Minimal Byzantine Fault Tolerance

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

DI–FCUL TR–08–29

Novembro 2008

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

Giuliana Santos Veronese[1], Miguel Correia[1], Lau Cheuk Lung[2],
Alysson Neves Bessani[1], Paulo Verissimo[1]
[1] *Universidade de Lisboa, Faculdade de Ciências, LASIGE, Portugal*
[2] *Universidade Federal de Santa Catarina, Departamento de Informática e Estatística, Brazil*

Abstract

This paper presents two 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: it provides an interface with a single function and is composed only by a counter and a signature generation primitive. 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. The paper is also the first to present BFT algorithms with $2f+1$ replicas that require a trusted service implementable using commercial off-the-shelf trusted hardware: this service can be implemented with the Trusted Platform Module currently available as a chip in the mainboard of many commodity PCs.

1 Introduction

The complexity and the 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 *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, 11, 12, 14, 16, 18, 23, 25, 28, 38, 48]. These algorithms have been used to design *intrusion-tolerant* services such as network file systems [11, 48], cooperative backup [4], large scale storage [3], secure DNS [10], coordination services [7], certification authorities [49] and key management systems [39].

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 [41] and later extended for Byzantine/arbitrary faults [38, 11]. The algorithms of the latter category are usually called simply BFT[1].

Minimal number of replicas BFT algorithms typically require $3f+1$ *servers* (or *replicas*[2]) to tolerate f Byzantine (or *faulty*) servers [11, 16, 23, 38]. 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 [26, 32]. 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.

Minimal trusted service A few years ago, some of the authors of the paper proposed the first BFT algo-

[1] In the paper we use BFT to mean specifically "Byzantine fault-tolerant state machine replication algorithms". There are many algorithms in the literature that are Byzantine fault-tolerant but that provide weaker semantics, e.g., registers implemented with quorum systems [29]. When we speak about BFT in the paper, we do not include those.

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

rithm with only $2f+1$ replicas [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 [12]. 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 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 that it is simple (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: it provides an interface with a single function and internally it is composed only by a *counter* (only to be increased by one at a time) and a *digital signature* generation primitive (e.g., RSA or ESIGN).

Table 1 summarizes the characteristics of the MinPBFT and MinZyzzyva, algorithms proposed in the paper and also presents a comparison of them with the main BFT algorithms in the literature.

	Replic.	Steps	Tr.Compon.	Speculat.?
PBFT	$3f+1$	5	no	no
Zyzzyva	$3f+1$	3	no	yes
BFT w/TTCB	$2f+1$	5	TTCB	no
A2M-PBFT-EA	$2f+1$	5	A2M	no
MinPBFT	$2f+1$	4	USIG/TPM	no
MinZyzzyva	$2f+1$	3	USIG/TPM	yes

Table 1: Comparison of some 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 one thousand times higher than in LANs[3].

The first algorithm we propose – MinPBFT – follows a message exchange pattern similar to PBFT's [11]. 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 MinPBFT 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 *USIG-certificate* is signed using the trusted component private key, so it can be verified by the other replicas using the corresponding public key. The service is called *Unrepeatable Sequential Identifier Generator (USIG)*. The core of MinPBFT is a *Confirmable Reliable Multicast (CRM)* primitive, which requires only $2f+1$ servers, instead of the $3f+1$ of reliable multicast in homogeneous systems, i.e., in systems with no "special" (tamperproof) component [8].

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 [23].

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. MinPBFT 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 [28, 23]. 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 may not be possible in some applications.

At this stage it is important to comment that there are no free lunches and that these improvements have their drawbacks also. For one, in relation to "classical" BFT algorithms that do not use a trusted component, our al-

[3] And in fact to tolerate disasters, replicas must be stored in different buildings or even different cities, requiring high communication delays.

gorithms (and the other two in the literature) 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. The second issue, is that we need to use signatures based on public key cryptography, unlike PBFT and others. This has a cost in terms of performance, specially in LANs, in which the communication delays are low.

COTS trusted hardware Another important aspect of the algorithms presented in the paper is that they are the first BFT algorithms with $2f+1$ replicas that require only commercial off-the-shelf trusted hardware: the trusted service they need can be implemented with the *Trusted Platform Module (TPM)*. The TPM was designed by the Trusted Computing Group[4] and is currently available as a chip in the mainboard of many commodity PCs. The TPM provides services like secure random number generation, secure storage and digital signatures [35]. 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.

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

- 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 presents the first Byzantine fault-tolerant state machine replication algorithm with $2f+1$ replicas using only COTS trusted hardware currently shipping in commodity PCs, also showing a novel application for the TPM.

2 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. Servers and clients know the public keys they need to check these signatures. 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 [26, 32].

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* [47]. 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 signature.

We do not make assumptions about processing or communication delays, except that these delays do not grow indefinitely (like PBFT [11]). This rather weak assumption has to be satisfied only to ensure the liveness of the system, not its safety.

3 USIG Service

The *Unique Sequential Identifier Generator (USIG)* assigns to messages (i.e., arrays of bytes) identifiers that are unique, increasing and sequential (e.g., $562, 563, 564, ...$). The service is implemented in a isolated, tamperproof, component that we assume can not be corrupted. This component contains essentially a *counter* and a digital signature primitive. The private key used to do the signatures is also stored in the component

[4] Formerly TCPA, `https://www.trustedcomputinggroup.org/`

(and can not be disclosed). The corresponding public key (*PK*) is made available some way to all the other servers (and also to the clients in the case of MinZyzzyva). The service never repeats identifiers (there is simply no interface to do it) and never assigns the same identifier to two different messages (executions of the service are sequential and each execution always increases the counter).

The interface of the service has a single function:

`createUI`(m) – increments the value of the counter and produces a signature of a data structure that includes the counter value and the hash of m. Returns a *unique identifier UI*, which is a data structure containing the counter value and the USIG-certificate that certifies that *UI* was created by this component (which includes the signature).

