BFT-TO: Intrusion Tolerance with Less Replicas

MIGUEL CORREIA[1], NUNO FERREIRA NEVES[2], PAULO VERISSIMO[2]

[1]*INESC-ID, Instituto Superior Técnico, Technical University of Lisbon - Lisboa, Portugal*
[2]*Universidade de Lisboa, Faculdade de Ciências, LASIGE - Lisboa, Portugal*
Email: miguel.p.correia@ist.utl.pt, nuno@di.fc.ul.pt, pjv@di.fc.ul.pt

State machine replication (SMR) is a generic technique for implementing fault-tolerant distributed services by replicating them in sets of servers. There have been several proposals for using SMR to tolerate arbitrary or Byzantine faults, including intrusions. However, most of these systems can tolerate at most f faulty servers out of a total of $3f + 1$. We show that it is possible to implement a Byzantine state machine replication algorithm with only $2f + 1$ replicas by extending the system with a simple trusted distributed component. Several performance metrics show that our algorithm, BFT-TO, fares well in comparison with others in the literature. Furthermore, BFT-TO is not vulnerable to some recently-presented performance attacks that affect alternative approaches.

Keywords: State Machine Replication; Byzantine Fault Tolerance; Intrusion Tolerance; Replication

Received 00 January 2000; revised 00 Month 2000

1. INTRODUCTION

This work was first developed towards the end of the Malicious and Accidental Fault Tolerance (MAFTIA) project, led by Professor Brian Randell. Brian's work influenced our own in many ways. We were fortunate to have Brian as a tireless participant in the discussions within the project about the notions underlying the concept of Intrusion Tolerance, which were very important for this work. Those discussions also contributed, for example, to the refinement of the dependability concepts and terminology in a paper authored by himself and colleagues [1]. The goal and direction of MAFTIA were themselves influenced by his vision, from at least two decades back, that the difficulty of constructing a secure operating system might be overcome through distribution [2]. That vision was well ahead of its time, since years later people were still writing that distributed systems impoverished security. One legacy of MAFTIA, to which Brian contributed, explained why this controversy lasted so long: under the AVI (attack, vulnerability, intrusion) composite fault model [3], it became clear that distribution may worsen security by increasing the impact of attacks; however, this may in turn be largely compensated for by a reduction of the impact of vulnerabilities brought in by distribution. As shown in numerous works, distribution is a powerful enabler of modular fault and intrusion tolerance mechanisms (actually, the main subject of this paper). In those early times, Brian had already understood the importance of concepts like 'security kernel' and 'reference monitor' for intrusion prevention, and again contributed to the conceptual evolution that led to using them for intrusion tolerance, as we do in this paper. Last but not least, Brian left an indelible mark with his seminal work in software fault tolerance and the idea that a program should be structured in a way that allows it to tolerate faults [4]. This cultural legacy made us for example be attentive to an aspect often disregarded: a Byzantine fault tolerant (BFT) system is of little utility in a security context if its software is not resilient to purposely made and common-mode faults (the makings of malicious faults). In fact, we recognize the need for diversity in this paper, and pursue the matter much further in other papers [5, 6].

Replication is a well-known technique to obtain fault tolerance and availability in distributed systems [7, 8, 9]. Currently it is often used in large datacenters and/or cloud providers in services such as BigTable [10], Dynamo [11], GFS [12], HDFS [13], MapReduce [14], and ZooKeeper [15]. A decade ago, a considerable interest appeared in using replication for *intrusion tolerance*, i.e., for ensuring that a service remains operational even if some of its replicas suffer intrusions and deviate arbitrarily from their correct behaviour [16, 17, 18]. These intrusion-tolerant or *Byzantine fault-tolerant* replication algorithms and systems have been argued to be useful to protect critical information infrastructures [19] and cloud services in federations of clouds [20, 21]. Like the original algorithms that coined the name [22], they continue to be useful to tolerate accidental arbitrary faults, such as data corruption in memory and disks due to hardware faults [23].

State machine replication is a generic solution to obtain fault- and intrusion-tolerant distributed services [9]. A service is implemented by a set of server replicas in such a way that it continues to behave as specified even if a number of servers is faulty. If the service is designed to tolerate arbitrary – also called Byzantine – faults, which model

accidental faults, attacks and intrusions, then the service is said to be intrusion-tolerant.

This paper describes an algorithm for implementing Byzantine fault-tolerant (or intrusion-tolerant) state machine replication (BFT-SMR) in practical distributed systems. These systems are characterized, among other aspects, by the uncertainty they show in terms of communication and processing delays. In such conditions, BFT-SMR requires at least $3f+1$ replicas to tolerate f being faulty [24], e.g., four replicas to tolerate one being faulty, or seven to tolerate two.

We present *BFT-TO*, an algorithm for asynchronous BFT-SMR with only $2f + 1$ replicas. The algorithm leverages a trusted component to cut f on the number of $3f + 1$ or more replicas required by many algorithms found in the literature [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]. When this algorithm was first published [35], it was the first to bring the lower bound on replicas for asynchronous BFT-SMR from $3f + 1$ to $2f + 1$. In the meantime, a few other algorithms appeared which also achieve that result: A2M-BFT-EA [36], MinBFT [37], MinZyzzyva [37], and EBAWA [38].

This result has practical as well as theoretical impact. In fact, SMR tolerating just crash faults needs only $2f + 1$ replicas [9], instead of $3f + 1$: the additional number of f replicas is the cost of arbitrary and/or malicious behaviour, and has a considerable economical impact, since each replica involves additional hardware, software and management complexity. It is also worth noting that Byzantine fault tolerance based on replication is only effective if intrusions in replicas occur independently, which in practice has to be enforced using some form of diversity [39, 40, 6]. This involves, for example, using different code or images, running on distinct operating systems, and thus, the higher the number of (diverse) replicas, the higher the cost in terms of management. In the light of this argument, cutting the needed resources by a percentage between 25%-33%, depending on the actual number of tolerated faults f, is an interesting practical result of our work.

In addition to the reduction of the number of replicas, our algorithm has a second benefit. Amir et al. have shown that BFT algorithms that rely on a coordinator to order messages are vulnerable to performance attacks [28]. BFT-TO belongs to a class of algorithms that avoid this problem by not relying on a coordinator - instead, ordering is performed in a distributed fashion.

