Minimal Byzantine Fault Tolerance: Algorithms and Evaluation

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

DI–FCUL TR–2009–15

June 2009

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

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

Minimal Byzantine Fault Tolerance: Algorithms and Evaluation

Giuliana Santos Veronese[1], Miguel Correia[1], Alysson Neves Bessani[1],
Lau Cheuk Lung[2] , Paulo Verissimo[1]

[1]Universidade de Lisboa, Faculdade de Ciências, LASIGE, Portugal

[2]Universidade Federal de Santa Catarina, Departamento de Informática e Estatística, Brazil

June 2009

Abstract

This paper presents two asynchronous Byzantine fault-tolerant state machine replication (BFT) algorithms that are minimal in several senses. 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 arguably minimal, so it is simple to verify and implement (which is possible even 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 of having less 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.

Comparing with the previous paper DI-TR-08-29 [49], this version presents a slight modifications of the algorithms, the full proof of their correctness and a new performance evaluation.

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 making computer security as a whole an important research challenge. To meet this challenge several *asynchronous Byzantine fault-tolerant algorithms* have been proposed. The main idea of these algorithms is to allow a system to continue to operate correctly even if some of its components exhibit arbitrary, possibly malicious, behavior [5, 12, 13, 14, 16, 18, 24, 26, 29, 39, 51]. These algorithms have already been used to design *intrusion-tolerant* services such as network file systems [12, 51], cooperative backup [4], large scale storage [3], secure DNS [10], coordination services [7], certification authorities [52] and key management systems [40].

Byzantine fault-tolerant systems are usually built using replication techniques. The *state machine approach* is a generic replication technique to implement deterministic fault-tolerant services. It was first defined as a means to tolerate crash faults [42] and later extended for Byzantine/arbitrary faults [39, 12]. The algorithms of the latter category are usually called simply BFT. There are, however, other algorithms in the literature that are Byzantine fault-tolerant but that provide weaker semantics, e.g., registers implemented with quorum systems [30]. When we speak about BFT in the paper, we do not include these.

Minimal number of replicas. BFT algorithms typically require $3f+1$ *servers* (or *replicas*[1]) to tolerate f Byzantine (or *faulty*) servers [12, 16, 24, 39]. Clearly a majority of the servers must be non-faulty because the idea is to do voting on the output of the servers and faulty servers can not be reliably identified, but these algorithms require f additional servers.

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 be 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. [27, 33] This involves additional considerable costs per-replica, in terms not only of hardware but especially of software development, acquisition and management.

The paper presents two novel BFT algorithms that are *minimal* in several senses. The first, 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. It was previously shown that a Byzantine fault-tolerant service can be executed in only $2f+1$ replicas,

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

while the agreement part of BFT algorithms requires $3f+1$ replicas [51]. However, the algorithms presented in this paper require only $2f+1$ replicas both for execution of the service and agreement.

Minimal trusted service. A few years ago, an algorithm that needs only $2f+1$ replicas was published [14]. 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 with only $2f+1$ replicas, which 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, which like the TTCB has to be tamperproof, but that is local to the computers, not distributed. Replicas utilizing 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.

These two works have shown that in order to reduce the number of replicas from $3f+1$ to $2f+1$ the replicas have to be extended with tamperproof components. While $3f+1$ BFT algorithms tolerated any failure in up to f replicas, $2f+1$ BFT algorithms also tolerate up to f faulty replicas, but these special components can not be compromised. 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 [20]). However, 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 minimal is that the trusted/tamperproof service in which they are based is simpler that the two previous in the literature and arguably minimal. Looking to the proof of why it is not possible to design a BFT agreement algorithm with less than $3f+1$ servers [25, 8], it can be seen that the problem is that the 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, the minimal functionality that a trusted service must provide is 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. On the other hand, the values generated by this counter should be unforgeable, so some kind of authentication must be employed. The trusted service presented in this paper (USIG) provides an interface with operations to increment a counter and to verify if other counter values (incremented by other replicas) are correctly authenticated.

A 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) [35]. This secure co-processor is currently available in the mainboard of many commodity PCs. The TPM provides services like secure random number generation, secure storage and digital signatures [36]. Using COTS trusted hardware is an obvious benefit in relation to previous algorithms, but also a challenge: we can not define the tamperproof abstraction that better suites our needs, but are restricted to those provided by the TPM. Our performance evaluation shows that the versions of our algorithms that use the TPM have very poor performance (unlike versions that use other implementations of the service). We discuss how the TPM design and implementations can evolve to become usable for practical BFT algorithms.

Minimal number of steps. The number of communication steps is an important metric for distributed algorithms, for the delay of the communication tends to have a major impact in the latency of the algorithm. This is specially important in WANs, where the communication delay can be as much as a thousand times higher than in LANs and, in fact 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. 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 other) and that the counter was incremented (so the same number can not be used twice). This is used to guarantee that all nonfaulty replicas take into account the same messages 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 [24].

		PBFT [11, 12] (+[51])	Zyzzyva [24]	A2M-PBFT-EA [13]	MinBFT this paper	MinZyzzyva this paper
Model	Tamperproof component	no	no	A2M	USIG	USIG
Speculative		no	yes	no	no	yes
Cost	Total replicas	$3f+1$	$3f+1$	$2f+1$	$2f+1$	$2f+1$
	Replicas with application state	$2f+1$ [51]	$2f+1$	$2f+1$	$2f+1$	$2f+1$
Throughput	MAC ops at bottleneck server	$2+(8f+1)/b$	$2+3f/b$	$2+(2f+4)/b$	$2+(f+3)/b$	$2+Sign/b$ (*)
Latency	Communication steps	5 / 4	3	5	4	3

Table 1: Comparison of BFT algorithms, expanding Table 1 in [24]. 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. The line "cost" considers the result by Yin et al. that the execution of the application can be done in only $2f+1$ [51]. In the algorithms that use a tamperproof component, some MACs are done inside this component. (*) MinZyzzyva does 2 batches and one signature.

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 minimal in terms of this metric for in nice executions the two algorithms run in the minimum known number of communication steps of non-speculative and speculative algorithms, respectively 4 and 3 steps [29, 24]. Notice that the gain of one step in speculative algorithms comes at a price: in certain situations Zyzzyva and MinZyzzyva may have to rollback some executions, which makes the programming model more complicated.

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.

At this stage it is important to comment that there are no free lunches and that these improvements have their drawbacks also. In relation to BFT algorithms that do not use a trusted component, our algorithms (and the other two in the literature [13, 14]) have the disadvantage of having one additional point of failure: the tamperproofness of the component. In practice, this prevents these algorithms from being used in settings in which the potential attacker has physical access to a replica, as protecting even hardware components from physical attacks is at best complicated.