In the algorithms we use another function that however is not part of the service, so it is implemented *outside* of the component:

`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*. The function calculates the hash of m, uses the public key *PK* to check if the signature was produced from the hash of m, and does other verifications that may be needed (see Section 3.2). In general the USIG-certificate is only a signature and no further verifications are needed. The exception is when the USIG service is implemented using the TPM (see Section 3.2). The function returns *true* (valid) or *false*.

3.1 Implementing the USIG Service

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 [12].

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 [42] and Proxos [44].

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 [31]. Flicker and similar mechanisms can be used to implement the USIG service.

Other solutions include using more or less complex hardware appliances, running their own (small) kernel, communicating with the servers though some well-controlled interface, like an RS-232C or USB cable.

3.2 Implementing USIG with the TPM

As mentioned before, the simplicity of the USIG service permits that it is implemented with the TPM, a chip currently available in many PCs. The implementation of the service requires TPMs compliant with the Trusted Computing Group (TCG) 1.2 specifications [35, 36]. We assume that the TPMs are tamperproof, i.e., resistant to any attacks. In reality TPMs are not secure against physical attacks [27] 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). 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 [35]. 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 [36]. 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 [35]. 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 [40]). This feature seriously constrain the performance of algorithms that use this implementation of the USIG service, so later we discuss how this might be improved.

The second feature is the *transport command suite* [35]. 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 [36, 22]. 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 USIG service interface (function `createUI`) 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. calculates a hash of m;
2. starts a session in the TPM by calling `TPM_EstablishTransport` and `TPM_ExecuteTransport`;
3. asks 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. ends the session by calling `TPM_ReleaseTransportSigned`, 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. returns 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`, which as we mentioned before is implemented in software, outside the tamperproof component, is implemented as follows:

`verifyUI`(PK, UI, m) – The function calculates the hash of m and checks if this hash is equal to the hash in UI. Then, it uses 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 the UI. Finnally, it checks if the log contains the call to the `TPM_IncrementCounter` command. If some of these checks fail, the function returns *false*, otherwise it returns *true*.

4 Confirmable Reliable Multicast

One of the problems that any BFT algorithm has to solve is the problem of forcing all the replicas to deliver the same messages. This is known as the *reliable multicast* problem and has been shown to require $3f+1$ servers/processes to be solvable in asynchronous systems with an homogenous Byzantine fault model [8]. In this section we present the *Confirmable Reliable Multicast* (CRM) algorithm, which uses the USIG service to solve reliable multicast with only $2f+1$ servers/processes. Additionally, CRM allows a replica that accepts a message m to prove that it did accept m following the algorithm. We start by presenting CRM because it is in the core of MinPBFT.

Normally reliable multicast is defined considering that the algorithm is executed by a set of *processes*, but here we use instead the word *servers* to let clear that the CRM algorithm is executed only by the servers of the BFT algorithm, never by the clients. CRM is defined in terms of the following properties (for a data message m that is multicasted):

- *Validity:* If a correct server multicasts a message m, then some correct server eventually accepts m.
- *Agreement:* if a correct server accepts a message m, then all correct servers eventually accept m.
- *Integrity:* For any unique identifier UI, every correct server accepts at most one message m with UI, and if *sender(m)* is correct then m was previously multicast by *sender(m)*.[5]
- *Confirmability:* If a correct server accepts a message m, then it can prove that it accepted m following the CRM algorithm.

Algorithm CRM is presented in Algorithm 1. The messages are called PREPARE and COMMIT as this is the role they play in the MinPBFT algorithm (next section). They also include a view number v that is used in MinPBFT but has no purpose in the reliable multicast algorithm.

The basic idea is that a data message m is sent in a PREPARE message to all servers (line 3), and each server resends it to all others in a COMMIT message (line 7) [6]. Each message sent has a unique identifier UI generated by the `createUI` function (lines 2, 6,10), so no two messages can have the same identifier. Servers check if the identifier of the messages they receive are valid for the messages using the `verifyUI` function (lines 5).

If a server did not receive a PREPARE message but it received a COMMIT message with a valid identifier generated by the sender (lines 8-11) then it sends its COMMIT message. This can happen if the sender is faulty

[5] *sender(m)* returns the sender field in the PREPARE message that is used to disseminate m.

[6] For simplicity in the algorithms some messages are said to be sent to "all servers", which includes the sender. However, in practice this can be implemented in a more economic way, starting by not really sending messages to oneself.

Algorithm 1 Confirmable Reliable Multicast

1: **when** s_i wants to multicast a data message m **do**
2: $UI_i = \texttt{createUI}(m)$
3: send $\langle \text{PREPARE}, v, s_i, m, UI_i \rangle$ to all servers

4: **when** s_j receives a $\langle \text{PREPARE}, v, s_i, m, UI_i \rangle$ from s_i **do**
5: **if** $\texttt{verifyUI}(PK_i, UI_i, m)$ **then**
6: $UI_j = \texttt{createUI}(m)$
7: send $\langle \text{COMMIT}, v, s_j, s_i, m, UI_i, UI_j \rangle$ to all servers

8: **when** s_k receives $\langle \text{COMMIT}, v, s_j, s_i, m, UI_i, UI_j \rangle$ **do**
9: **if** s_k did not receive the PREPARE message and $\texttt{verifyUI}(PK_i, UI_i, m)$ and $\texttt{verifyUI}(PK_j, UI_j, m)$ **then**
10: $UI_k = \texttt{createUI}(m)$
11: send $\langle \text{COMMIT}, v, s_k, s_i, m, UI_i, UI_k \rangle$ to all servers

12: **when** s_l receives $\langle \text{COMMIT}, v, s_j, s_i, m, UI_i, UI_j \rangle$ messages from $f+1$ different servers for which $\texttt{verifyUI}(PK_j, UI_j, m)$ **do**
13: accept m

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 data message m is *accepted* by a server following the algorithm if the server receives $f+1$ valid COMMIT messages from different servers for m (lines 12, 13).