Architectural hybridization [41] is the key to the reduction of the number of replicas. The baseline (and untrusted) system – a set of hosts interconnected by some network, subject to Byzantine failures – is extended with a simple component, on which we place a higher amount of trust. In this paper, this component is distributed and (the only one) assumed to be tamperproof. The trusted component (nicknamed *wormhole* [41]) executes a simple repertoire of functions, pretty much like a TPM would do [42]. Like the TPM, a wormhole can be implemented in hardware, increasing its shielding [43]. Unlike a TPM, the wormhole participates continuously, albeit at selected instants, in the execution of the Byzantine fault-tolerant algorithms.

This paper presents a novel wormhole called *Trusted Ordering Wormhole (TO wormhole)*. This wormhole provides a low-level ordering service that is the main building block of the BFT-SMR solution presented in the paper. The name of the algorithm, BFT-TO, comes from the fact that our Byzantine fault-tolerant (BFT) replication service is based on a trusted ordering (TO) service.

The paper provides the following main contributions:

- it presents BFT-TO, the first algorithm to bring the lower bound on the necessary number of replicas to perform asynchronous Byzantine fault-tolerant state machine replication from $3f + 1$ to $2f + 1$, more rigorously and in more detail than in the original publication [35];
- it presents the wormhole algorithm;
- it presents an analytical evaluation of the algorithm and a comparison with others in the literature;
- it presents correctness proofs of the algorithm.

The paper is organized as follows. Section 2 presents the system model and the TO wormhole. Section 3 presents the BFT-TO algorithm. Section 4 evaluates the performance of the algorithm analytically. Section 5 reviews related work and Section 6 concludes the paper. Finally, Appendices A and B present respectively the TO wormhole internal algorithm and correctness proofs of BFT-TO.

2. SYSTEM MODEL AND WORMHOLE

This section presents the distributed system model. As a precondition for being able to formally state global properties about the whole system, in our proposal the design and validation of a system extended with trusted components is backed by a *hybrid* distributed systems model [41], which recognizes useful things like: (i) the baseline system– let us call it *payload*– and the trusted component– wormhole– may have different properties and may rely on different sets of assumptions (w.r.t. faults, synchronism, etc.); (ii) there is a well-defined interface between both, where the properties of the trusted component provably emerge to the payload system components. The designer of hybrid distributed systems and their algorithms is thus obliged to provide evidence that despite the presence of threats (faults and attacks): the payload system and the trusted component each provide their services or functions correctly; the trusted component exchanges information with the payload system in a way useful to the global system. For example: synchronous wormhole vs. asynchronous payload; or secure wormhole vs. untrusted, Byzantine payload.

2.1. System Model

The system is in essence composed of a set of hosts interconnected by a network that we call payload network (Figure 1). We model the system components as automata. An automaton receives input actions and generates output and internal actions. A system is represented by a composition of automata.

The BFT-TO algorithm is executed by a set of *servers* $S = \{s_1, s_2, ..., s_n\}$ (modeled as automata with the same

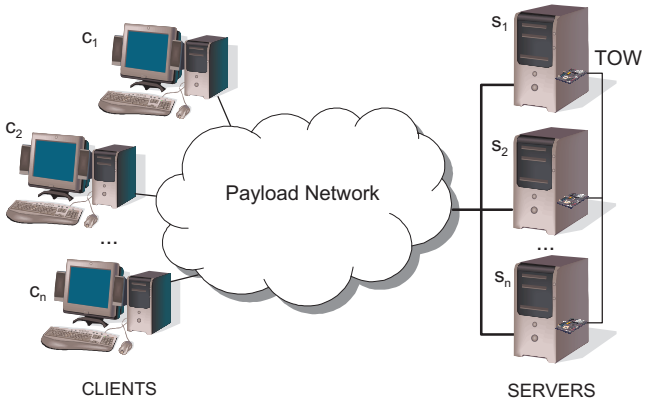

FIGURE 1. Architecture of the system.

identifiers). The service can be invoked by a set of *clients* (automata) $C = \{c_1, c_2, ..., c_m\}$ (m has to be finite but does not have to be known). We use the word *processes* to denote both servers and clients, so the set of processes is defined as $P = S \cup C$.

This environment is *asynchronous*, i.e., there are no assumptions about processing delays or message delivery delays [44]. The hosts have clocks but there are no assumptions about either local clock drift rates or the reliability of the readings they provide. The servers and clients are interconnected by a fully-connected network, although their communication can be delayed arbitrarily, e.g., as a consequence of an attack. Processes may behave in a *Byzantine* manner, that is, exhibit arbitrary faults.

2.1.1. *TO wormhole*

The asynchronous and Byzantine environment is extended with a partially-synchronous and tamperproof wormhole called Trusted Ordering Wormhole (TO wormhole), a distributed component with local parts in some of the hosts (local TO wormholes) and its own communication channel (TO wormhole control channel). The design is based on two assumptions about the TO wormhole:

1. the TO wormhole is tamperproof, in the sense that the integrity of the service it provides and the confidentiality of keys stored therein are preserved even if it is attacked;
2. the TO wormhole has enough synchrony for consensus to be solvable.

Assumption 1 does not say that the TO wormhole is fault-free, only that it has to be constructed in such a way that intrusions are not possible, i.e., that it is tamperproof. Local TO wormholes can still crash and the control channel can have periods of unavailability. Assumption 2 means partial-synchrony, implying that it is possible to circumvent the FLP impossibility result [44] and solve consensus inside the TO wormhole. As such, we consider that consensus inside the TO wormhole is a done deal, and in Appendix A, we present an internal TO wormhole protocol using a consensus protocol as building block.

Every host with a server needs a local TO wormhole, but not the hosts with clients (see Figure 1). Therefore the TO wormhole is formed by n local TO wormholes (automata) $T = \{t_1, t_2, ...t_n\}$, where local TO wormhole t_i is in the host of server s_i.