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

- is presents the USIG service and shows that it allows the implementation of BFT algorithms with only $2f+1$ replicas, being arguably the simplest service that allows the implementation of BFT replication;
- it presents two BFT algorithms that are minimal 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 minimality of the presented BFT algorithms in terms of number of replicas, trusted service 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)* assigns to messages (i.e., arrays of bytes) identifiers that are unique, increasing and sequential. 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 (increasing), and (3) will never assign an identifier that is not the successor of the previous one (sequentiality). The main components of the service are a *counter* and cryptographic mechanisms (details below). The interface of the service has two functions:

`createUI`(m) – returns a *USIG certificate* that certifies that the *unique identifier UI* contained in the certificate was created by this tamperproof component for message m. The unique identifier includes 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, in terms of the certificate scheme used:

USIG-Hmac: a certificate contains a HMAC obtained using the message and a secret key owned by this USIG but known by all the others, for them to be able to verify the certificates generated.

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

Notice that 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 component.

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 a HMAC primitive (for USIG-Hmac) or a digital signature primitive (for USIG-Sign).

Implementing the tamperproof component. In this section we briefly survey a set of solutions that can be used to make the USIG service tamperproof. Several options have been discussed in papers about the TTCB [15] and A2M [13].

The main difficulty is to isolate the service from the rest of the system. Therefore a solution is to use virtualization, i.e., a hypervisor that provides isolation between a set of virtual machines with their own operating system. Examples include Xen [6] and other more security-related technologies like Terra [19], Nizza [43] and Proxos [45].

AMD's Secure Virtual Machine (SVM) architecture [2] and Intel's Trusted Execution Technology (TXT) [21] are 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 [32]. 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 permits that it is implemented with the TPM. The service requires 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 the attestation identity key pair (private AIK for simplicity) that never exits the TPM. The corresponding public keys have to be distributed by all servers.

The TPM (1.2) provides a 32-bit *monotonic counter* in which only two commands can be executed: `TPM_ReadCounter` (returns counter value), `TPM_IncrementCounter` (increments the counter and returns the new value). The TCG imposes that the counters are not increasable arbitrarily often to prevent that they burn out in less than 7 years. In the TPMs we used in the experiments, counters could not be increased more than once every 3.5 seconds approximately (and the same is verified in other TPMs [41]). This feature seriously constrains the performance of our algorithms, so later we discuss how this might be improved.

The implementation of the USIG service is based on the *transport command suite*. The idea is to create a session that is used to do a sequence of TPM commands, to log the executed commands, and to obtain a hash of this log along with a digital signature of this hash (with the private AIK). The main idea is that in this version of the USIG, the USIG certificate not only contains a signature but also the hash of the log, and the log must contains a call to `TPM_IncrementCounter` to be considered valid.

More details can be found in the Appendix C.

3 System Model

The system is composed by a set of n servers $P = \{s_0, ..., s_{n-1}\}$ that provide 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 for being retransmitted. An attacker may have access to the network and be able to modify messages, so messages contain digital signatures or message authentication codes (MACs). Servers and clients know the keys they need to check these signatures/MACs. 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 this requires that the servers are diverse [27, 33]. 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 well-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 (see next section). Therefore, the fault model we consider is *hybrid* [48]. The Byzantine model states that any number of clients and any f servers can be faulty. However, the USIG service is tamperproof, i.e., always satisfies its specification, even if it is 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 PBFT [12]). This rather weak assumption has to be satisfied only to ensure the liveness of the system, not its safety.

4 MinBFT

This section presents MinBFT, the non-speculative $2f+1$ BFT algorithm. 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*. 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.

MinBFT follows a message exchange pattern similar to PBFT (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 $s_p \triangleq v \bmod n$, where v is the current view number. Clients issue *requests* with operations.

In *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 valid 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 $s'_p \triangleq v' \bmod n$ 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.

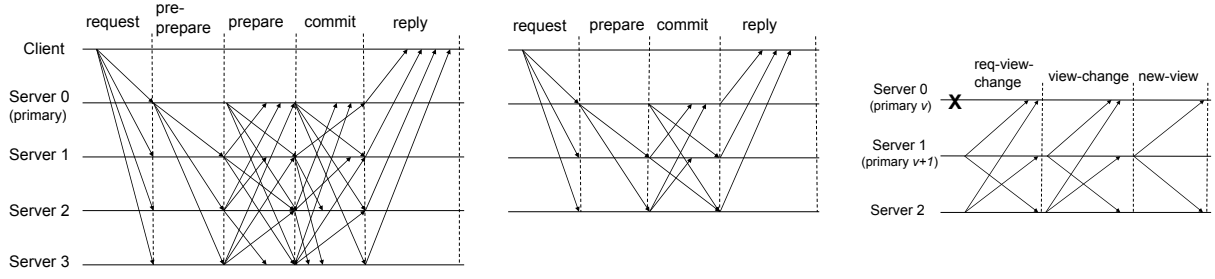
Clients. A client c requests the execution of an operation *op* by sending a message $\langle \text{REQUEST}, c, seq, op \rangle_{\sigma_c}$ to all servers. *seq* is the request identifier that is used to ensure exactly-once semantics: (1) the servers store in a vector V_{req} the *seq* of the last request they executed for each client; (2) the servers discard requests with *seq* lower than the last 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 steps, not three like PBFT [12] or A2M-PBFT-EA [13]. 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 a unique identifier *UI* obtained by calling the `createUI` function, so no two messages can have the same identifier. Servers check if the identifier of the messages they receive are valid for these messages using the `verifyUI` function.

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 that client (to prevent a faulty primary from forging requests); and (3) s_j already received a message 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.



(a) PBFT (b) MinBFT (c) MinBFT view change

Figure 1: Normal case operation of PBFT (a) and MinBFT (b). View change operation of MinBFT (c)

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, NEW-VIEW 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.

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, very similar to PBFT's.

Replicas generate checkpoints periodically when a request sequence number (the counter value of the *UI* assigned by the primary) is divisible by the constant cp (checkpoint period). After the replica j produces the checkpoint it multicasts $\langle \text{CHECKPOINT}, j, UI_{last}, d, UI_j \rangle$ where UI_{last} is the unique identifier of the last executed request, d is the hash of the 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 signed by different replicas with the same UI_{last} and d. We call this set of messages a *checkpoint certificate*, which proves that the replica'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_{last}.

The checkpoint is used to limit what messages are added to 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 last 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.

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[2].

When a server s_i receives $f+1$ REQ-VIEW-CHANGE messages, it moves to view $v+1$ and multicasts $\langle \text{VIEW-CHANGE}, s_i, v', C_{last}, O, UI_i \rangle$, where C_{last} is the latest stable checkpoint certificate and O is a set of COMMIT messages sent by the replica since the last checkpoint was generated. At this point, a 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_{last} 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 $\langle \text{VIEW-CHANGE}, s_i, v', C_{last}, O, UI_i \rangle$ messages that are consistent with the system state: (1) the checkpoint certificate C_{last} contains $f+1$ valid *UI* identifiers; (2) the counter value (cv_i) in UI_i is $cv_i = cv+1$, where cv is the highest

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

counter value of the *UI*s signed by the replica in O; if O is empty the highest counter value will be the *UI* in C_{last} signed by the replica when it generated the checkpoint; and (3) there are no holes in the sequence number of COMMIT 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. The primary of v' uses the information in the C_{last} and O fields in the VIEW-CHANGE messages to define S, which is the set of requests that were prepared/accepted since the checkpoint, in order to define the initial state for v'. 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 in the sequence number of the last request that it executed and 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 [12]).