CRM produces a certificate that a message has been accepted. This *CRM-certificate* is composed by the message m and the $f+1$ *UI* identifiers used in line 12.

Correctness A proof that the algorithm satisfies the four properties above is provided in the Appendix A, but the rationale is the following.

There are $2f+1$ servers so at least $f+1$ are correct (at most f are faulty). If a correct server multicasts a message m, then at least f other correct servers receive it and reply with COMMIT messages, thus all correct servers receive $f+1$ COMMIT messages and accept m (Validity). No different message can be accepted due to the unique identifier mechanism (Agreement/Integrity). A correct server can prove that it accepted message m by providing the CRM-certificate (Confirmability).

5 MinPBFT

This section presents MinPBFT, 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. All servers follow the same sequence of states if two properties are satisfied:

- *Agreement:* all servers execute the same operations.
- *Total order:* all servers execute the same operations in the same order.

These are precisely the safety properties that BFT algorithm has to enforce (it also has to make progress).

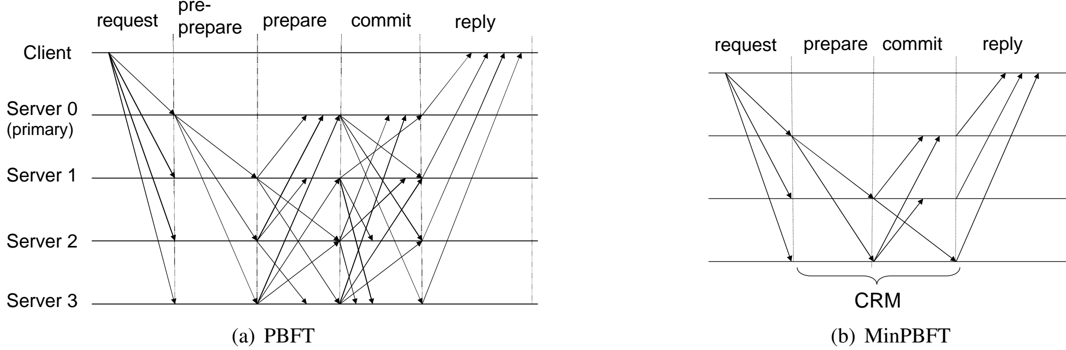
MinPBFT follows a message exchange pattern similar to PBFT, which in turn is similar to Lamport's Paxos algorithm [24] (see Figure 1). The servers/replicas move through successive configurations called *views*. Each view has a *primary* replica and the rest are *backups*. The primary is the server $s_p \triangleq v \bmod n$, 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 using CRM; (3) Each server executes the operation and returns a reply to the client; (4) the client waits for $f+1$ matching responses of the issued operation from different servers.

When at least $f+1$ 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 fails.

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 reply.

Servers: normal case operation The core of the algorithm executed by the servers is the CRM algorithm. When the primary receives a client request, it uses the CRM primitive to multicast the request to all servers (m in Algorithm 1 is the client request). 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



(a) PBFT (b) MinPBFT

Figure 1: PBFT and MinPBFT normal case operation.

there is a view change, an issue that we discuss later.

Unlike PBFT, MinPBFT has only two steps: PREPARE and COMMIT, which are the steps of the CRM algorithm presented above. The CRM algorithm executed in MinPBFT has a slight modification in relation to what is presented in Section 4. The verification in line 5 evaluates to *true* on replica s_j iff two additional conditions are also satisfied:

- the PREPARE message contains a view number v that corresponds to the current view on s_j and its sender is the primary of v;
- the request of the client contains a valid signature produced by that client (to prevent a faulty primary from forging requests).

There is a second modification to CRM. Consider that a primary server multicasts two messages m and m' with two counter values cv and cv' stored in the corresponding UI and UI', with $cv < cv'$. To prevent a faulty primary from creating "holes" in the sequence of messages, no correct backup server sends the COMMIT for message m' without first accepting m. This ensures that not only the requests are *executed* in the order defined by the counter of the primary, but they are also *accepted* in that same order (in normal case operation). In fact, whenever CRM accepts a request, it is executed immediately[7]. The only exception is that if the server is faulty it can "order" the same request twice. So, when a server accepts a request, it first checks in V_{req} if the request was already executed and executes it only if not.

This message ordering mechanism imposes a FIFO ordering and is also enforced to other messages (in the view change operation) that also take a unique identifier UI:

- *FIFO order:* no correct server processes a message $\langle ..., s_i, ..., UI_i, ... \rangle$ sent by any server s_i with counter value cv in UI_i before it has processed message $\langle ..., s_i, ... UI_i', ... \rangle$ sent by s_i with counter value $cv - 1$

[7] Unlike previous BFT algorithms in the literature, in which the messages can be accepted to be executed in an order different from the one in which they will be executed. They can only be executed after all previous are.

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.

Messages sent by a server are kept in a message log in case they have to be resent. To discard messages from this logs, MinPBFT uses a garbage collection mechanism based on checkpoints, very similar to PBFT's. The main difference is only that MinPBFT does not need high/low water marks as the FIFO order imposed with the USIG service constrains the ability of generating future/past sequence numbers.

Optimizations The basic algorithm presented multicasts each request using CRM. 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 CRM. There can be several policies for how to batch requests. In a LAN the execution of `createUI` is probably 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 CRM.

Another optimization that can have a considerable impact in the performance of the algorithm is not sending the complete requests in the COMMIT messages, but sending only their hash. This requires two other modifications to CRM. First, no correct server sends a COMMIT message before receiving the complete request (the client and/or the primary can be faulty and not send the request to a backup). Second, if a server accepts a request before receiving it, it has to get the request from one of the servers (at least one correct server must have it or it would not be accepted).