2.1.2. *Failure model in detail*

A process (client, server) is said to be *correct* if it follows the protocol it is supposed to execute. We assume that any number of clients can fail, but the number of servers that can fail is limited to $f = \lfloor \frac{n-1}{2} \rfloor$ (this is a strong assumption for long running systems, so in these systems mechanisms like proactive recovery should be used to clean compromised replicas [24, 45]). Failures may be Byzantine (or arbitrary), meaning that the processes can for instance simply stop, omit messages, send incorrect or forged messages, send several messages with the same identifier, and do so in an inconsistent manner across recipients. Faulty processes can pursue their goal of breaking the properties of the protocol alone or in collusion with other faulty processes. A process is also considered to be faulty if one of the secret keys discussed below is disclosed, or if it is not able to communicate with the local TO wormhole (e.g., because its local TO wormhole has crashed).

Direct communication between clients and servers, and among servers, is done exclusively through the payload network. A message m is sent to a channel when a process generates an output action `send(m)` and is said to be received when there is an input action `receive(m)` in a process. Servers may exchange data indirectly via the wormhole.

We assume that each client-server pair $\{c_i, s_j\}$ and each pair of servers $\{s_i, s_j\}$ is connected by a *reliable authenticated FIFO channel* that authenticates messages (prevents impersonation of the sender), ensures their integrity (detects and discards messages that are modified), and guarantees their eventual delivery in the order they were sent and without duplicates. In practice, these properties will be obtained with retransmissions and using cryptography (unless the channels are physically dedicated and isolated). Message authentication codes (MACs) are cryptographic checksums that can serve this purpose [46]. Processes have to share symmetric keys in order to use MACs. In the paper we assume these keys are distributed before the protocol is executed, e.g., using Kerberos. A solution to implement these channels would be to use TCP over IPsec [47].

Wrapping up, the payload system is *asynchronous Byzantine*: there are no bounds on the processing and communication delays; and the processes can fail arbitrarily. This system is extended with the TO wormhole, which forms a minimal part of the global system that is partially-synchronous and secure, providing a "well-behaved" service that the processes can use to perform some steps of their protocols.

2.1.3. Cryptography

We assume there is a *collision-resistant hash function* $Hash(x)$ that maps an input x into a fixed length output Hash(x) ("the hash of x"), for which it is computationally infeasible to find two different inputs that hash to the same output (often called strong collision resistance [46]). An example hash function currently believed to satisfy that property is SHA-256 [48]. We also assume the existence of a MAC function $Mac(x, k_{s_i c_j})$, where $k_{s_i c_j}$ is the key shared between server s_i and client c_j. This function provides essentially the same properties as the hash function but it can only be calculated by processes that have access to the key. Therefore, it can be used to protect the integrity and ensure the authenticity of x.

2.2. TO Wormhole's Service

From the point of view of the system executed in the payload network, the TO wormhole provides runtime support, similarly to a middleware or a library. The TO wormhole offers a single service that has the specific purpose of assisting the execution of the *atomic multicast protocol*, which is at the core of the BFT-TO algorithm.

The interaction between a server automaton and a local TO wormhole automaton is done by actions but, for simplicity, we say that a server *invokes* a function in a local TO wormhole API to mean that it generates an output action – say `TOW_register` – that corresponds to an input action with the same name in the local TO wormhole. Likewise, the TO wormhole must respond with some output action that corresponds to an input action in the same server. Therefore, these functions are a way to simplify the description of pairs of actions that are always executed together.

Before being able to use the TO wormhole's service, a server invokes function `TOW_register` to register and obtain an identifier *id* from the TO wormhole (Table 1, line 1). This function can return an error code but we omit it from the table for simplicity (notice that Table 2 provides some of the error codes for functions `TOW_sent` and `TOW_received`). By convention, the identifier that `TOW_register` returns to a server s_i is designated by s_i.

The TO wormhole assists the execution of the atomic multicast protocol by indicating two things for each message m atomically multicast: (1) *when* m can be delivered; and (2) the *order* in which m has to be delivered. The rationale is quite straightforward: when a server atomically multicasts a message or receives a message from the network, it indicates that event to the TO wormhole (see Figure 2). When $f + 1$ servers do that, the TO wormhole can be assured that at least one correct server has the message, because at most f servers can be faulty. If one correct server has it, then the server will send it to all the others. Therefore, the TO wormhole: (1) indicates that the message can be delivered and (2) associates an order number to it. Notice that the TO wormhole service *does not* implement the atomic multicast protocol. The messages atomically multicast are transmitted through the payload network, *not* through the TO wormhole control channel. This is worthwhile noting, to emphasize the asynchronous and Byzantine nature of the system as a whole, in which the TO wormhole is a mere trusted component, supplying simple services, provided through limited processing and communication capacities when compared to the rest of the system.

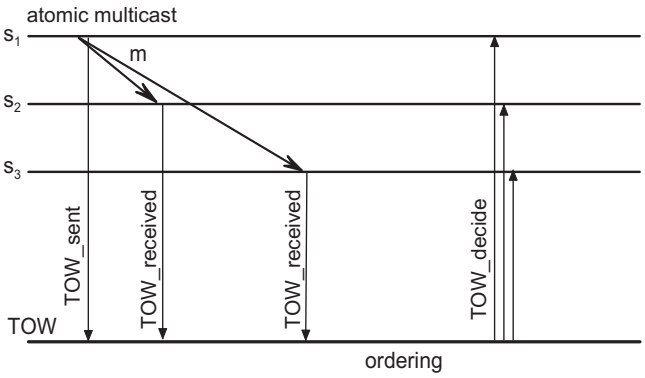

FIGURE 2. Time diagram of an example execution of the atomic multicast using the TO wormhole service.

When a server multicasts a message, it indicates that event to the TO wormhole by invoking the function `TOW_sent` (Table 1, line 2). The parameters of the function have the following meanings. The first, *id*, is the identifier of the server from the point of view of the TO wormhole (the id returned by function `TOW_register`). *members* is a set with the *ids* of the servers registered to implement the BFT-TO algorithm (has to be configured when the system is set up). *msg_id* is a message number that has to be unique for the server that sends it. *msg_hash* is a hash of the message. A server invokes `TOW_received` when a message is received (table, line 3). The parameters are the same as for `TOW_sent` plus the id of the sender, *sender_id*.