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 $v+2$ [3]. If additional view changes are needed, the timer is multiplied by two each time, increasing exponentially until a new primary server respond. 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 Appendix A.

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

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 [24].

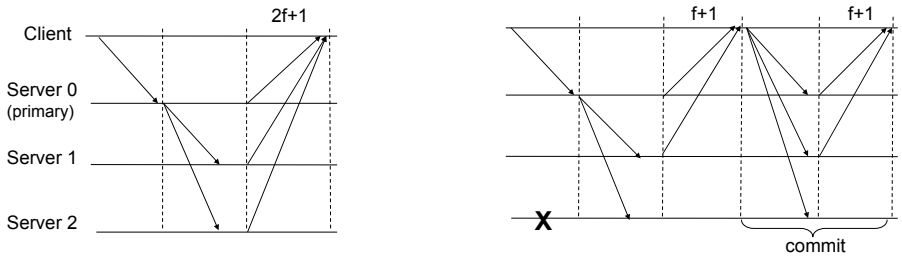
The idea of *speculation* is that servers respond to clients' requests without first agreeing on the order in which the requests are executed. They optimistically adopt the order proposed by the primary server, execute the request, and respond immediately to the client. This execution is speculative 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 last 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. Notice also that the COMMIT messages does not contains the message history (as in Zyzzyva [24]), since the UI_p keeps the sequence of operations.

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



(a) Gracious execution (b) Faulty replica case

Figure 2: MinZyzzyva basic operation.

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 last request received. In this situation, it sends to the primary a $\langle \text{FILL-HOLE}, s_i, v', max_{cv}+1, cv \rangle$ 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. 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. 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.

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[4].

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. A commit certificate is 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* detailed above, (2) obtained when a view change occurs or (3) obtained from a set of $f+1$ matching checkpoints.

Like in MinBFT, replicas generate checkpoints periodically when the counter value in a UI generated by the primary is divisible by the constant cp. After the replica 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 last 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 $\langle \text{VIEW-CHANGE}, s_i, v', C_{last}, O, UI_i \rangle$, where C_{last} is the latest commit certificate collected by the replica and O is a set of ordered requests since C_{last} 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 ex-

[4]Therefore Zyzzyva's *proof of misbehavior* [24] is not needed in MinZyzzyva.

ecution. At this point, the replica stops accepting messages other than CHECKPOINT, VIEW-CHANGE and NEW-VIEW.

Like in MinBFT, correct servers only consider $\langle$VIEW-CHANGE$, s_i, v', C_{last}, O, UI_i\rangle$ messages that are consistent with the system state: (1) the checkpoint certificate C_{last} contains $f+1$ 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_{last} signed by the replica when it generated the checkpoint; and (3) there are no holes in the sequence number of COMMIT messages in O.

When the new primary receives $f+1$ VIEW-CHANGE messages it multicasts $\langle$NEW-VIEW$, s_i, v', V_{vc}, S, UI_i\rangle$ message to define the initial state to v'. When a replica receives a NEW-VIEW message it verifies if the *new-view certificate* is valid and if S was computed properly. A replica begins in the new view v' when all requests in S that have not been executed before, are executed. 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 did not receive neither $2f+1$ RESPONSE messages, nor $f+1$ LOCAL-COMMIT messages, thus the operations did not complete. Rollback is a 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 rollback 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 Appendix B.

6 Optimizations

The basic algorithm presented multicasts each request in a PREPARE message. Like PBFT and other BFT algorithms in the literature, the cost of this operation can be greatly reduced by *batching* several requests in a single message. There can be several policies for how to batch requests. In a LAN the execution of `createUI` can be the bottleneck of the algorithm, so while the primary is executing this function it can go on batching requests that will be sent together in the next PREPARE message.

Another optimization that can have a considerable impact in the performance of the algorithm is not sending the complete requests in the PREPARE and COMMIT messages, but sending only their hash. This requires that no correct server sends a COMMIT message before receiving the client request (the client and/or the primary can be faulty and not send the request to a backup). When a replica receives PREPARE/COMMIT messages without having received the corresponding request, it has to get the request from one of the servers.

The communication between clients and servers is signed with digital signatures. The replies from servers to the clients can be signed in a faster way with MACs generated with a secret key shared between the server and the client.

7 Implementation

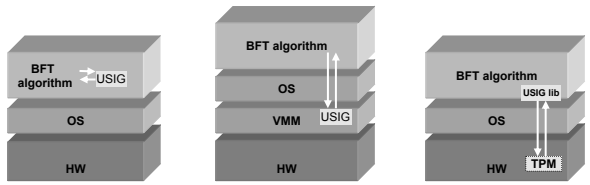
We implemented the prototypes for both MinBFT and MinZyzzyva in Java[5] We chose Java for three reasons. First, we expect that avoiding bugs and vulnerabilities will be more important than performance in most BFT deployments, and Java offers features like memory protection, strong typing and access control that can make a BFT implementation more dependable. The second reason is to improve the system portability making it easier to deploy 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.

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 [50]) 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.

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 as presented in Figure 3.

- *Non-secure USIG:* The non-secure version of USIG 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.

[5] Code available at `http://sites.google.com/site/minibft/`



(a) Non-Secure (b) VM-based (c) With TPM

Figure 3: Different versions of the USIG service: NS (Non-secure) - the service is inside the process; VM (Virtual Machine) - the service is in an isolated virtual machine; TPM - the service is supported by trusted hardware.

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 [6] 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 (see Section 2). 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* signature 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 version of the USIG service: NS-Hmac, NS-Sign(ESIGN), NS-Sign(RSA), VM-Hmac, VM-Sign(ESIGN), VM-Sign(RSA) 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 against physical attacks.

8 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.

We also present performance results for a macro-benchmark that was used to understand the impact of using our algorithms in a real application: we integrated MinBFT and MinZyzzyva with the Java Network File System (JNFS) [38].

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[6]. 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 executed from 0 to 120 logical clients distributed through 6 machines. The servers and clients machines were 2.8 GHz Pentium-4 PCs with 2 GBs RAM running Sun JDK 1.6 on top of Linux 2.6.18 connected 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

[6] `http://www.pmg.lcs.mit.edu/bft/`.

run only in normal case operation, without faults or asynchrony, which is usually considered to be the normal case.

8.1 Micro-Benchmarks

For the first part of the performance evaluation we chose the versions of MinBFT and MinZyzzyva that presented best performance and we compare them with PBFT. We evaluated the performance of four algorithms PBFT, JPBFT, MinBFT-Hmac and MinZyzzyva-Sig(ESIGN) on a LAN. 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 show in Table 2.

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

Table 2: Latency results varying request and response size for the best versions of MinBFT and MinZyzzyva, together with PBFT and a similar Java implementation.