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.

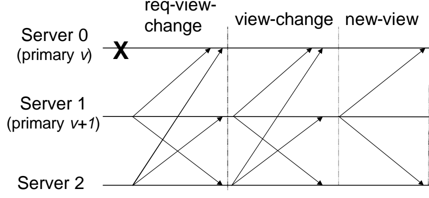


Figure 2: MinPBFT view change.

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 the CRM algorithm. This algorithm strongly constrains what a faulty primary can do: it can not repeat sequence numbers or assign arbitrarily higher sequence numbers. However, a faulty primary can still prevent progress by not assigning sequence numbers to some requests (or even all).

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

A time diagram of the view change operation is depicted in Figure 2. 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[8]. When a server s_i receives $f+1$ REQ-VIEW-CHANGE messages, it moves to view $v+1$ and multicasts $\langle \text{VIEW-CHANGE}, s_i, v', C_{last}, UI_i \rangle$, where C_{last} is the CRM-certificate for the last request accepted.

The 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} to different subsets of the servers, leading to different decisions on which was the last request of the previous view. Faulty processes 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 count $\langle \text{VIEW-CHANGE}, s_i, v', C_{last}, UI_i \rangle$ messages with a CRM-certificate C_{last} that is consistent with the system state: (1) that is valid (contains $f+1$ valid UI identifiers); and (2) that contains a UI' with a counter value $cv' = cv - 1$, where cv is the counter value in UI_i.

When the new primary for view $v+1$ receives $f+1$ VIEW-CHANGE messages from different servers, it stores them in a set V_{vc} and multicasts a message $\langle \text{NEW-VIEW}, s_i, v', V_{vc}, UI_i \rangle$. Then, it uses the information C_{last} fields of the VIEW-CHANGE messages to define which were the clients' requests already received that were not accepted/executed, and multicasts these requests in one or more CRMs.

In previous BFT algorithms, messages are assigned sequential execution order numbers even when there are view changes. This is not the case in MinPBFT 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: it 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 $v+2$ [9]. 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.

Consider two quorums (i.e., subsets) of servers, one that accepted the last executed request, and another that is executing the view change operation (the rest of the servers are slow). PBFT reasons in terms of quorums of $2f+1$ servers, while MinPBFT uses quorums of $f+1$ servers. In contrast with PBFT, that requires at least one correct server in the intersection of the two quorums, in MinPBFT we do not have this requirement. The reason is that the Confirmability property of the CRM algorithms and the FIFO order prevent faulty servers in the intersection of lying about the last request accepted.

A proof of the correctness of the algorithm can be found in Appendix B.

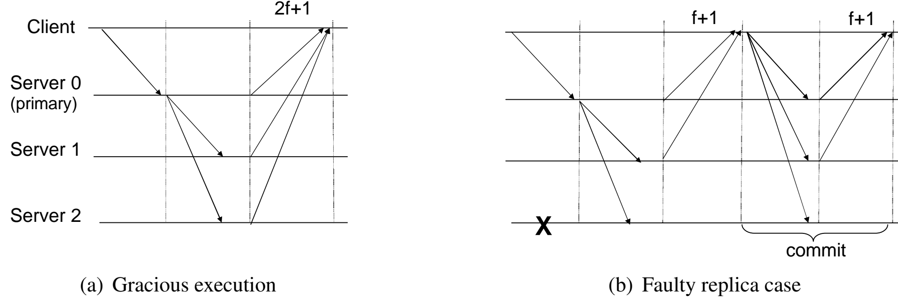
6 MinZyzzyva

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 [23], so it also serves to show that the USIG service can be used with other BFT algorithms to reduce the number of replicas.

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

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

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



(a) Gracious execution

(b) Faulty replica case

Figure 3: MinZyzzyva basic operation.

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. We divide the presentation of MinZyzzyva in three parts: gracious execution, faulty server case and view change. Figure 3 ilustrastes the basic operation of MinZyzzyva.

Gracious execution This is the optimistic mode of the algorithm. It works essentially as follows: (1) A client sends a request 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 MinPBFT), and forwards the request and UI_p to the 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 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, it completes the request and delivers the response to the application. Notice that $2f+1$ are all the servers; this is a requirement for MinZyzzyva (as Zyzzyva) to do gracious execution. Clients use request identifiers (*seq*) to ensure exactly-once semantics, just like in MinPBFT.

Faulty server case 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 UI_p and the 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. The client and 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 unique identifier generator 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 them two different UI_p 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 MinPBFT[10]

View change The view change operation works essentially as MinPBFT's. When a server suspects that the primary is faulty it sends a REQ-VIEW-CHANGE message. When a server receives $f+1$ REQ-VIEW-CHANGE messages, it multicasts a VIEW-CHANGE message. The new primary sends a NEW-VIEW message. Messages are processed in FIFO order like in MinPBFT.

A server constructs a checkpoint every T_{cp} containing UI_p and UI_s identifiers. A server considers the check-

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

point stable when it receives matching checkpoint messages from $f+1$ different servers. Then all messages executed before a given UI_p are removed from the log.

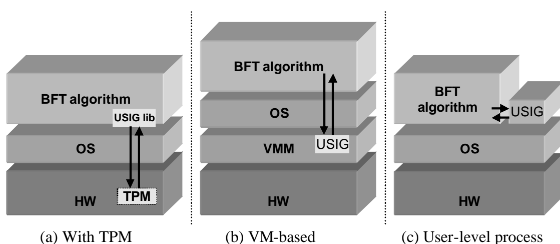
A proof of the correctness of the algorithm can be found in Appendix C.