Suppose one server s_i invoked `TOW_sent` for a message and f other servers invoked `TOW_received` for the same message (with the same *msg_id*, *msg_hash* and *sender_id*$= s_i$). When that happens, the TO wormhole gives the next order number to the message and makes an output action `TOW_decide` (table, line 4). The parameters of this action are the identifier of the server that sent the message (*sender_id*), the message identifier (*msg_id*), the order number (*order_n*), the hash of the message (*hash*) and a set with the ids of the servers that have the message or, more precisely, that invoked either `TOW_sent` or `TOW_received` for the message (*has_msg*)[1]. `TOW_decide` will give the same order number to all servers because the TO wormhole is tamperproof.

Notice that the *members* parameter of functions `TOW_sent` and `TOW_received` plays the important role of specifying the set of servers implementing a replicated service using SMR. Although we consider only one service in the paper, it would be possible to have several, with different *members* sets. The *members* set is also used

[1] In practice, this set is implemented more efficiently by a mask with bits set to 1 or 0 depending on whether the corresponding server invoked `TOW_sent`/`TOW_received` or not.

	Dir.	Function
1	in	*id* ← `TOW_register`()
2	in	`TOW_sent`(*id*, *members*, *msg_id*, *msg_hash*)
3	in	`TOW_received`(*id*, *members*, *msg_id*, *msg_hash*, *sender_id*)
4	out	`TOW_decide`(*sender_id*, *msg_id*, *order_n*, *hash*, *has_msg*)

TABLE 1. TO wormhole programming interface

Function `TOW_sent`		
1	NO_RESOURCES	the TO wormhole is processing too many multicasts and has no resources for another one
2	NOT_IN_MEMBERS	the sender's id is not in the *members* set
3	RECALL	the function was already called for the same message
Function `TOW_received`		
4	UNKNOWN	the local TO wormhole has no information about this message
5	INVALID_HASH	the hash provided is not the hash of the message or is out of the range of valid hashes
6	RECALL	the function was already called for the same message

TABLE 2. TO wormhole error codes

by the TO wormhole to obtain the values of the constants n and f: $n = |members|$ and $f = \lfloor \frac{n-1}{2} \rfloor$.

Functions `TOW_sent` and `TOW_received` are invoked in the local TO wormholes, which are interconnected by the TO wormhole control channel, so the service provided by the TO wormhole involves the execution of a distributed protocol. This protocol is presented in Appendix A.

The TO wormhole's service satisfies the following properties:

- *TO Wormhole Validity.* If a correct (i.e., non-crashed) local TO wormhole generates an action `TOW_decide`(*sender_id*, *msg_id*, *order_n*, *hash*, *has_msg*), then there was a call to `TOW_sent`(*sender_id*, *members*, *msg_id*, *msg_hash*) and at least $\lfloor \frac{n-1}{2} \rfloor$ calls to `TOW_received`(*, *members*, *msg_id*, *msg_hash*, *sender_id*) in different correct local TO wormholes. Moreover, if *order_n*$>$ 1, then the local TO wormhole has done an action `TOW_decide`(*, *, *order_n*-1, *, *).
- *TO Wormhole Agreement.* No two different local TO wormholes generate two actions `TOW_decide` with the same *order_n* but different *sender_id*, *msg_id* or *hash*.
- *TO Wormhole Termination.* If there is a call to `TOW_sent`(*sender_id*, *members*, *msg_id*, *msg_hash*) and $\lfloor \frac{n-1}{2} \rfloor$ calls to `TOW_received`(*, *members*, *msg_id*, *msg_hash*, *sender_id*) in different correct local TO wormholes, then eventually every correct local TO wormhole does an action `TOW_decide`(*sender_id*, *msg_id*, *order_n*, *hash*, *has_msg*).

2.3. TO Wormhole Implementation

The approach presented in the paper requires the implementation of a TO wormhole satisfying assumptions 1 and 2 above. It is out of the scope of this paper to describe in detail the implementation of a TO wormhole, so we present only a brief discussion with the objective of helping the reader grasp how it might be done. There are several possible solutions experimented by us, which were presented and discussed in another paper, for a different wormhole, the TTCB [49].

Assumption 1 boils down to preventing intrusions in local TO wormholes and in the TO wormhole control channel. The state of the art does not allow 100% security to be guaranteed in arbitrarily complex systems. However, it is accepted that reasonably simple, static and small systems are verifiable in order to obtain a very high degree of assurance. For example, this is the reasoning behind the TCG Trusted Platform Module (TPM) [42] or the IBM cryptographic modules [50]. The TO wormhole is simple enough to fall within this concept.

A local TO wormhole prototype can be deployed in a straightforward and repeatable manner, by implementing it in a hardware appliance, with its own CPU and memory, isolated from the rest of the host, and with a secure interface to it. Several commercial programmable appliances that could be used with this purpose are currently available, e.g., we have experimented with devices from the Lantronix XPort family [51]. An even more straightforward solution that would not require specific hardware would be to implement the TO wormhole as a virtual machine, for example, on top of an hypervisor like Xen [52]. The payload legacy operating system and applications would run as other VMs on the machine. Xen would ensure isolation between the wormhole and the remaining VMs. We also have prototypes of wormholes based on Xen. In both cases, the interface between the payload system and the local TO wormhole also has to be protected against malformed inputs (e.g., buffer overflows or format string attacks), using well-known techniques like white-listing valid inputs and checking input lengths (see, e.g., [53]). Proper concurrency control is also important to prevent the possibility of race conditions [53]. For the wormhole control channel, the integrity of the communication can be enforced using, for instance, IPsec [47].

3. THE BFT-TO ALGORITHM

A *state machine* is characterized by a set of *state variables*, which define the state of the machine, and a set of *commands* that modify the state variables [9]. Commands have to be atomic in the sense that they can not interfere with other commands. The *state machine approach* consists of replicating a state machine in n servers $s_i \in S$. The set of servers S implements the *service*. All correct servers follow the same history of states if four properties are satisfied:

- *SM1 Initial state.* All correct servers start in the same state.
- *SM2 Agreement.* All correct servers execute the same commands.
- *SM3 Total order.* All correct servers execute the commands in the same order.
- *SM4 Determinism.* The same command executed in the same initial state in two different correct servers generates the same final state.

Property SM1 states that each state variable has the same initial value in all servers, something that is usually simple to guarantee. Properties SM2 and SM3 demand that the (correct) servers agree both on the set of commands to execute and on the order in which to execute them. This is enforced by sending the commands to the servers using an atomic multicast protocol. Property SM4 is about the semantics of the commands at the application level, so in this paper we simply make the assumption that the commands are deterministic. In practice determinism can be enforced using software wrappers [54].

