only two commands can be executed: `TPM_ReadCounter` (returns the counter value), `TPM_IncrementCounter` (increments the counter and returns the new value).

The idea is to implement `createUI` in a way that the TPM produces a certificate containing not only a signature obtained using the message and the private key of this USIG (like before), but also a reading of the counter and a proof that is was incremented. This idea is implemented using the TPM's *transport command suite*. Each call to `createUI` involves a communication session between the USIG software library and the local TPM. During this session a sequence of TPM commands are issued, a log of the commands is kept, a hash of this log is obtained, and a digital signature of this hash (with the private AIK) is obtained. The certificate is composed by the signature plus the log, which must contain a call to `TPM_IncrementCounter` to be considered valid.

If two applications need to use BFT replication in the same set of servers and they can not share the same replication service, two USIG service instances are needed per machine. This is not a limitation because with a software-implemented USIG it is simple to implement several USIG services. For a hardware-implemented USIG, it is possible to use a virtualized TPM [7], so our service can also be used by several replicated services with a single instance of the hardware.

Implementations of the USIG service Our algorithm implementations access the USIG service through a small abstract Java class that was extended to implement the several versions of USIG. In all these versions the fundamental idea was to isolate the service from the rest of the system but the levels of isolation obtained are different (see Figure 3):

- *Non-secure USIG:* The non-secure version of USIG (NS) is a simple class that provides methods to increment a counter and return its value together with a signature. This version of the USIG service is not isolated and thus can be tampered by a malicious adversary that controls the machine. The practical interest of this version is to allow us to understand what would be the performance of our algorithms if the time of accessing the USIG service was 0.
- *VM-based USIG:* This version runs the USIG service as a process in a virtual machine (VM) different from the one in which the normal system (operating system, algorithm code) runs. In each system replica we use the Xen hypervisor [5] to isolate the replica process and the USIG service. The replica with the algorithm code runs on *domain1*, which is connected to the network and contains all untrusted software. The USIG service (a hundred lines of Java code plus the crypto lib) runs on *domain0*, which is not connected to the network and contains as little services as possible. The communication between the replica process and the USIG service is done using sockets. To ensure that the counter value is kept when a replica reboots, its value should be stored in a flash memory or other high speed secondary storage (but this feature was not implemented in our prototype).
- *TPM USIG:* The TPM-based version of USIG is the most secure version of the service we have deployed so far since the service is implemented by trusted hardware, providing stronger isolation. In this version the USIG service is implemented by a thin layer of software (a function in a library) and by the TPM itself, as discussed above. The identifier generated by the service is signed using the TPM's private AIK, a RSA key with 2048-bits. We used TPM/J, an object-oriented API written in Java, to access the TPM [41].

To explore the costs associated with the authentication operations, versions NS and VM of the USIG service were implemented using several methods of authenticating an *UI*: NTT ESIGN with 2048-bit keys (Crypto++ lib accessed through the Java Native Interface), RSA with 1024-bit keys and SHA1 to produce HMACs (both from the Java 6 JCA default provider). Using HMAC, the servers have a shared key therefore the *UI* verification has to be carried out inside of the trusted service. For this reason, only MinBFT can use the USIG service implemented with HMAC (USIG-Hmac). In MinZyzzyva the client verifies if the *UI* is the same in all server replies, which turns impossible the use of HMACs in this algorithm in our system model (only the servers have a trusted module).

Considering the two possible implementations of the USIG service (Section 2) and the kinds of isolation that can be seen in Figure 3, we have implemented seven versions of the USIG service: NS-Hmac, NS-Sign(ESIGN), NS-Sign(RSA), VM-Hmac, VM-Sign(ESIGN), VM-Sign(RSA)

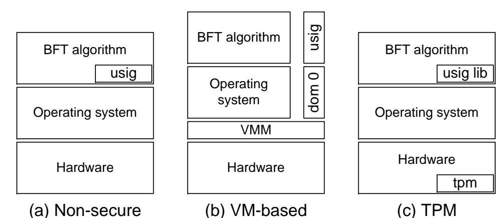

Figure 3. USIG versions: in process (Non-Secure, NS); in a virtual machine (VM); using trusted hardware (TPM).

and TPM.

The counter used in the NS and VM versions of the USIG has 64 bits (a Java long variable), which is enough to prevent it from burning out in less than 2^{33} years if it is incremented twice per millisecond.