7 Implementation

We implemented a single prototype, configurable to run as MinPBFT or MinZyzzyva, with one of three implementations of the USIG service: TPM-based, virtual machine based and as a user-level process. The objective was to compare the performance of several ways of implementing the USIG service, but the main motivation is that the current versions of the TPM provide poor performance for two reasons:

1. the TCG's design option of not permitting the monotonic counters to be incremented arbitrarily often to prevent burn out [35] (once every 3.5 seconds in our TPMs and approximately the same in others [40]);
2. the implementation of the digital signature algorithm in current TPM chips is very slow.

We believe both limitations will be solved in the future. Further discussion on this issue can be found in Section 8.2.



(a) With TPM (b) VM-based (c) User-level process

Figure 4: Implementation solutions.

The three different implementations of the USIG service are presented in Figure 4. In all versions the fundamental idea is to isolate the service from the rest of the system (but the levels of isolation obtained are different). In all versions the service has the interface and semantics presented in Section 3. A simple authentication mechanism is used to prevent processes other than the algorithms from accessing the service. We assume 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`. In that case the server is faulty but the algorithms are not affected (as long as no more than f servers are faulty, which is a basic assumption of any BFT algorithm).

The TPM-MinPBFT and TPM-MinZyzzyva versions (Figure 4 (a)) use the USIG service based on the TPM. In these versions the USIG service is implemented by a thin layer of software (a function in a library) and by the TPM itself (see Section 3.2). 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 do low-level access to the TPM [40].

The VM-MinPBFT and VM-MinZyzzyva versions (Figure 4 (b)) run 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, and that we assume can not be corrupted. We use the Xen hypervisor [6] to implement these versions. Xen allows multiple applications to run in different virtual machines. Each virtual machine runs in its own protection domain, providing strong isolation between virtual machines. The algorithms are executed in a virtual machine called *domain1* and might coexist with other applications. The USIG service runs in the *domain0*.

The UP-MinPBFT and UP-MinZyzzyva versions (Figure 4 (c)) use the USIG service implemented as a trusted user-level process. This approach takes advantage of the isolation provided by the operating system, but is not resilient against a corrupted operating system or an attacker with superuser privileges. The USIG is a program that runs as a separated process in the same server.

The prototype was implemented in Java because the level of protection of a program in Java is typically higher than a program in C, due to Java's features like the language being strongly typed and the access control provided by the sandbox implemented by the JVM. The versions based on virtual machines and user-level processes, use NTT ESIGN with 2048-bit keys for doing signatures, instead of RSA, because the performance of ESIGN is much better. We used the NTT ESIGN C++ implementation provided in the Crypto++ library, accessed with the Java Native Interface. In both implementations the communication between the USIG service and the algorithm software is done using sockets. These versions are able to continue to work correctly even under attacks coming from the network against the server software. However only the versions with the TPM are tolerant to a malicious administrator that can manipulate all services hosted by f servers, and even those ones are not tolerant against physical attacks.

PBFT is often considered to be the baseline of BFT algorithms, so we were interested in comparing our algorithms with it. However, for this comparison to make sense we did not use the original implementation, written in C, but made our own implementation of PBFT's normal case operation in Java. In fact, we have a single software prototype that can be configured to run as PBFT, and or any versions of MinPBFT and

MinZyzzyva. PBFT does not use signatures but authenticators based on MACs.

8 Performance Evaluation

This section presents performance results of our algorithms using micro and macro-benchmarks. The micro-benchmark was used to measure the latency and throughput of the MinPBFT and MinZyzzyva implementations using operations that do nothing (null operations). The macro-benchmark was used to understand the impact of using our algorithms in a real application. We integrated MinPBFT and MinZyzzyva with the Java Network File System (JNFS) [37].

Unless where noted, we considered at most one faulty server ($f = 1$), requiring $n = 4$ servers for PBFT and $n = 3$ servers for MinPBFT and MinZyzzyva, and one client. The servers and clients were Dell Optiplex 745 2.8GHz Pentium 4 PCs with 2GB RAM and were connected over a Fast Ethernet at 100Mbps. The PCs had a Atmel TPM 1.2 chip. All experiments were done with Java's Just-In-Time (JIT) compiler enabled, and run a warm-up phase to transform the bytecodes into native code. All experiments run only in normal case operation (PBFT, MinPBFT) and gracious execution (MinZyzzyva).

8.1 Micro-Benchmark

The first part of the micro-benchmark was used to evaluate the performance of all 7 algorithms/versions: TPM-MinPBFT, TPM-MinZyzzyva, VM-MinPBFT, VM-MinZyzzyva, UP-MinPBFT, UP-MinZyzzyva and PBFT. We measured the latency of the algorithms using a simple service with no state, that executes null operations, with arguments and results with 0 bytes.

The latency was measured at the client by reading the local clock immediately before the request was sent, then immediately after the response was accepted[11], and subtracting the former from the latter. Each request was executed synchronously, i.e., it waited for a reply before invoking a new operation. In the case of the TPM-based implementations, the requests were issued approximately once every 4 seconds to avoid the delay of waiting for the TPM monotonic counter to be incremented. The results were obtained by timing 10,000 requests in two executions. The values presented are averages of these two executions.

The measurements are presented in Figure 5. The bars for the TPM-based versions are broken due to the difference of time scale: the time taken by the TPM-based USIG service to run `createUI` is 797ms, almost all of which are taken by the TPM to increment the counter and produce an RSA signature. Counters implemented in software and signatures done with ESIGN take much less time (less than 2ms).

In the experiments, PBFT has shown the best performance of all algorithms/implementations, followed by the MinZyzzyva implementations and the MinPBFT versions, which were the worse. Zyzzyva is known to be faster than PBFT in most cases [23, 43], but Zyzzyva (like PBFT) uses only MACs, while MinZyzzyva uses signatures, so MinZyzzyva ends up being slower than PBFT. Notice that MinZyzzyva did only gracious executions in the experiments; the performance is expected to be become worse than MinPBFT if that is not the case. The VM-based versions had slightly higher latencies than the user-level versions, which was expected as the hypervisor creates some overhead and the communication delay between VMs is higher than the communication delay between user-level processes in the same operating system.