The BFT-TO algorithm works essentially in the following way: (1) a client sends a command to one of the servers; (2) the server sends the command to all servers using an atomic multicast protocol; (3) each server executes the command and sends a reply to the client; (4) the client waits for $f + 1$ identical replies from different servers; the result in these replies is the result of the issued command.

This is a very simplified description of the process, so let us first delve into the details of the clients, and later we describe the protocol executed by the servers.

3.1. Clients

A client c_i issues a command *cmd* to the service by sending a REQUEST message to one of the servers, s_j. The message is sent through the payload network (clients do not have access to the TO wormhole). The format of the message is given by a tuple ⟨REQUEST, *src*, *dst*, *num*, *cmd*, *vec*⟩, where REQUEST is the type of the message, *src* is the address of the client c_i (source), *dst* is the address of the server s_j (destination), *num* is the request number, *cmd* is the command to be executed (including its parameters) and *vec* a vector of MACs (see discussion below). The request number is returned in the reply so that the client can associate the reply with the request. The client has to start it with a sequence number of 1 and increase it for each new request since this number is used to guarantee that the request is executed only once[2].

If the client and the server are correct, the REQUEST message is eventually received by s_j, due to the properties of the reliable authenticated FIFO channels used to connect each client/server pair (Section 2.1). Then, if the server is correct it atomically multicasts the message to all servers in S, all correct servers execute the command and send a reply to the client. The format of the reply message is ⟨REPLY, *src*, *dst*, *num*, *res*⟩, where REPLY is the type of the message, *src* is the address of the server, *dst* is the address of the client, *num* is the request number, and *res* is the result of the executed command.

This simple protocol is vulnerable to some attacks, therefore some modifications are necessary. First, a server s_j can be malicious and forward the message only to a subset of S, or discard it altogether. To solve this problem, if c_i does not receive $f + 1$ replies from different servers after T_{resend} units of time according to its local clock, it assumes that s_j did not forward the request, so it multicasts the message to another f servers. If this happens, it sends the message to a total of $f + 1$ servers, therefore at least one must be correct, and the request will be atomically multicast.

A malicious server might attempt a second attack by modifying the message before multicasting it to the other servers. To tolerate this attack, the request message takes a vector of MACs *vec*. This vector takes a MAC per server, each obtained with the key shared between the client and that server ($vec[s_i]$ has the MAC calculated with the key shared between the client and server s_i). Therefore, each server can test the integrity of the message by checking if its MAC is valid, and discard the message otherwise. A malicious client might build a vector of MACs with a combination of valid and invalid MACs. This attack would be ineffective: if enough correct servers receive the message with the correct MAC the command will be executed by all of them, otherwise it will be discarded.

In general, there will be restrictions on the commands that each client can execute. For instance, if the commands are queries on a database, probably not all the clients are allowed to query all registers in the same way. This involves implementing an access control mechanism. There are several schemes available in the literature and this issue is orthogonal to the problem we are addressing in the paper, so we do not propose any particular scheme.

We do not present this protocol in a more formal way due to its simplicity.

3.2. Servers

The protocol The servers execute Algorithm 1. The protocol is short because its core is the Byzantine fault-tolerant atomic multicast protocol that is presented later. The algorithm has an initialization part (lines 1-3) and a set of tasks either reacting to events (like the message reception in line 4) or conditions (e.g., like in line 9). State variables are tagged with a subscript with the number of the server (e.g.,

[2] If a malicious client uses the same number for several requests, at most one is executed.

Last$_i$, line 2). The operator '.' (dot) extracts a field from a message, e.g., m.*cmd* gives the *cmd* field in message m. A star '*' in a tuple is a wildcard that means any value (line 19). The state variables are used concurrently in all tasks so we assume that the three tasks are executed atomically (i.e., there is mutual exclusion among them).

The first task simply stores REQUEST messages received from clients in a set *R_received*$_i$ (lines 4-8). A message is only stored if its command has not yet been executed.

The second task essentially does an atomic multicast of all the requests from a client stored in the set *R_received*$_i$ (lines 9-12). To avoid doing an atomic multicast whenever a request is received (and inserted in *R_received*$_i$), the algorithm uses the vector *Wait*$_i$ to wait for the delivery of the server's previous atomic multicast for that client before doing the next one. This implements a form of batching of several requests in a single atomic multicast, reducing the overhead of ordering the requests[3].

The third task is executed when a message is delivered by the atomic multicast protocol (lines 13-26). A correct server atomically multicasts messages with requests from a single client but a faulty server can mix requests from different clients, so the for loop executes requests from each client c in order. The order of clients has to be deterministic, i.e., has to be the same in all servers, since the requests have to be executed in total order (property SM3). The while loop executes the requests of each client. It starts with an attempt to execute the request number *Last*$_i$[k]+1; if there is no such request, it discards any other requests from c (this cannot happen if the client is correct, thus follows its protocol). If it does execute that request, then it attempts to execute *Last*$_i$[k]+2, if it succeeds it attempts to execute *Last*$_i$[k]+3, and so on until there are no more requests. If some request is discarded, the client may have to timeout and resend the request. This mechanism makes the requests be executed in the order in which they are issued by the client, to avoid having to store the numbers of all the requests already processed. When a request is executed, its result is sent to the client and any request from that client and with the same number is removed from *R_received*$_i$ (lines 17-19). If the message delivered by the atomic multicast protocol has been sent by this server, the entry in the vector *Wait*$_i$ corresponding to the client for which requests were executed is set to FALSE to indicate that the *R_received*$_i$ set can now be multicast, and all requests in *A_delivered* are removed from *R_received*$_i$, so the server will not atomically multicast them again (lines 22-25).

DoS attacks and buffer management Algorithm 1 stores the requests received from the clients in the *R_received*$_i$ set. A request stored in *R_received*$_i$ must be removed from that set, either when that request is executed (line 19) or when it is atomically multicast and delivered (lines 9-12 and 13/22-25). Nevertheless, the number of requests and the rate at which they are sent is unbounded, so the size of this set might also need to be unbounded, which can create difficulties because memory is finite. This is especially a problem if a client attempts to do a denial of service (DoS) attack against the SMR service.