We assume that it is not possible to tamper with the service, e.g., decrementing the counter, but privileged software like the operating system might call the function `createUI`. This is a case of faulty replica as the replica deviates from the expected behavior (does not use sequential values for UI) but the service remains correct. A simple authentication mechanism is used to prevent processes other than the replica processes from accessing the service. Both, the TPM and VM-based version are able to continue to work correctly even under attacks coming from the network against the server software. However, only the version with the TPM is tolerant to a malicious administrator that manipulates the services hosted by f servers, and even this version is not tolerant to physical attacks.

7 Performance Evaluation

This section presents performance results of our algorithms using micro-benchmarks. We measured the latency and throughput of the MinBFT and MinZyzzyva implementations using null operations.

PBFT [12] is often considered to be the baseline for BFT algorithms, so we were interested in comparing our algorithms with the implementation available on the web[7]. To compare this implementation with our MinBFT and MinZyzzyva algorithms, we made our own implementation of PBFT's normal case operation in Java (JPBFT). We did not compare with the TTCB-based algorithm and A2M-PBFT-EA because their code was not available.

Unless where noted, we considered a setup that can tolerate one faulty server ($f = 1$), requiring $n = 4$ servers for PBFT and JPBFT and $n = 3$ servers for MinBFT and MinZyzzyva. We did not explore higher values of f because increasing the number of replicas is costly (e.g., to enforce diversity), so we believe in practice $f = 1$ will be used. We executed from 1 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 through a Dell gigabit switch. The PCs had a Atmel TPM 1.2 chip. 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. All experiments run only in normal case operation, without faults and timeout expirations, which is usually considered to be the normal case. The above-mentioned batch mechanism was used.

7.1 Micro-Benchmarks

For the first part of the performance evaluation we chose the versions of MinBFT and MinZyzzyva that presented best performance and we compared them with PBFT. We evaluated the performance of four algorithms, PBFT, JPBFT, MinBFT-Hmac and MinZyzzyva-Sign(ESIGN), on a LAN. The two last algorithms used the VM-based USIG service. We measured the latency of the algorithms using a simple service with no state that executes null operations, with arguments and results varying between 0 and 4K bytes. The latency was measured at the client by reading the local clock immediately before the request was sent, then immediately after a response was consolidated (i.e., the same response was received by a quorum of servers), and subtracting the former from the latter. Each request was executed synchronously, i.e., it waited for a reply before invoking a new operation. The results were obtained by timing 100,000 requests in two executions. The obtained latencies are averages of these two executions. The results are shown in Table 2.

Req/Res	PBFT	JPBFT	MinBFT-Hmac	MinZyzzyva-Sign
0/0	0.4 ms	1.8 ms	2.3 ms	2.9 ms
4K/0	0.6 ms	2.2 ms	2.9 ms	3.1 ms
0/4K	0.8 ms	2.5 ms	3.0 ms	3.2 ms

Table 2
Latency varying request and response size for the best versions of MinBFT and MinZyzzyva, plus PBFT and JPBFT.

In this experiment, PBFT has shown the best performance of all algorithms/implementations, followed by JPBFT, MinBFT-Hmac and MinZyzzyva-Sign, which was the worse. This experiment shows clearly that our Java implementation runs an agreement much slower than the original PBFT prototype. 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. When compared with JPBFT, MinBFT-Hmac has a small extra cost because of the overhead to access the USIG service to create and to verify the UI. Zyzzyva is known to be faster than PBFT in most cases [26], [44], but Zyzzyva (like PBFT) uses only HMACs, while MinZyzzyva uses signatures, so MinZyzzyva ends up being slower than PBFT, JPBFT and MinBFT-Hmac.

The second part of the micro-benchmark had the objective of measuring the peak throughput of the algorithms with different loads. We ran experiments using requests and responses with 0 bytes. We varied the number of logical clients between 1 and 120 in each experiment, where 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.

Figure 4 shows that the fewer communication steps and number of replicas in MinZyzzyva is reflected in higher throughput by achieving around 30,000 operations per second. For the same reason, the MinBFT-Hmac throughput is 10% higher than the one observed for PBFT. It is interesting to notice that the reduced number of communication steps and replicas (which reduces the quorum sizes used by the algorithms) makes the replicas process less

7. http://www.pmg.lcs.mit.edu/bft/ – the updated version that compiles with gcc4.