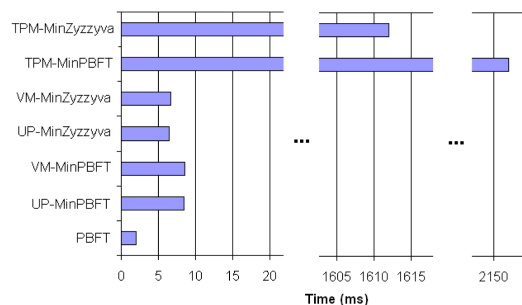


Figure 5: Micro-benchmark latency results for all versions (null operations, empty messages).

The second part of the micro-benchmark was used to evaluate the latency of the versions that do not use the TPM with different message sizes. The results are in Figure 6. The measurements were made like in the previous set of experiments, except that the arguments and results of the requests varied between 0 and 4000 bytes. The measurements show that the relation between the latencies of algorithms/versions is not affected by the size of the messages, and that the latency becomes higher with larger requests or responses, as expected.

The third part of the micro-benchmark had the objective of measuring the throughput of the algorithms. We ran experiments using requests and responses with 0 bytes. The client processes where evenly distributed over 2 client machines following the same test pattern used for PBFT [11]. The values reported in Figure 7 were obtained by measuring in the servers the number of client requests executed per second. The values are the mean of two independent executions with 10,000 requests. The throughputs of MinPBFT and MinZyzzyva are about 2 times lower than PBFT due to the overhead

[11] Involving receiving $f+1$ matching responses in PBFT and MinPBFT, and $2f+1$ in MinZyzzyva.

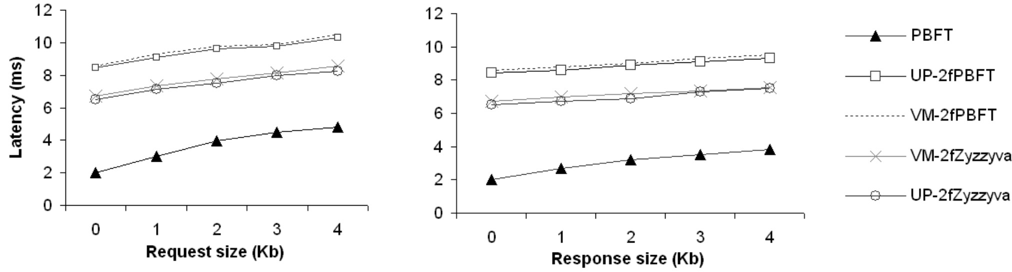


Figure 6: Micro-benchmark latency results varying request and response sizes (in Kbytes) for versions without TPM.

of obtaining the unique identifiers with digital signatures. All messages exchanged by these algorithms are signed using ESIGN. Signature creation and verification for 20 bytes of data (a SHA-1 hash) take on average 1.4ms and 0.88ms, respectively. Creation and verification of MACs in PBFT take on average 0.042ms and 0.035ms.

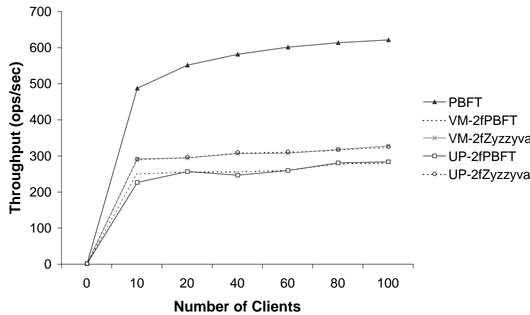


Figure 7: Throughput varying the number of clients (null operations, 0-byte requests/responses).

In the micro-benchmark the algorithms did not use batching. The throughputs would probably be closer of the ones already published (e.g., [43]) if this mechanism was used[12]. Nevertheless, our experiments provided results aligned to those obtained for A2M-PBFT-EA [12], which uses a trusted component, and also with DEPSPACE [7], which uses a BFT algorithm implemented in Java.

8.2 TPM Performance Considerations

As described in Section 7, there are two reasons for the poor performance of our TPM-based prototypes. In this section we discuss these reasons and how they can be overcomed in future TPM implementations.

The first reason is the maximum increment rate of the TPM monotonic counter (one increment by 3.5 seconds). Due to this reason the throughput of these versions is approximately one message by 3.5 seconds, as the bottleneck is the increment of the TPM counter. However, the use of the batching mechanisms would make a huge improvement of this throughput, as all pending requests would be assigned a sequence number (*UI*) every 3.5 seconds (which is still bad).

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"* [35]. 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 52ms still attaining this 7-year objective[13]. This would allow much better throughput and latency. Furthermore, if in a future TPM version the counter size is increased to 64 bits, it becomes possible to increment a counter every 12 picoseconds, which will make this limitation disappear.

The second reason for the poor perfomance we observed is the time the TPM takes to do a signature (approximately 700ms). 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. Currently there are already some applications [34], but their use seems to be far from being much spread. 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]. Other have also been pushing for faster TPM cryptography [30].

If these changes are implemented, the TPM performance will be improved and its functionality can be used to implement minimal BFT algorithms.

[12] We do not provide a study of the impact of the batching mechanism as a few have already been made [11, 43].

[13] 7 years divided by $2^{32} = 51.4$ms.

8.3 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 [37]. We compare the latencies obtained with a single server running plain JNFS and with three different replication scenarios: JNFS integrated with our PBFT implementation, JNFS with VM-MinPBFT and JNFS with VM-MinZyzzyva.

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 2 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 MinPBFT 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 (from 10% to 50%).