If requests are received faster than the atomic multicast delivers and the server executes the commands then, when the set is full, the request is simply not removed from the communication channel and, ultimately, the client will have to store it temporally in a buffer (if TCP over IPsec is used, this is done by TCP's flow control mechanism). This solution may create a problem of starvation/fairness because whenever requests are removed from *R_received*$_i$, the space gained may be filled with requests from the same client. To solve this problem, a simple mechanism, like trying to read one message from each of the clients before trying to read another one from the same client, can be used.

There is another problem caused by limited memory: if a client is malicious or crashed and does not receive its replies, the buffer(s) of the send primitive that store(s) replies can become full. The solution is to use a large buffer and to discard old messages, like in [24].

Read optimization Castro and Liskov propose an optimization for commands that only read the state of the service, which can also be used in our algorithm [24]. The client sends the request to all servers, which reply immediately with the value supposed to be read. If the client receives $f+1$ identical replies it can be sure that one was sent by a correct server, so it can accept that reply. Otherwise, it reissues the command using the normal protocol. There is only one case in which there may not be $f + 1$ identical replies, which is when there are concurrent writes modifying the data being read. Otherwise, all correct servers return the same value.

Recovery from replica failures State machine replication systems have to consider the possibility of a replica recovering after a crash. This is important, for instance, for situations in which many servers in a datacenter are switched off due to a power failure. BFT-TO deals with such failures similarly to PBFT and in-memory databases, so we do not present the mechanism in detail, only briefly sketch it.

Requests have to be stored in a log in secondary storage before they are executed. Such a log tends to grow a lot so periodically each server creates a checkpoint that represents its state. After a checkpoint is taken and stored in secondary storage, the log up to that point is removed. Recovering from a crash basically involves retrieving the last checkpoint saved, exchanging its hash with other replicas to understand if it is corrupted or not, installing that checkpoint as the replica state, and executing the operations in the log to update it. When that operation finishes, the replica can start executing the algorithm again, possibly starting by asking other replicas for missed requests. If the checkpoint or the log are corrupted, a state transfer has to be requested from the other replicas. One of the other replicas actually transfers the state and the others send a hash to check that the transferred state is not corrupted (the sender can be faulty).

[3] A server s_i does not send *all* requests in *R_received*$_i$ in an atomic multicast, but only requests of a single client. This is required by the request validation mechanism described in Section 3.2.1, which has to discard an atomically multicast message if one of the MACs is invalid.

Algorithm 1 SMR protocol (server s_i).

```
1: R_received_i ←∅                                   {set with requests received}
2: ∀c ∈ C, Last_i[c] ←0                              {vector with the number of the last request of each client that was executed}
3: ∀c ∈ C, Wait_i[c] ←FALSE                          {is there an atomic multicast of requests from client c not yet delivered?}

4: when receive(m_req =⟨REQUEST, c, s_i, num, cmd, vec⟩) do
5:    if (num > Last_i[c]) then
6:       R_received_i ←R_received_i ∪ {m_req}
7:    end if
8: end when

9: when (∃m_req ∈ R_received_i : Wait_i[m_req.src] = FALSE) do
10:   Wait_i[m_req.src] ←TRUE
11:   atomic_mcast({m_req' ∈ R_received_i : m_req'.src = m_req.src})
12: end when

13: when atomic_dlv(A_delivered) from s do
14:   for all c ∈ {m_req.src : m_req ∈ A_delivered} do
15:      while ∃m_req ∈ A_delivered : m_req.num = Last_i[m_req.src]+1 do
16:         Last_i[m_req.src] ←Last_i[m_req.src]+1
17:         res_i ←execute(m_req.cmd)
18:         send ⟨REPLY, s_i, m_req.src, m_req.num, res_i⟩ to m_req.src
19:         R_received_i ←R_received_i \ {⟨REQUEST, m_req.src, *, m_req.num, *, *⟩}
20:      end while
21:   end for
22:   if (s_i = s) then
23:      Wait_i[m_req.src] = FALSE
24:      R_received_i ←R_received_i \ A_delivered
25:   end if
26: end when
```

3.2.1. Atomic multicast protocol

The core of the algorithm executed by the servers is the atomic multicast protocol, which guarantees two properties: all correct servers deliver the same messages in the same order; if the sender is correct, all servers deliver the message that was sent. A server is said to (atomically) multicast a message m if it invokes `atomic_mcast(m)`, and it is said to (atomically) deliver a message m if action `atomic_dlv(m)` is done in the server. The protocol is more formally defined in terms of four properties:

- *AM1 Validity.* If a correct server multicasts a valid message m, then some correct server eventually delivers m.
- *AM2 Agreement.* If a correct server delivers a message m, then all correct servers eventually deliver m.
- *AM3 Integrity.* For any identifier *ID*, every correct server delivers at most one message m with identifier *ID*, and if *sender(m)* is correct then m was previously multicast by *sender(m)* [4].
- *AM4 Total order.* If two correct servers deliver two messages m_1 and m_2 then both servers deliver the two messages in the same order.

This definition is similar to other definitions found in the literature [55]. However, property AM1 is modified in such a way that it does not guarantee that the message m is delivered if m is not *valid*. This notion takes into account that the proposed atomic multicast protocol is used in the context of the state machine replication protocol to multicast messages that are sets with requests transmitted by clients. A message $m = S_{req}$ is said to be *valid* iff each request message in S_{req} contains a vector filled with valid MACs, i.e., with MACs properly obtained using the key shared between the client and each of the servers. Albeit the objective is to deal with malicious servers, if the client itself is malicious and sends a message with some not properly calculated MACs, the message may also not be delivered by the atomic multicast protocol.

The protocol The atomic multicast protocol is Algorithm 2. This protocol uses the service provided by the TO wormhole, so understanding it involves a mental jump: there are n servers that execute Algorithm 2, $f = \lfloor \frac{n-1}{2} \rfloor$ of which can fail in a Byzantine way (i.e., arbitrarily); however, the service provided by the TO wormhole is always correct, since we assume that it cannot be tampered with. The algorithm is presented as multicasting and delivering a set S_{req} instead of a message m to emphasize that it is used in the state machine replication algorithm to atomically multicast sets of requests (lines 7 and 46). This option was made for clarity at cost of generality (just in terms of presentation).