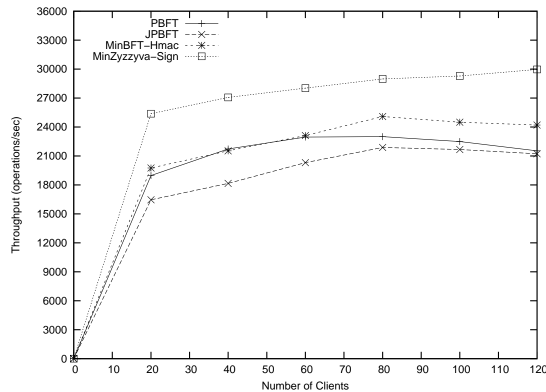

Figure 4. Peak throughput for 0/0 operations for the best versions of MinBFT and MinZyzzyva, together with PBFT and a similar Java implementation.

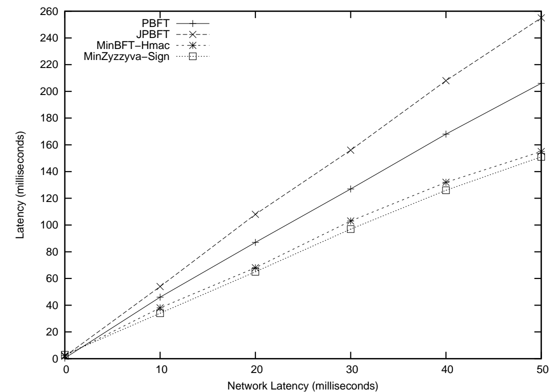

Figure 5. Latency for 0/0 operations with several link latency values.

messages (less I/O, less HMAC verification, etc.), which increases the throughput. Due to the optimizations for scalability discussed in Section 6, JPBFT presented only 5% lower throughput when compared with the original PBFT implementation.

The throughput values in the figure together with the latency values of Table 2 show the effect of adaptive batching. The similarity on the peak throughput values is explained by the fact that, in our experiments under heavy load (e.g., 120 clients accessing the system), PBFT runs more agreements with batches of up to 70 messages while our algorithms use batches of up to 200 messages.

7.2 Effects of Communication Latency

In the third experiment we emulated WAN network delays on all links and run latency experiments to better understand how these algorithms would behave if the replicas and clients were deployed on different sites. Despite the fact that this scenario does not correspond to what is most common today (all replicas inside a data center), it makes sense if one considers the deployment of a fault independent replicated system: it can tolerate malicious attacks (such as DDoS), link failures, site misconfigurations, natural disasters and many other problems that can affect whole sites. We used *netem*[8] to inject delays in each machine by varying the delays between 1 and 50 ms and use a standard deviation of 10% of the injected delay. The latency was measured in the same way as in previous section. Figure 5 presents the results.

As expected, the measurements show that the latency becomes higher with larger delays. Due to the *tentative execution* optimization (described in Section 6.1 of [12]), PBFT reduces the number of communication steps from 5 to 4 communication steps, and it is reflected in the results obtained. We did not implement this optimization in JPBFT, therefore it presents the worse latency in our experiments. MinBFT and MinZyzzyva presented the best latency results when the latency is greater than 2 ms, due to their smaller number of communication steps. Surprisingly, MinBFT executes requests with almost the same latency as MinZyzzyva, which contradicts the theoretical number of communication steps of these algorithms: 4 and 3 respectively. The explanation for this fact highlights one interesting advantage of MinBFT when tolerating a single fault. 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 MinBFT algorithm needs only $f + 1$ COMMIT messages to accept a request, with $f = 1$ only two COMMITs are required. These two COMMITs would be received just after the PREPARE: one from the leader and another from the server itself. Therefore, the client request is executed soon after the PREPARE message from the primary server arrives, making MinBFT reach the latency of MinZyzzyva. In setups with $f > 1$ this nice feature will not appear since the quorum for COMMIT acceptance should contains at least 3 replicas. Therefore, the use of small quorums can make our algorithms particularly efficient in real networks due to their large variance in link latency [25].

7.3 Comparing Different USIG Versions

To explore the different implementations of the USIG service and the computational overhead added by different cryptographic mechanisms, we measured the latency and throughput of MinBFT and MinZyzzyva with all USIG implementations in a LAN, except the TPM-based that is evaluated in the next section.

Figure 6 shows the results for our latency experiments. The signature-based versions of the algorithms add significant computational overheads when compared with HMAC-based authenticators.