Phase	JNFS	PBFT	VM-MinPBFT	VM-2fZyzzyva
1	26	29	34	32
2	681	686	699	689
3	20	23	30	26
4	5	7	14	11
5	108	111	118	114
Total	840	856	895	872

Table 2: 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. [33] and Fraga and Powell [18]. However, the concept started raising more interest much later with works such as Rampart [38] and PBFT [11].

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 [45].

The idea of exploring a *hybrid fault model* in the context of intrusion tolerance or Byzantine fault tolerance, was first explored in the MAFTIA project with the TTCB work [15, 13]. 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 [47].

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 pattern, but made destination agreement, i.e., consensus on the order of execution [17]. Very recently Chun et al. presented another $2f+1$ BFT algorithm based on similar ideas, A2M-PBFT-EA [12]. This algorithm requires only local tamperproof components (to implement the A2M abstraction) and follows a Paxos-like pattern. MinPBFT and MinZyzzyva are also $2f+1$ BFT algorithms but that, on the contrary the previous two, are minimal in the several senses discussed above. They also follow a Paxos-like pattern.

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) [48]. 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 [25].

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 [29].

The main quest in BFT algorithms has been for speed. PBFT has shown that these algorithms "can be fast" [11] 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 [23]. An instruc-

tive comparison of these algorithms based on simulations was recently published [43].

Monotonic counters are a service of the TPM that appeared only in version 1.2 [35, 36]. 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 [46]. 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 [35]. Sarmenta et al. override this limitation by implementing virtual monotonic counters on an untrusted machine with a TPM [40]. 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 WANs. In contrast with the two previous $2f+1$ BFT algorithms, we were able to use the TPM as the trusted component due to the simplicity of our USIG service.

Acknowledgments

We thank Klaus Kursawe, Luis Sarmenta, Jean-Philippe Martin, Hans P. Reiser, Paulo Sousa, Eduardo Alchieri and Wagner Dantas for discussions about the algorithms, comments on the paper and help with several implementation aspects.

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 and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[12] 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.

[13] M. Correia, L. C. Lung, N. F. Neves, and P. Verissimo. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 2–11, Oct. 2002.

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

[23] 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.

[24] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] Trusted Computing Group. Enterprise security: Putting the TPM to work. 2006.

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

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

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

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

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

[48] 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.

[49] 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 CRM Correctness

CRM is correct if it satisfies the four properties stated in Section 4:

Validity: If a correct server multicasts a message m, then some correct server eventually accepts m.

Proof: If s_i is correct, then it multicasts m in a PREPARE message with a valid UI (lines 1-3). At least $f+1$ servers are correct, verify that UI is valid for message m and send f COMMIT messages with a valid UI to all the other servers (lines 4-7 and 8-11). This allows some correct server to accept m (lines 12-13).

Agreement: if a correct server accepts a message m, then all correct servers eventually accept m.

Proof: For a correct server to accept a message m, it must have received $f+1$ COMMIT messages (lines 12-13), at least one of which must have been sent by a correct server. Before a correct server accepts m, it multicasts a COMMIT message to all servers in lines 7 or 11. Therefore, all correct servers receive m either from the sender (i.e., the server that executed lines 1-3) or from another correct server and send a COMMIT message to all others. All correct servers accept m when they get COMMIT messages from $f+1$ correct servers, which is the minimum number of servers that are correct (lines 12-13).

Integrity: For any unique identifier *UI*, every correct servers accepts at most one message m with *UI*, and if *sender(m)* is correct then m was previously multicast by *sender(m)*.

Proof: This property derives trivially from the properties of the unique identifiers *UI* generated by the USIG service.

Confirmability: If a correct server accepts a message m, then it can prove that it accepted m.

Proof: A correct server accepts a message m in line 11 after receiving $f+1$ COMMIT messages with a valid *UI* (line 8). It can prove that it accepted m simply by presenting a CRM-certificate composed by these $f+1$ *UI* identifiers.

B MinPBFT Correctness

This section sketches proofs of the correctness of MinPBFT. We have to prove that the two *safety* properties are always satisfied (i.e., that all servers execute the same requests in the same order, the Agreement and Total Order properties) and the same for *liveness* (i.e., that it always makes progress). We divide our argument in normal case operation and view change.

B.1 Normal case operation

In normal case operation the primary remains always the same. For this case we need only to prove that the safety properties are satisfied, as liveness is guaranteed by doing a view change when the primary is suspected of being faulty.

The case in which all servers are correct is simple, since the primary uses the CRM algorithm that relies on `createUI` to define the sequence number of each request it receives, and the USIG service always provides sequential numbers in a server.

With a faulty primary, we have to prove that two different requests never get the same sequence number. Correct servers pick the sequence number of a request from the *UI* when they accept a message sent using the CRM algorithm. The Integrity property of CRM ensures precisely that two different messages never get the same *UI* identifier, thus also the same sequence number.

We also have to prove that a faulty primary can not cause the execution of the same request twice. Every server stores in a vector V_{req} the *seq* of the last request executed from each client. Therefore, if in normal case operation it detects that CRM accepted again a sequence number for a request already in V_{req}, it does not execute it.

B.2 View change

The main properties of the view change operation that we have to prove are: (1) no requests executed in v are executed in $v' > v$; (2) requests received but not executed in view v are executed in a view $v' > v$; and (3) the algorithm does progress. A view v' becomes the current view in a server when it receives a NEW-VIEW message with a valid *UI* and with $f+1$ VIEW-CHANGE messages with the same v'.