The algorithm has an initialization part (lines 1-6) and a set of tasks, following the same conventions as Algorithm 1. It uses only one type of message: ⟨ACAST, *src*, *dst*, S_{req}, *msg_id*⟩, where ACAST is the message type, *src* the address of the sender server, *dst* the set of addresses of

[4] The term *sender(m)* gives the sender field of the header of m.

the destination servers, S_{req} the set of request messages atomically broadcast, and *msg_id* a message number unique for the sender server that has atomically multicast the message.

Lines 1-6 initialize several state variables, including three sets used to store messages in different stages of processing: $A\_received_i$, $A\_wait\_decide_i$ and $A\_deliver_i$.

The first task is executed when `atomic_mcast`(S_{req}) is invoked (line 7). The server starts by testing if the MAC that corresponds to itself (s_i) in the vector of MACs is valid (line 8). If it is not, the server simply discards the message. If the MAC is valid, the request set S_{req} is enveloped in an ACAST message and multicast to all servers except the sender (line 9). Then, the server informs the TO wormhole about this message by invoking `TOW_sent` (line 10) and puts the message in the set $A\_wait\_decide_i$, waiting until enough servers invoke `TOW_received` and the TO wormhole does an action `TOW_decide` (line 33).

The second task is executed when an ACAST message is received by a server (lines 15-21). It simply tests if the MAC corresponding to itself in the vector of MACs is valid, and in the affirmative case, it stores the message in $A\_received_i$. If the MAC is not valid the message may still be decided so it is stored in $A\_wait\_decide_i$.

The third task takes the messages that the second task puts in $A\_received_i$ and invokes `TOW_received` (lines 22-32). This task is a loop executed every T_{sleep} because whenever a server s_i invokes `TOW_received` and the *local* TO wormhole still does not know that `TOW_sent` was invoked for this message in the sender, say s_j, it returns UNKNOWN, so the s_i has to invoke `TOW_received` again later. If the local TO wormhole knows about the message but the hash is invalid (INVALID_HASH), then the message is corrupted, so it is discarded. Otherwise, the message is stored in $A\_wait\_decide_i$.

The fourth task is executed when the TO wormhole does an action `TOW_decide` for a message (lines 33-35). The information given by the TO wormhole is simply stored in $T\_decided_i$.

The fifth task (lines 36-48) is executed when a message was both sent or received and inserted in the $A\_wait\_decide_i$ set (lines 12, 19 or 28) *and* there was a `TOW_decided` action for that message (lines 33-35), meaning that the message can be delivered. This tasks does some housekeeping and inserts the message to be delivered in $A\_deliver_i$ (lines 37-39). Then, if the server is not the message sender, it resends the message to the servers that did not 'contribute' to the threshold, i.e., to the servers not in *has_msg* (lines 40-42). The rationale for resending the message is that a malicious sender can send the message only to a subset of the servers (or send it with valid MACs only to a subset of servers); therefore, these servers may not have the message. The set $A\_deliver_i$ keeps messages that already have an order number assigned by the TO wormhole, therefore they can be delivered. These messages are handled in lines 43-47. The algorithm keeps a number with the next message to be delivered, $next\_deliv_i$. If the next message to be delivered is stored in $A\_deliver_i$, then the task delivers it (lines 41-45). Otherwise, the message has to wait for its turn. Notice that a message is delivered even if it was initially received with an invalid MAC, otherwise the client could violate the agreement property by sending invalid MACs to some of the servers.

DoS attacks and buffer management The problem of DoS attacks generated by clients of the SMR service is solved by the BFT-TO algorithm, which does not let the $R\_received_i$ set overflow (see Section 3.2). The problem in the atomic multicast problem is DoS attacks carried out by the malicious servers. Such a server –say s_j– can try to overflow the TO wormhole with requests, but this issue is left for Appendix A. It can also send messages to one or more servers without calling `TOW_sent` (lines 9-10), thus causing the insertion of messages in the $A\_received_i$ of those servers that will never be removed, since the TO wormhole will always return UNKNOWN when `TOW_received` is called (lines 24-25). Such an attack might eventually fill $A\_received_i$ with messages from s_j, preventing correct servers from sending it messages. The solution is simply to have a quota of messages that can be stored in $A\_received_i$ per server. In such a way, a malicious server can only fill its quota of space in $A\_received_i$, not the full set, allowing the correct servers to work normally. A similar reasoning and solution can be applied to set $A\_wait\_decide_i$.

FLP impossibility result The consensus problem has been proven to be impossible to solve deterministically in asynchronous systems if a process is allowed to fail, even if only by crashing [44]. This FLP impossibility result also applies to the atomic multicast problem because it is equivalent to consensus in several system models [55, 56, 57], thus also to state machine replication. In our case, the system is not fully asynchronous: the payload is asynchronous but the TO wormhole contains enough synchrony to solve consensus, which is needed to implement the TO wormhole service (see Appendix A). Therefore, the impossibility result does not apply.

4. BFT-TO PERFORMANCE

The main advantage of the BFT-TO algorithm is its lower number of replicas, which was a novelty when the algorithm first appeared. A second advantage is the lack of a leader and the consequent prevention of certain attacks against performance. This section presents an argument that BFT-TO's performance is similar to that of other BFT-SMR algorithms in the literature, both in terms of *latency* (the delay of processing a request) and *throughput* (the number of requests that can be processed per unit of time). We compare our BFT-TO algorithm with a set of SMR services in the literature in terms of: *number of communication steps*, where a step involves the process/server sending a message to the others and waiting to receive messages from $n - f$ of them; *number of messages sent*; and *number of signatures* and *signature verifications*.

All these parameters have an impact in terms of

Algorithm 2 Atomic multicast protocol (server s_i).