Algorithm	`createUI`	`verifyUI`
Hmac (SHA1)	0.008 ms	0.007 ms
ESIGN (2048 bits)	1.035 ms	0.548 ms
RSA (1024 bits)	10.683 ms	0.580 ms

Table 3
Overhead of UI creation and verification for messages with 20 bytes (the size of request hash).

Since the algorithms require two `createUI` calls and one (in MinZyzzyva) or $f+1$ (in MinBFT) `verifyUI` call in their

8. http://www.linuxfoundation.org/en/Net:Netem.

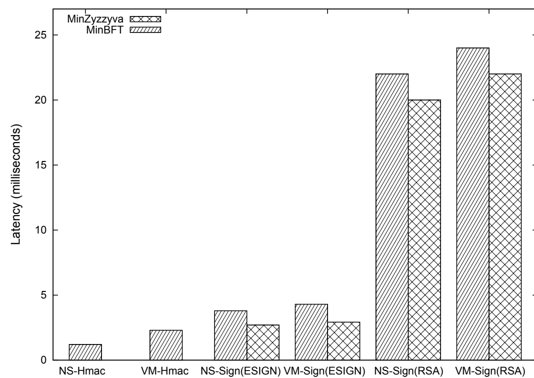

Figure 6. Latency of 0/0 operations for MinBFT and MinZyzzyva using several USIG implementations.

critical path, the algorithms latencies are very dependent of the USIG implementation. To better understand the nature of that relation, it is worth understanding the costs of the cryptography employed in these versions. Table 3 presents the latency of `createUI` and `verifyUI` on several implementations of the USIG (with has no access cost, i.e., NS versions). The data in this table explains the results observed on Figure 6: the use of ESIGN adds roughly 2.5 ms to the latency of MinBFT when compared with Hmac, while RSA adds 17.5 ms when compared with ESIGN.

Figure 7 shows the throughput of the algorithms with the different USIG implementations. The VM-based versions have throughput lower than the non-secure versions due to the overhead to access the trusted service. This difference is especially relevant when comparing the values for the VM-based MinBFT using Hmac and its corresponding Non-Secure version because the *UI* verification is executed inside of the trusted component.

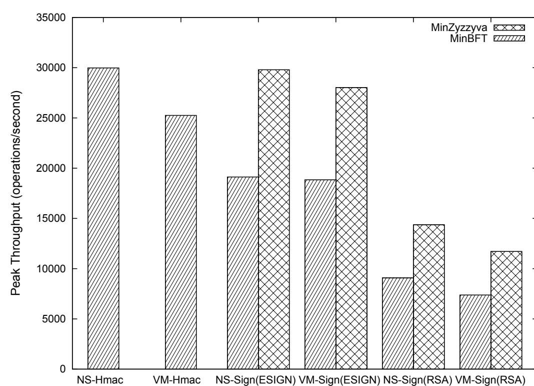

Figure 7. Peak throughput for 0/0 operations in for MinBFT and MinZyzzyva using several USIG implementations.

This graph shows that the costs of accessing the VM-based USIG lowers the peak throughput by a factor from 6% (MinZyzzyva-Sign(ESIGN)) to 16% (MinBFT-Hmac). It shows that the VM-based isolation is a cost-effective solution in the sense that a moderate level of isolation can be obtained without losing too much performance.

7.4 Hardware-based USIG Performance

Table 4 presents the latency and peak throughput of the USIG service implemented with an Atmel TPM 1.2 chip in the computer mainboard.

Algorithm	Latency	Peak Throughput
MinBFT	1617 ms	23404 ops/s
MinZyzzyva	1552 ms	24179 ops/s

Table 4
Latency and peak throughput of MinBFT and MinZyzzyva using the TPM USIG.

The time taken by the TPM-based USIG service to run `createUI` is 797 ms, almost all of which is taken by the TPM to increment the counter and produce an RSA signature. In this sense, the latency values can be explained by the execution of two `createUI` executed in the critical path of the algorithm. The verification of a *UI* takes approximately 0.07 ms, since it is executed outside of the TPM, so, its effect in the latency is minimal.

The peak throughput shows that the values are not so bad if compared with the values presented in Figure 7. However, to obtain these values with TPM USIG we needed to batch a large number of requests in the PREPARE messages because the restriction of one increment by 3.5 seconds. The throughput is strictly dependent of the number of messages batched during this time, in our experiments we found that the peak throughput was achieved with batches with more than 20000 messages. So, the behavior of the execution of our system is: 3.5 seconds without accepting any message followed by one second accepting with more than 20000 messages, which may be unacceptable in many practical services.