We first prove the first property, i.e., that no requests executed in v are executed in v'. First, however, it is important to show that this is a problem, not trivially solved with the *seq* number of the clients' requests. Suppose that the new view was just installed, that $n=3$, that s_0 was the primary in view v, that new primary for view $v+1$, s_1, is faulty and that the last two requests accepted/executed in view v were seq_1 and seq_2 (this one was the last). Consider also that server s_2 is slow and did not accept the request seq_2 yet because it did not receive all COMMIT messages yet (allowed by our lack of assumptions on communication delays); therefore, when s_2 multicasts the VIEW-CHANGE message it puts in C_{last} the request seq_1. Finally, suppose that the new primary also puts in C_{last} the request seq_1, not because it did not accept the request seq_2 – it did – but because it is faulty. The new primary can prove in message NEW-VIEW that request seq_2 was *not accepted* in view v, while in fact it was. Server s_2 does not execute request seq_2 in view v, breaking the safety properties of the algorithm. The *seq* number in clients' requests clearly does not help here.

This problem is avoided by using the FIFO order and V_{acc} to prevent processes from lying about the last request they accepted. The simplest way to explain this is with the previous example. If s_2 did not accept seq_2 then s_1 (new primary) must have accepted it or no server would have accepted it (there are no other $f+1$ servers). Therefore, s_1 has sent a COMMIT message with a *UI* identifier with a monotonic counter value, say, *cv*. When it sends the VIEW-CHANGE message, it has to send another *UI* with a monotonic counter value that must be

$cv+1$ (due to the properties of the USIG service). On the contrary, when it has sent the COMMIT message for request seq_1, the counter number must have been at most $cv-1$. Therefore, to prevent s_1 (or any other) from lying, a correct server only considers a VIEW-CHANGE message sent by s_i with a *UI* that contains a counter with value $cv+1$, being cv the value stored in V_{acc} in the position for server s_i.

Now we prove the second property, i.e., that requests received but not executed in view v are eventually executed in view $v' > v$. The proof resumes to noticing that a client resends a request that is not executed before a timeout. It the request is resent without need, it is not executed twice due to the *seq* mechanism. Nevertheless, when there is a view change the new primary tries to execute any requests that were not executed in the previous view, as long as it has received them.

The third property is about progress of the algorithm. The main property that must be guaranteed is that view changes do not go on indefinitely. A first cause for this effect might be one or more faulty servers sending REQ-VIEW-CHANGE messages, but $f+1$ servers are needed to force a view change, one of which must be correct. A second cause might be long delays in the network (or processors). To deal with this effect the timer used to detect if the primary is faulty starts with T_{vc} and goes on being multiplied by two whenever there is a view change. Therefore, as we assume that delays do not grow indefinitely (see Section 2), eventually a correct primary is selected and the delays are lower than the timeout used.

C MinZyzzyva Correctness

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 (Zyzzyva needs $2f+1$ but for MinZyzzyva $f+1$ are enough as the sequence number is defined by the USIG service). These certificates can be sent by a client in the *faulty server case* (see Section 6) or obtained from a set of $f+1$ matching checkpoints (see the same section). Notice that a COMMIT-certificate does not commit the execution of only one request, with a certain sequence number, but of all requests up to that sequence number. The reason is immediate: the certificate contains a confirmation (i.e., a reply with a UI) from $f+1$ servers, at least one of which must be correct and that does not execute a request without executing the previous ones.

The two properties that we have to prove about MinZyzzyva are the same that were proved for Zyzzyva [23]. 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.

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 $seq' \geq seq$ has a history $h_{seq'}$ that includes h_{seq} as a prefix.

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

Next we sketch a proof that MinZyzzyva satisfies these two properties.

C.1 Safety

Consider first the case in which there are only *gracious executions*. Clients send requests that are speculatively and sequentially executed by all $2f+1$ servers, which reply to the clients. Periodically the servers exchange checkpoint messages, which are used to compose COMMIT-certificates and pass part of the history from speculative to committed. Clearly any request that completes with a higher sequence number $seq' \geq seq$ has a history $h_{seq'}$ that includes h_{seq} as a prefix.

Now consider the case in which the client and the primary are correct but there are from 1 to *f faulty servers* that do not reply to the client (if they execute the requests or not is indifferent). A clients that sends a request, receives at least $f+1$ matching replies, but not $2f+1$, so it sends a COMMIT-certificate to all servers, passing part of their history from speculative to committed. As before, the history committed by the certificate includes the previously committed history as prefix.

A variant of this case is when the client is faulty and does not send the COMMIT-certificate. In that situation, the part of the history that would become confirmed if the COMMIT-certificate was sent, remains speculative until there is a checkpoint, something that will eventually happen.

The case in which there is a *view change* is easily proved considering that this operation is similar to MinPBFT's. For that algorithm we proved that the view change operation defines which was the last sequence number for the previous view and the first sequence number of the new view. Therefore, the process of committing parts of a history is exactly as before. If, for instance,

a client commits the execution of a request with sequence number *seq* in view v, it also commits all requests in previous views as all correct servers agree in which were the requests executed in previous views.

It is interesting to clarify why MinZyzzyva is indeed speculative, similarly to Zyzzyva. The reason is that some correct servers can (speculatively) execute requests in view v that are not considered to have been executed when there is a view change. For example, if the primary is faulty and sends the request with the sequence number to a backup b but not to the others, b speculatively executes the request but this request will not be considered to having been executed when there is a view change, and b will have to rollback the execution. Notice that however the behavior of the faulty primary is much more constrained that in Zyzzyva as the USIG will prevent any other request from being executed in the same view, causing a view change.

C.2 Liveness

The liveness property states that a request issued by a correct client eventually completes. We consider the same cases of the previous section. The case of a gracious execution is the simplest as all servers acknowledge the execution of the request and it becomes immediately complete.

The case of 1 to f faulty servers (other than the primary) requires that the client sends a COMMIT-certificate, which it does as it is correct. This certificate is then used by the servers to commit the history up to that request, and the correct servers send another reply making the request complete.

In the case the client is faulty, it may not send a COMMIT-certificate but, as discussed above, the execution of that request will eventually become committed by a checkpoint.

The last case is when the primary is faulty and tries to prevent progress in some way. That problem is solved by the view change operation, exactly as in MinPBFT.