In this experiment, PBFT has shown the best performance of all algorithms/implementations, followed by JPBFT, MinBFT-Hmac and MinZyzzyva-Sig, which was the worse. This experiment shows clearly that our Java implementation runs an agreement much slower than PBFT. One of the possible reasons for this can be the overhead of our event-driven socket management layer that maintains several queues and event listeners to deal smoothly with a high number of connections. 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 [24, 44], but Zyzzyva (like PBFT) uses only MACs, 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.

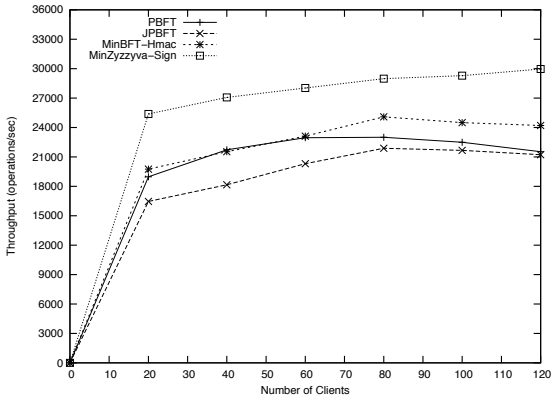


Figure 4: Peak throughput for 0/0 operations for the best versions of MinBFT and MinZyzzyva, together with PBFT and a similar Java implementation.

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 minimal number of communication steps and replicas (which reduces the quorum sizes used by the algorithms) makes the replicas process less messages (less I/O, less MAC verification, etc.), which increases the throughput. Due to the optimizations for scalability discussed in Section 7, 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.

8.2 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 is not what is expected when one consider the data centers used today (in which all replicas are inside a data center), it is very important 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*[7] 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.

[7] `http://www.linuxfoundation.org/en/Net:Netem`.

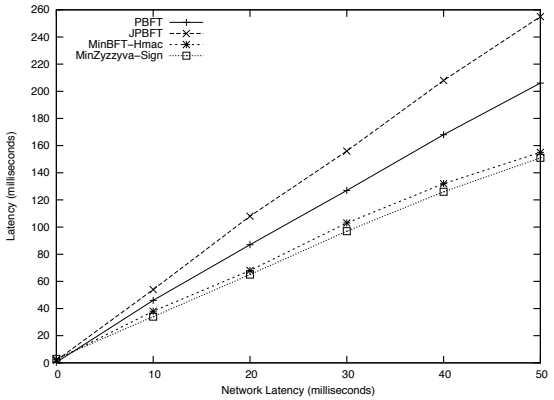


Figure 5: Latency for 0/0 operations with several link latency values.

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. However, the use of smaller quoruns can make our algorithms more efficient in real networks due to their large variance in link latency [22].

8.3 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 our different USIG implementations in a LAN, excluding the TPM-based USIG that will be discussed 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 MAC-based authenticators.

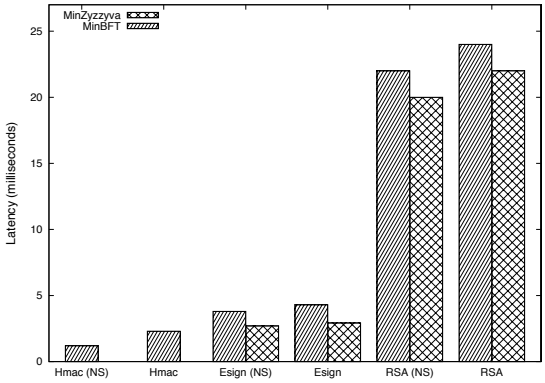


Figure 6: Latency of 0/0 operations for MinBFT and MinZyzzyva using several USIG implementations.

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

Table 3: Overhead (in milliseconds) 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 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 NS-USIG (with has no access cost). The data in this table explains the results observed on Figure 6: roughly, the use of ESIGN adds 2.5 ms to the latency of MinBFT when compared with Hmac, while RSA adds 17.5 ms when compared with ESIGN. As a side note, the same RSA implementation when run on a 64-bit 2.3GHz quadcore Xeon machine, with a 64-bit JVM, execute `createUI` and `verifyUI` in 2.496 and 0.128 ms, respectively, a gain of 80% when compared with the results of the table. This allow us to conclude that asymmetric crypto can be used successfully in high-end servers, specially if one can make separate threads run the tasks of signature generation.

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.

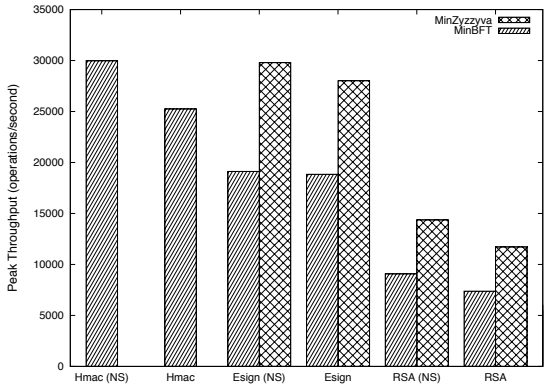


Figure 7: Peak throughput for 0/0 operations in for MinBFT and MinZyzzyva using several USIG implementations.

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.

8.4 Hardware-based USIG Performance

Table 4 presents the latency and peak throughput of the USIG service implemented on a Atmel TPM 1.2 chip integrated to the mainboard of machines.

Algorithm	Latency	Peak Throughput
MinBFT	1617	23404
MinZyzzyva	1552	24179

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 [9]. Others have also been pushing for faster TPM cryptography [31].

8.5 Macro-Benchmark

To explore the costs of the algorithms in a real application, we integrated them with JNFS, an open source implementation of NFS that runs on top of a native file system [38]. We compare the latencies obtained with a single server running plain JNFS and with three different replication scenarios integrating JNFS with JPBFT, MinBFT-Hmac and MinZyzzyva-Sig.

The macro-benchmark workload consists of five phases: (1) create/delete 6 subdirectories recursively; (2) copy/remove a source tree containing 7 files with up 4Kb; (3) examine the status of all files in the tree without examining their data (returning information as owner, size, date of creation); (4) examine every byte of data in a file with 4Kb size; (5) create a 4Kb file.

Table 5 shows the results of the macro-benchmark execution. The values are the mean of the latencies of 200 runs for each phase of the workload in two independent executions. The standard deviations for the total time to run the benchmark with MinBFT and MinZyzzyva were always below 0.4% of the value reported. Note that the overhead caused by the replication algorithms is uniform across the benchmark phases in all algorithms. The total

time of an operation in a replication scenario is defined by the operation time observed in JNFS in a single server plus the algorithm latency. The main conclusion of the macro-benchmark was that the overhead introduced by the replication is not too high (no more than 3%).