There are at least two important reasons for the poor performance of the TPM USIG. The first is the maximum increment rate of the TPM monotonic counter, which makes the system able to execute one agreement (to order a batch of messages) every 3.5 seconds. The TPM specification version 1.2 defines that the monotonic counter "must allow for 7 years of increments every 5 seconds" and "must support an increment rate of once every 5 seconds" [36]. The text is not particularly clear so the implementers of the TPM seem to have concluded that the counter must not be implemented faster than once every 5 seconds approximately, while the objective was to prevent the counter from burning out in less than 7 years. The counter value has 32 bits, so it might be incremented once every 52 ms still attaining this 7-year target. Furthermore, if in a future TPM version the counter size is increased to 64 bits (as it is in our VM-based USIG), it becomes possible to increment a counter every 12 picoseconds, which will make this limitation disappear. The second reason for the poor performance we observed is the time the TPM takes to do a signature (approximately 700 ms). A first comment is that normally cryptographic algorithms are implemented in hardware to be faster, not slower, but our experiments have shown that with the TPM the opposite is true. This suggests that the performance of the TPM signatures might be much improved. We believe that it will be indeed improved with the development of new applications for the TPM. Moreover, at least Intel is much interested in developing the TPM hardware. For instance, it recently announced that it will integrate the

TPM directly into its next generation chipset [10]. Others have also been pushing for faster TPM cryptography [32].

8 Related Work

The idea of tolerating intrusions (or arbitrary/Byzantine faults) in a subset of servers appeared in seminal works by Pease et al. [35] and Fraga and Powell [19]. However, the concept started raising more interest much later with works such as Rampart [39] and PBFT [12].

The idea of using a hybrid fault model in the context of intrusion tolerance or Byzantine fault tolerance, was first explored in the MAFTIA project with the TTCB work [16]. The idea was to extend the replicas with a tamperproof subsystem. It was in this context that the first $2f+1$ state machine replication solution appeared [15]. However, it was based on a distributed trusted component, with harder to enforce tamperproofness.

More recently Chun et al. presented another $2f+1$ BFT algorithm based on similar ideas, A2M-PBFT-EA [13]. This algorithm requires only local tamperproof components, dubbed Attested Append-Only Memory (A2M). The A2M is an abstraction of a trusted log. A2M offers methods to append values and to lookup values within the log. It also provides a method to obtain the end of the log and to advance the suffix stored in memory (used to skip ahead by multiple sequence numbers). There are no methods to replace values that have already been assigned. The main goal of this trusted component is to provide a mechanism for algorithms to become immune to duplicity, similarly to the USIG used in our algorithms. Replicas using the A2M are forced to commit to a single, monotonically increasing sequence of operations. Since the sequence is externally verifiable, faulty replicas can not present different sequences to different replicas. MinBFT and MinZyzzyva are BFT algorithms that also require only $2f+1$ servers, but use a simpler trusted service and a lower number of communication steps.

The quest for reducing the number of replicas of BFT algorithms had other interesting developments. Yin et al. presented a BFT algorithm for an architecture that separates agreement (made by $3f+1$ servers) from service execution (made by $2f+1$ servers) [49]. This was an important contribution to the area because service execution is expected to require much more computational resources than agreement. However agreement still needs $3f+1$ machines, while in the present work we need only $2f+1$ replicas also for agreement. Li and Mazieres proposed an algorithm, BFT2F, that needs $3f+1$ replicas but if more than f but at most $2f$ replicas are faulty, the system still behaves correctly, albeit sacrificing either liveness or providing only weaker consistency guarantees [29].

Monotonic counters are a service of the TPM that appeared only in version 1.2 [36], [37]. Two papers have shown the use of these counters in very different ways than we way we use them. van Dijk et al. addressed the problem of using an untrusted server with a TPM to provide trusted storage to a large number of clients [45]. Each client may own and use several different devices that may be offline at different times and may not be able to communicate with each other except through the untrusted server. The challenge of this work is not to guarantee the privacy or integrity of the clients' data, but in guaranteeing the data freshness. It introduces freshness schemes based on a monotonic counter, and shows that they can be used to implement tamper-evident trusted storage for a large number of users.

The TCG specifications mandate the implementation of four monotonic counters in the TPM, but also that only one of them can be used between reboots [36]. Sarmenta et al. override this limitation by implementing virtual monotonic counters on an untrusted machine with a TPM [41]. These counters are based on a hash-tree-based scheme and the single usable TPM monotonic counter. These virtual counters are shown to allow the implementation of count-limited objects, e.g., encrypted keys, arbitrary data, and other objects that can only be used when the associated counter is within a certain range.