Phase	JNFS	JPBFT	MinBFT-Hmac	MinZyzzyva-Sig
1	26	28	29	29
2	681	685	687	688
3	20	22	23	23
4	5	7	8	8
5	108	111	113	114
Total	840	852	860	862

Table 5: Macro-benchmark: latencies of JNFS alone and replicated with BFT algorithms (milliseconds)

9 Related Work

The idea of tolerating intrusions (or arbitrary/Byzantine faults) in a subset of servers appeared in seminal works by Pease et al. [34] and Fraga and Powell [18]. However, the concept started raising more interest much later with works such as Rampart [39] and PBFT [12].

Most work in the area uses a *homogeneous fault model*, in which all components can fail in the same way, although bounds on the number of faulty components are established (e.g., less than a third of the replicas). With this fault model and an asynchronous time model it has been shown that it is not possible to do Byzantine fault-tolerant state machine replication with less than $3f + 1$ replicas [46].

The idea of exploring a *hybrid fault model* in the context of intrusion tolerance or Byzantine fault tolerance, was first explored in the TTCB work [15]. The idea was to extend the "normal" replicas that might be faulty with a tamperproof subsystem. This concept was later generalized with the notion of wormholes [48].

It was in this context that the first $2f + 1$ state machine replication solution appeared [14]. The TTCB had the job of ordering the clients' requests. The atomic multicast algorithm did not follow a Paxos-like message pattern, but made destination agreement, i.e., consensus on the order of execution [17].

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 (to implement the A2M abstraction). MinBFT and MinZyzzyva are also $2f + 1$ BFT algorithms but that, on the contrary the previous two, are minimal in the several senses discussed above. Furthermore, A2M-PBFT-EA is a modification of PBFT, while MinBFT is a novel algorithm that is simpler and more efficient in terms of number of communication steps and MACs at the bottleneck server.

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) [51]. 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 [26].

Quorum systems are a way to reason about subsets of servers (quorums) from a group. Quorums can be used to implement data storage in which data can be written and read. These systems are less powerful than state machine replication that is a generic solution to implement (Byzantine) fault-tolerant systems. Martin et al. have shown that it is possible to implement quorum-based data storage with only $2f + 1$ replicas [30].

The main quest in BFT algorithms has been for speed. PBFT has shown that these algorithms "can be fast" [12] but others appeared that tried to do even better. HQ combined quorum algorithms with PBFT with very good performance when the operations being done do not "interfere" [16]. Another similar algorithm, Q/U, uses lighter, quorum-based algorithms, but does not ensure the termination of the requests in case there is contention [1]. Very recently Zyzzyva exploits speculative execution to reduce the number of communication steps and cryptographic operations establishing a new watermark for the performance of these algorithms [24]. An instructive comparison of these algorithms based on simulations was recently published [44].

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. Dijk et al. addressed the problem of using an untrusted server with a TPM to provide trusted storage to a large number of clients [47]. 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.

10 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 in the cost of the system. We show that using a minimal 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 minimal, 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.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, Oct. 2005.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, Feb. 1999.

[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] 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.

[15] 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.

[16] 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.

[17] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, Dec. 2004.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] S. Kinney. *Trusted Platform Module Basics*. Elsevier, 2006.

[24] 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.

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

[26] 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.

[27] B. Littlewood and L. Strigini. Redundancy and diversity in security. In P. Samarati, P. Rian, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004, 9th European Symposium on Research Computer Security*, LNCS 3193, pages 423–438. Springer, 2004.

[28] J. Marchesini, S. Smith, O. Wild, and R. MacDonald. Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear. Computer Science Technical Report TR2003-476, Dartmouth College, Dec. 2003.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[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] M. J. Radwin. Java network file system. http://www.radwin.org /michael/projects/jnfs/paper/jnfs.html.

[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] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 279–292, 2006.

[46] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, Aug. 1984.

[47] 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.

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

[49] G. S. Veronese, M. Correia, A. Bessani, L. Chung, and P. Verissimo. Minimal Byzantine fault tolerance. Technical report, Mar. 2008. http://sites.google.com/site/minibft/.

[50] 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 - SOSP '01*, pages 230–243, 2001.

[51] 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.

[52] 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.

A MinBFT Correctness

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

Safety The proof that MinBFT satisfies the safety properties is the following.

Lemma 1 *In a view v, if a correct server executes an operation o with sequence number i, no correct server will execute o with sequence number* $i' \neq i$.

Proof: If a correct server s executes o with sequence number i, it must have received $f+1$ valid COMMIT messages for $\langle o,i\rangle$ from a quorum Q of servers. The proof is by contradiction. Suppose there is another correct server s' that executes o with sequence number $i' > i$. By the MinBFT algorithm, this can only happen if it receives $f+1$ valid COMMIT messages for $\langle o,i'\rangle$, from a quorum Q' of $f+1$ servers. Since $n = 2f+1$ and $|Q|+|Q'| = 2f+2 \geq 2f+1$, there must be at least one server r (called *intersection server*) that sends COMMIT messages both for $\langle o,i\rangle$ and $\langle o,i'\rangle$. Assuming that the primary on view v is the server p, we have to consider four cases:

1. *the primary and the intersection server are correct*: in this case it is trivial to see that the primary will not generate two UIs for the same request operation o, so the intersection server will never send two valid COMMIT messages for $\langle o,i\rangle$ and $\langle o,i'\rangle$.

2. *the primary is correct and the intersection server is faulty*: The (faulty) intersection server would only be able to send valid COMMIT messages for $\langle o,i\rangle$ and $\langle o,i'\rangle$ if these messages contained $UI_p = \langle i, H(o)\rangle_p$ and $UI'_p = \langle i', H(o)\rangle_p$, respectively. Since the primary is correct, it will never invoke `createUI` for the same operation o twice and consequently it will never produce these two UIs.

3. *the primary is faulty and the intersection server is correct*: Now the primary will create PREPARE messages containing $UI_p = \langle i, H(o)\rangle_p$ and $UI'_p = \langle i', H(o)\rangle_p$ and send them to the intersection server in order to try to make it send COMMIT messages both for $\langle o, i\rangle$ and $\langle o, i'\rangle$. However, due to the verification of the *seq* field of the request operations, which is part of o, the intersection replica will not accept the second PREPARE for the same operation o, issued by client c, because $o.seq = V_{req}[c]$, and the servers only accept operations from clients if their sequence number is greater than their previous number (i.e., if $o.seq > V_{req}[c]$).

4. *the primary and the intersection server are faulty*: Now both the primary and the intersection server are in collusion to make two servers execute the same operation with different numbers. Suppose that the intersection server r sends $\langle \text{COMMIT}, v, r, s, UI_p, UI_r\rangle$ message to s for $\langle o, i\rangle$ and $\langle \text{COMMIT}, v, r, s', UI'_p, UI'_r\rangle$ message to s' for $\langle o, i'\rangle$. Suppose $UI_p = \langle i, H(o)\rangle_p$ and $UI'_p = \langle i', H(o)\rangle_p$ such that $i < i'$. In this case, there are two cases to consider:

   (a) s' executed some operation with sequence number i. In this case its sequence of operations does not contain a "hole", but since the USIG service does not allow the primary p to generate two *UI*s for different message with the same sequence number, the operation with sequence number i must be o (the same executed by s), and thus s' will not execute o again with i' because $o.seq \leq V_{req}[c]$;

   (b) s' did not execute some operation with sequence number i. In this case the server will only execute o with sequence number i' if it executed all operations with sequence number $< i'$, and since it did not execute any operation with sequence number i, it will halt waiting for this operation to be executed.

Consequently, it is not possible for two clients to executes the same operation with different sequence number in view v. ■*Lemma 1*

Lemma 2 *If a correct server executes an operation o with sequence number i in a view v, no correct server will execute o with sequence number $i' \neq i$ in any view $v' > v$.*

Proof: If a correct server s executes o with sequence number i in a view v, it must have received $f+1$ valid COMMIT messages for $\langle o, i, v\rangle$ from a quorum Q of servers. The proof is again by contradiction. Suppose there is another correct server s' that executes o with sequence number $i' > i$ in a view $v' > v$. By the MinBFT algorithm, this can only happen if it receives $f+1$ valid COMMIT messages for $\langle o, i', v'\rangle$, from a quorum Q' of servers. Since $n = 2f+1$ and $|Q| + |Q'| = 2f+2 \geq 2f+1$, there must be at least one server r (called again *intersection server*) that sends COMMIT messages both for $\langle o, i, v\rangle$ and $\langle o, i', v'\rangle$.

Now let us prove that this leads to a contradiction. For simplicity we start by considering that $v' = v+1$, then we expand the proof for arbitrary values of v'.

First we show that the primary of the new view (p) must assert that o was accepted/executed before v', i.e., that it can not deny this fact. This assertion is done explicitly or implicitly in the new-view certificate V_{vc} that it sends in the NEW-VIEW message that starts view v': $\langle \text{NEW-VIEW}, p, v', V_{vc}, S, UI_i\rangle$. This certificate is composed by $f+1$ VIEW-CHANGE messages that p received from a quorum Q'' with that many servers, one of which must be correct. Consider a server $r \in Q''$ for which the VIEW-CHANGE message included in V_{vc} is $\langle \text{VIEW-CHANGE}, r, v', C_{last}, O, UI_s\rangle$. We have to consider four cases:

1. *the primary p is correct and there is a correct server $r \in Q''$ that executed o*: if p is correct it inserts in V_{vc} $f+1$ VIEW-CHANGE messages, including the one that comes from r. There are two possibilities:

   (a) *o was executed after the last stable checkpoint*: r is correct so O contains the COMMIT message that r sent for o, therefore V_{vc} and S assert explicitly that o was executed.

   (b) *o was executed before the last stable checkpoint*: the execution of o is implicit in C_{last} so V_{vc} asserts implicitly that o was executed.

2. *the primary p is correct but there is no correct server in Q'' that executed o*: In this situation at least one faulty server $r \in Q''$ accepted o because $|Q| + |Q''| = 2f+2 \geq 2f+1$. Again there are two possibilities:

   (a) *o was executed after the last stable checkpoint*: r might be tempted to not include the COMMIT message for o in O but if it did it p would not put the VIEW-CHANGE message from r in V_{vc}. The reason is that for not putting o in O r would have to do one of two things that would be detected by p: (1) if r executed a request o' after o, r might put the COMMIT message for o' in O but not the message for o, which would leave a "hole" in O that would be detected by p; (2) if r sent the COMMIT message for o with a *UI* with counter value cv, it might not put in O any COMMIT with $cv' \geq cv$, but that would be detected by p because r would have to sign the VIEW-CHANGE message with a *UI* with counter value $cv'' > cv+1$. Therefore, for the VIEW-CHANGE message from r to be inserted in V_{vc} by p, r must include the COMMIT message for o in O, falling in case 1a above.

(b) *o was executed before the last stable checkpoint*: in this situation the execution of o is implicit in the certificate of the last stable checkpoint (see case 1b above). The faulty server r may attempt to put an older checkpoint in the VIEW-CHANGE message but p will never insert this message in V_{vc} because r would have to do one of the two detectable things pointed out in case 2a. Therefore, we fall in case 1b.

3. *the primary p is faulty but there is a correct server $r \in Q''$ that executed o*: in this case the faulty primary may attempt to modify the content of O that it inserts in V_{vc}. If it simply removes o from O it leaves a hole, which is detectable. If it removes o and all later messages this is also detectable because p can not forge a UI from r with the following counter value. If p tries to substitute the checkpoint certificate C_{last} for an older one it also can not forge the UI. Even if it substitutes the checkpoint for an older checkpoint sent by r, this is detectable (servers known that r sent messages afterwards). This shows that these attacks are detectable so we have only to show that they are indeed detected. This is the case because when a correct server receives a NEW-VIEW message it checks the validity of V_{vc}. Therefore, a faulty primary can not tamper the content of the correct server VIEW-CHANGE message so we fall in case 1.

4. *the primary p is faulty and there is no correct server that executed o in Q''*: even if there is no correct server that executed o in Q'', there must be one faulty server $r \in Q''$ that executed o because $|Q| + |Q''| = 2f + 2 \geq 2f + 1$. For case 2 we already showed that r can not make the primary believe that it did not execute o. However, in this case the primary p is faulty so p can insert the VIEW-CHANGE message sent by r in V_{vc} anyway. However, this falls in case 3 because correct servers will validate V_{vc} when they receive the NEW-VIEW message from p. Therefore we end up falling in case 1.

This shows that the primary p of the new view v' must assert that o was accepted/executed before v' in the new-view certificate V_{vc}. Now we prove that no correct server will execute o with sequence number $i' \neq i$ in view v'. We have to consider two cases:

1. *the primary is correct*: as already shown, the primary must know that o was executed so it will never generate a second UI for the same request and correct servers will not send a COMMIT message for o in view v'.

2. *the primary is faulty*: in this case the primary can create a new PREPARE message containing $UI'_p = \langle i', H(o) \rangle_p$ and send it to the servers, say, to r. However, r will verify the request number in the *seq* field of o and discover that $o.seq \leq V_{req}[c]$ meaning that the request was already executed, so it will not execute it again.

This proves the lemma for $v' = v + 1$. Now we have to expand for arbitrary values of v'. There are two cases:

1. *$v' = v + k$ but no requests were accepted in any view v'' such that $v' < v'' < v + k$*: this situation can be caused but an instability in the network that leads to several consecutive executions of the view change algorithm. It is trivial to understand that this case falls into the case of $v' = v + 1$ because nothing relevant happens in the views v''.

2. *the generic case where there are "real" views between v and v'*: an analysis of the proof for the case of $v' = v + 1$ shows that the information that is propagated from view v to $v + 1$ about requests that were executed is also propagated to later views, either explicitly in the O sets while there are no checkpoints, or implicitly in the checkpoints. This is the information used to prevent requests from being re-executed so this case falls into the case of $v' = v + 1$.