9 Conclusion

BFT algorithms typically require $3f+1$ servers to tolerate f Byzantine servers, which involves considerable costs in hardware, software and administration. Therefore reducing the number of replicas has a very important impact on the cost of the system. We show that using a simple trusted service (only a counter plus a signing function) it is possible to reduce the number of replicas to $2f+1$ preserving the same properties of safety and liveness of traditional BFT algorithms. Furthermore, we present two BFT algorithms that are better than others in the literature, not only in terms of number of replicas and trusted service used, but also of communication steps in nice executions: 4 and 3 steps, respectively without and with speculation. This is an important aspect in terms of latency, especially in networks with non-negligible communication delays. In contrast with the two previous $2f+1$ BFT algorithms, we were able to use the TPM as the trusted component due to the simplicity of our USIG service.

Acknowledgments

We warmly thank Klaus Kursawe, Paulo Sousa, Luis Sarmenta, Jean-Philippe Martin, Hans P. Reiser, Eduardo Alchieri, Wagner Dantas and the anonymous reviewers for discussions about the algorithms, help with several implementation aspects and feedback on the paper. This work was partially supported by the EC through project TCLOUDS (FP7/2007-2013, ICT-257243), and by the FCT through project REGENESYS (PTDC/EIAEIA/100581/2008), the Multiannual Program, and the INESC-ID multiannual PIDDAC Program funds.

References

[1] Advanced Micro Devices. Amd64 virtualization: Secure virtual machine architecture reference manual. Technical report, May 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, 2002.

[3] 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.

[4] 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 *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 105–114, June 2006.

[5] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Oct. 2003.

[6] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 281–289, 2003.

[7] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320, Aug. 2006.

[8] 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.

[9] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, Aug. 1984.

[10] M. Branscombe. How hardware-based security protects PCs. Tom's Hardware, http://www.tomshardware.com/reviews/ hardware-based-security-protects-pcs,1771.html, Feb. 2008.

[11] C. Cachin and A. Samar. Secure distributed DNS. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 423–432, 2004.

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

[13] 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.

[14] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290, 2009.

[15] 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.

[16] M. Correia, P. Verissimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the 4th European Dependable Computing Conference*, pages 234–252, Oct. 2002.

[17] 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 Symposium on Operating Systems Design and Implementations*, pages 177–190, Nov. 2006.

[18] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

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

[20] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2011.

[21] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for Byzantine fault-tolerant database replication. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems*, page 107, Apr. 2011.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, 2003.

[23] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988.

[24] Intel Corporation. LaGrande technology preliminary architecture specification. Intel Publication D52212, May 2006.

[25] F. Junqueira, Y. Mao, and K. Marzullo. Classic paxos vs. fast paxos: Caveat emptor. In *Proceedings of 3rd Workshop on Hot Topics on System Dependability - HotDep'07*, 2007.

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

[27] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. IETF Request for Comments: RFC 2104, Feb. 1997.

[28] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[29] J. Li and D. Mazieres. Beyond one-third faulty replicas in Byzantine faul tolerant systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, pages 131–144, Apr. 2007.

[30] J. P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[31] J. P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, volume 2508 of *LNCS*, pages 311–325. Springer-Verlag, Oct. 2002.

[32] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? recommendations for hardware-supported minimal TCB code execution. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–25, Mar. 2008.

[33] J. M. McCune, B. J. Parno, A. P., M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference*, Apr. 2008.

[34] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, Univ. of Lisbon, Sept. 2006.

[35] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[36] Trusted Computing Group. TPM Main, Part 1 Design Principles. Specification Version 1.2, Revision 103. July 2007.

[37] Trusted Computing Group. TPM Main, Part 3 Commands. Specification Version 1.2, Revision 103. July 2007.

[38] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, pages 83–92, Oct. 2007.

[39] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 1995.

[40] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright. The Ω key management service. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 38–47, 1996.

[41] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing*, pages 27–42, Nov. 2006.

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

[43] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Operating Systems Review*, 40(4):161–174, 2006.

[44] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, Apr. 2008.

[45] M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing*, pages 41–48, Nov. 2007.

[46] P. Verissimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[47] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.

[48] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems*, pages 260–269, Oct. 2003.

[49] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, Oct. 2003.

[50] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.