■*Lemma 2*

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

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

Assume that the theorem is false, i.e., that there is a correct server s' that executed some sequence of operations S' that is not a prefix of S. More formally, assume that $prefix(S', j) = prefix(S, j-1).\langle o'_j \rangle$ and $prefix(S', j-1).\langle o_j \rangle = prefix(S, j)$, with $o'_j \neq o_j$. In this case o'_j is the j-th operation executed in s' and o_j is the j-th operation executed in s. Assume that o_j was executed in view v by s and o'_j was executed in view v' by s'. If $v = v'$, this contradicts Lemma 1, and if $v \neq v'$ it contradicts Lemma 2. Consequently, the theorem holds. ■*Theorem 1*

Liveness In the following we present the proof of liveness for the MinBFT algorithm. We say that an operation request issued by a client c *completes* when c receives the same response for the operation from at least $f + 1$ different servers. We define a *stable view* as a view in which the primary is correct and no timeouts expire at correct replicas.

Lemma 3 *During a stable view, an operation requested by a correct client completes.*

Proof: If the client c is correct it will send its operation o with a sequence number greater than any of its previous requests to all servers. Since, in a stable view the primary p is correct, it will generate an $UI = \langle i, H(o) \rangle_p$ and send it to all servers in a PREPARE message. A correct server will receive this message, verify the validity of *UI* by calling `verifyUI`, and send a COMMIT message for $\langle o, i \rangle$. Since there are at most f faulty servers on the system, there are at least $f+1$ correct servers (the primary plus other f servers) that will produce these COMMIT messages and send them to all servers. When a correct server receives $f+1$ COMMIT messages, it executes o[8]) and send a reply to the client c. When c receives $f+1$ equal replies the operation completes, which must happen since there are $f+1$ correct servers and all of them will produce the same result when executing o as their i-th operation. ■*Lemma* 3

Lemma 4 *A view v eventually will be changed to a new view $v' > v$ if at least $f+1$ correct servers request its change.*

Proof: To request a view change, a correct server s sends a $\langle$REQ-VIEW-CHANGE$, s, v, v' \rangle$ to all servers, where v is the current view number and $v' = v+1$ the new view number. Consider that a quorum of $f+1$ correct servers Q requests this view change from view v to view $v+1$. The primary for the view is by definition $p \triangleq (v+1)$ *mod n*. There are two cases:

1. *the view is stable*: this means that all servers in Q receive the REQ-VIEW-CHANGE messages from each another. When one of these servers (s) receives the $f+1$th of these messages, it sends to all other servers a message $\langle$VIEW-CHANGE$, s, v', C_{last}, O, UI_s \rangle$. All the VIEW-CHANGE messages sent by servers in Q are received by all servers. The primary p for view $v+1$ is correct so it sends a message $\langle$NEW-VIEW$, p, v', V_{vc}, S, UI_p \rangle$ to all servers. No timeouts expire (the view is stable) so all servers receive this message and the view changes to $v+1$.

2. *the view is not stable*: this case can be divided in two cases:

   (a) *p is faulty and does not send the* NEW-VIEW *message, or p is faulty and sends an invalid* NEW-VIEW *message that is discarded by all correct servers, or p is not faulty but the communication is slow and the timeout expires in all correct servers*: when the servers send a VIEW-CHANGE message they start a timer that expires after T_{vc} units of time. In this case this timeout expires at all correct servers, which start another view change.

   (b) *p is faulty and sends the* NEW-VIEW *message but only to a quorum Q' with at least $f+1$ servers but less than $f+1$ correct, or p is correct and the same effect happens due to communication delays*: in this case faulty servers in Q' can follow the algorithm making the correct servers in Q' believe that the algorithm is running normally. More precisely, the servers in Q' can exchange PREPARE and COMMIT messages following the algorithm. At the correct servers that are not in Q', a timer will expire after T_{vc} units of time and these servers will send REQ-VIEW-CHANGE messages, but there will not be $f+1$ one of them so a view change will not happen. When faulty servers start to deviate from the algorithm, requests will stop being accepted, the correct servers in Q' will send REQ-VIEW-CHANGE messages and a view change will start.

In these last two cases (2a and 2b), when another view change starts the system can fall again in one of the cases 1 or 2. However, eventually the view will become stable, the system will fall in case 1 and the view will be changed to a new view $v' > v$. ■*Lemma* 4

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

Proof: The proof comes from the previous lemmas. In stable views, operations requested by correct clients eventually complete (Lemma 3). If the view v is not stable, there are two possibilities:

1. *timers expire and at least $f+1$ correct servers request a view change*: in this case the view will be changed to a new view $v' > v$ (Lemma 4).

2. *timers expire but less than $f+1$ correct servers request a view change*: this case is similar to case 2b of Lemma 4. If there is a quorum Q of at least $f+1$ servers that do not request the view change and that go on following the algorithm in view v, exchanging PREPARE and COMMIT messages, then the system will stay in view v and requests from correct clients will be executed. When there is no such a quorum or requests are not executed within T_{exec}, all correct servers request a view change and we fall in the previous case.

■*Theorem* 2

[8] Since the primary p is correct, when it was elected it disseminated any pending requests of the previous view, and thus this server will not have holes in its sequence of operations and will be able to execute the request immediately.

B MinZyzzyva Correctness

The two properties that we have to prove about MinZyzzyva are the same that were proved for Zyzzyva [24]. These properties are defined from the point of view of what is observed by a client. Informally, a request is said to have complete if the client can use the reply to that request, i.e., if the client can be certain that the (speculative) execution of that request will not be rolled back. Formally, a request is said to have *complete* at a client if the client received $2f+1$ matching RESPONSE messages or $f+1$ matching LOCAL-COMMIT messages for the request. The properties that MinZyzzyva has to satisfy are:

Safety: If a request with sequence number *seq* and history h_{seq} completes, then any request that completes with a higher sequence number $seq' > seq$ has a history $h_{seq'}$ that includes h_{seq} as a prefix.

Liveness: Any request issued by a correct client eventually completes.

Safety The proof that MinBFT satisfies the safety properties is the following.

Lemma 5 *In a view v, if a correct server executes an operation o with sequence number i, no correct server will execute o with sequence number $i' \neq i$.*

Proof: Despite the differences of MinZyzzyva and MinBFT, the proof of this lemma is similar to the proof of Lemma 1. The basic idea is that replicas process the primary messages by order of counter value in *UI* and the primary can not associate the same *UI* to two different requests due to the properties of the USIG service. ■*Lemma 5*

Theorem 3 *If a request with sequence number seq and history h_{seq} completes, then any request that completes with a higher sequence number $seq' > seq$ has a history $h_{seq'}$ that includes h_{seq} as a prefix.*

Proof: Consider that the request o with sequence number *seq* completes in view v and o' with sequence number seq' in view v'. We have to consider two cases:

1. $v = v'$: the proof is by contradiction. Assume that $h_{seq'}$ does not include h_{seq} as a prefix. This is in contradiction with Lemma 5.

2. $v \neq v'$: suppose that $v' = v+1$ (the expansion for an arbitrary relation $v' > v$ is simple and similar to the one done in the proof of Lemma 2). First we have to show that the primary of the new view (p) must assert that o was completed before v', i.e., that it can not deny this fact. This assertion is done explicitly or implicitly in the new-view certificate V_{vc} that it sends in the NEW-VIEW message that starts view v'. However, the view change operation of MinZyzzyva is almost identical to the same operation of MinBFT so the proof is the same as the one made in the proof of Lemma 2 and we skip it here. Then, we have to consider two cases:

   (a) *o' is the first request executed in view v'*: when the new view is installed the NEW-VIEW message serves as a commit certificate to the requests completed in view $v'-1$. Therefore, $h_{seq'-1}$ is committed and is a prefix of $h_{seq'}$ (the theorem states that o' completes).

   (b) *o' is not the first request executed in view v'*: this is trivially proved by induction considering case 2a as the base case and using Lemma 5 to prove the induction step.

■*Theorem 3*

Liveness Now we prove the liveness of MinZyzzyva.

Lemma 6 *During a stable view, an operation requested by a correct client completes.*

Proof: If the client c is correct it will send its operation o with a sequence number greater than any of its previous requests to all servers. Since, in a stable view the primary p is correct, it will generate an $UI = \langle i, H(o) \rangle_p$ and send it to all servers and to the client. A correct server will receive this message, verify the validity of *UI* by calling `verifyUI`, and send a RESPONSE message for $\langle o, i \rangle$ to the client. Since there are at most f faulty servers on the system, there are at least $f+1$ correct servers (the primary plus other f servers) that will produce these RESPONSE messages and send them to the client. There are two cases:

1. *no faulty servers*: the client receives $2f+1$ RESPONSE messages and the operation completes.

2. *there are faulty servers*: the client receives between $f+1$ (stable view) and $2f$ RESPONSE messages. When the timer expires the client sends a COMMIT message to the servers, all correct servers reply with a LOCAL-COMMIT message, and the operation completes.

■*Lemma 6*

Lemma 7 *A view v eventually will be changed to a new view $v' > v$ if at least $f+1$ correct servers request its change.*

Proof: The view change operation of both algorithms is similar so this proof is similar to the proof of Lemma 4. ■*Lemma 7*

Theorem 4 *An operation requested by a correct client eventually completes.*

Proof: The proof comes from the previous lemmas similarly to the proof of Theorem 2. In stable views, operations requested by correct clients eventually complete (Lemma 6). If the view v is not stable, there are two possibilities:

1. *at least $f+1$ correct servers suspect of the primary and request a view change*: in this case the view will be changed to a new view $v' > v$ (Lemma 7).

2. *less than $f+1$ correct servers suspect of the primary and request a view change*: this case is similar to case 2b of Lemma 4. If there is a quorum Q of at least $f+1$ servers that do not request the view change and that go on following the algorithm in view v, then the system will stay in view v and requests from correct clients will be executed. When there is no such a quorum or requests are not executed within the timeout, all correct servers request a view change and we fall in the previous case.

■*Theorem* 4

C Implementing USIG-Sign with the TPM

The simplicity of the USIG service allows it to be implemented with the TPM. The implementation of the service requires TPMs compliant with the Trusted Computing Group (TCG) 1.2 specifications [36, 37]. We assume that the TPMs are tamperproof, i.e., resistant to any attacks. In reality TPMs are not secure against physical attacks [28] so we assume an attacker never has physical access to the servers and their TPMs (attacks come through the network, e.g., from the Internet).

The kind of the USIG service that can be implemented using the TPM is USIG-Sign, i.e., the kind of USIG service that is based on public-key cryptography. TPMs have the ability to sign data using the private key of the *attestation identity key pair* (AIK), which we call *private AIK* for simplicity and that can never leave the TPM (there is no API that allows extracting it from the TPM). We assume that servers know the other TPMs' AIK public keys so they can verify the signatures produced.

We explore mainly two features defined in the TPM 1.2 specification [36]. The first are the *monotonic counters*. The TPM provides two commands on counters: `TPM_ReadCounter` that returns the counter value, and `TPM_IncrementCounter` that increments the counter and returns its new value [37]. No command is provided to set or decrement counters. The TCG imposes that the counters have 32 bits and can not be increased arbitrarily often to prevent that they burn out in 7 years [36]. In the TPMs we used in the experiments, counters could not be increased more than once every 3.5 seconds approximately (and the same is verified in other TPMs [41]). This feature seriously constrains the performance of algorithms that use this implementation of the USIG service.

The second feature is the *transport command suite* [36]. This set of commands protects the communication with the TPM as a process that wants to use the TPM services may not trust software that may interpose between the two. More precisely, using the commands `TPM_EstablishTransport`, `TPM_ExecuteTransport` and `TPM_ReleaseTransportSigned`, it is possible to create a *session* that is used to do a sequence of TPM commands, to log all executed commands, and to obtain a hash of this log along with a digital signature of this same hash obtained with the private AIK [37, 23]. The communication between the process and the TPM is protected using common mechanisms like message authentication codes (MACs) produced with hash functions and nonces. From the point of view of our algorithms, the important is that the `TPM_ReleaseTransportSigned` command returns a proof that a set of commands was executed by the TPM. This proof can be verified by holders of the public AIK.

The interface of the USIG service (functions `createUI` and `verifyUI`) does not change in the TPM-based implementation. However, the service is not completely implemented inside the TPM, which can not be modified, but by the TPM plus a thin software layer on top of it. This layer does *not* have to be trusted.

The implementation of the USIG service on top of the TPM is straightforward. The function `createUI`(m) is implemented the following way:

1. calculate a hash of m;

2. start a session in the TPM by calling `TPM_Establish Transport` and `TPM_ExecuteTransport`;

3. ask the TPM to increment the counter by calling `TPM_IncrementCounter`, assuring that all messages are assigned a sequential number (concurrency is not an issue as no two sessions can be open simultaneously in the same TPM);

4. end the session by calling `TPM_ReleaseTransport Signed`, which takes as parameter the hash of the message (antiReplay parameter), and produces a signature of a data structure that includes the monotonic counter value, the hash of m and the hash of the transport session log;

5. return a data structure with all those items plus the signature that is what we call the *unique identifier UI*.

Notice that the USIG certificate we mentioned when first describing the `createUI` is now composed by the signature and the hash of the transport session log. The latter is used to prove that the TPM increased the counter. Therefore, we do not have a tamperproof service that always increment the counter, but a non-tamperproof layer of software that requests the tamperproof hardware to increment the counter and give a proof that it did it.

The function `verifyUI`(PK, UI, m) is implemented entirely in software, i.e., outside the TPM. It is implemented the following way:

1. calculate the hash of m and check if this hash is equal to the hash in UI;
2. use the TPM's public AIK (PK) to check if the signature was produced from the hash of m together with the hash of the transport log and the hash of the monotonic counter (both part of UI);
3. check if the log contains the call to the `TPM_IncrementCounter` command;
4. if some of the checks fail, return *false*, otherwise return *true*.