1: $next\_acast_i \leftarrow 1$ {number of next ACAST message to send}
2: $next\_deliv_i \leftarrow 1$ {number of next request to atomically deliver}
3: $A\_received_i \leftarrow \emptyset$ {set with received ACASTs}
4: $A\_wait\_decide_i \leftarrow \emptyset$ {set with ACASTs waiting for other servers to call TOW_received}
5: $A\_deliver_i \leftarrow \emptyset$ {set with requests waiting for delivery}
6: $T\_decided_i \leftarrow \emptyset$ {set with info about TOW_decide events}

7: **when** atomic_mcast(S_{req}) is invoked **do** {sender}
8: **if** ($\forall m_{req}$ =⟨REQUEST, c, s_i, *num*, *cmd*, *vec*⟩ $\in S_{req}$, Mac(⟨REQUEST, c, s_i, *num*, *cmd*⟩, k_{s_ic}) = $vec[s_i]$) **then**
9: $\forall s \in S \backslash \{s_i\}$, send m_{acast} = ⟨ACAST, s_i, S, S_{req}, $next\_acast_i$⟩ to s
10: TOW_sent(s_i, S, $next\_acast_i$, $Hash(S_{req})$)
11: $next\_acast_i \leftarrow next\_acast_i + 1$
12: $A\_wait\_decide_i \leftarrow A\_wait\_decide_i \cup \{m_{acast}\}$
13: **end if**
14: **end when**

15: **when** receive(m_{acast}) **do** {recipient}
16: **if** ($\forall m_{req}$ = ⟨REQUEST, c, s_i, *num*, *cmd*, *vec*⟩ $\in m_{acast}.S_{req}$, Mac(⟨REQUEST, c, s_i, *num*, *cmd*⟩, k_{s_ic}) = $vec[s_i]$)) **then**
17: $A\_received_i \leftarrow A\_received_i \cup \{m_{acast}\}$
18: **else**
19: $A\_wait\_decide_i \leftarrow A\_wait\_decide_i \cup \{m_{acast}\}$
20: **end if**
21: **end when**

22: **when** T_{sleep} elapses **do**
23: **for all** $m_{acast} \in A\_received_i$ **do**
24: $err \leftarrow$ TOW_received(s_i, S, $m_{acast}.msg\_id$, $Hash(m_{acast}.S_{req})$, $m_{acast}.src$)
25: **if** $err \neq$ UNKNOWN **then**
26: $A\_received_i \leftarrow A\_received_i \setminus \{m_{acast}\}$
27: **if** $err \neq$ INVALID_HASH **then**
28: $A\_wait\_decide_i \leftarrow A\_wait\_decide_i \cup \{m_{acast}\}$
29: **end if**
30: **end if**
31: **end for**
32: **end when**

33: **when** TOW_decide(*sender_id*, *msg_id*, *order_n*, *hash*, *has_msg*) **do** {sender and recipient}
34: $T\_decided_i \leftarrow T\_decided_i \cup \{\langle sender\_id, msg\_id, order\_n, hash, has\_msg\rangle\}$
35: **end when**

36: **when** $\exists m_{acast} \in A\_wait\_decide_i$, $\exists t_{dec} \in T\_decided_i$: $m_{acast}.src = t_{dec}.sender\_id \wedge m_{acast}.msg\_id = t_{dec}.msg\_id \wedge Hash(m_{acast}.S_{req}) = t_{dec}.hash$ **do**
37: $A\_wait\_decide_i \leftarrow A\_wait\_decide_i \setminus \{m_{acast}\}$
38: $T\_decided_i \leftarrow T\_decided_i \setminus \{t_{dec}\}$
39: $A\_deliver_i \leftarrow A\_deliver_i \cup \{\langle m_{acast}.S_{req}, t_{dec}.order\_n\rangle\}$
40: **if** $m_{acast}.src \neq s_i$ **then** {if s_i is not the sender of the message}
41: $\forall s \in S \backslash \{s_i\} \setminus t_{dec}.has\_msg$, send m_{acast} to s
42: **end if**
43: **while** $\exists \langle S_{req}, order\_n\rangle \in A\_deliver_i : order\_n = next\_deliv_i$ **do** {messages waiting to be delivered}
44: $A\_deliver_i \leftarrow A\_deliver_i \setminus \{\langle S_{req}, order\_n\rangle\}$
45: $next\_deliv_i \leftarrow next\_deliv_i + 1$
46: atomic_dlv(S_{req})
47: **end while**
48: **end when**

performance. The *latency* is mostly affected by the number of communication steps and the number of signatures and verifications in the critical path of the run, i.e., between the moment the client sends the request and gets the reply. The parameter that has most impact depends on the system: typically signatures and verifications are slower than the

communication in LANs (without acceleration hardware), and the contrary is true on WANs. The *throughput* is mostly affected by the number of messages sent and the total number of signatures and verifications in the run.

The comparison with other algorithms is done in two cases: in *nice* runs in which there are no faulty processes and no leader suspicions (for algorithms in which there is a leader); and bad runs with faulty processes or leader suspicions due to long communication delays. We start with an evaluation of the performance of the TO wormhole service, which is necessary for assessing the performance of the BFT-TO algorithm.

4.1. Wormhole Performance

The performance of the TO wormhole obviously depends on its implementation, an aspect that we purposely leave open in the paper. However, in Appendix A we present a possible instantiation of the internal protocol of the wormhole, which we use to assess its performance in terms of the metrics above. Notice that this is only a possible instantiation of the internal algorithm of the TO wormhole, not necessarily the most efficient.

The TO wormhole algorithm in Appendix A uses two primitives: *reliable multicast* and *multi-valued consensus*. These primitives only have to tolerate crash faults because they are executed inside a tamperproof wormhole. There are several possible instantiations. A simple reliable multicast protocol works the following way [55]: when a local TO wormhole wants to multicast a message, it sends it to all other local TO wormholes; these local TO wormholes deliver the message and send it to all other local TO wormholes. This protocol delivers the message in one communication step without faults (although it runs in two steps), sends $n^2 - n$ messages, and uses no signatures.

For multi-valued consensus, we consider an efficient protocol based on a failure detector, Schiper's early consensus [58]. The protocol uses a rotating coordinator, which sends its estimate to all others. If the coordinator is not suspected, when a local TO wormhole receives the coordinator's estimate, it sends it to all others. If a local TO wormhole gets estimates from more than a half of the local TO wormholes, it sends a special DECIDE message to all. Therefore, in a fault-free run in which the first coordinator is not suspected, the protocol decides in 2 communication steps, sends $2(n^2 - n)$ messages, and uses no signatures. If the coordinator is suspected, there are extra rounds of message exchange.

The TO wormhole algorithm in Appendix A works basically as follows. When a process calls `TOW_sent` or `TOW_received`, the information passed in the call is sent to all other local TO wormholes using the reliable multicast primitive. When there has been a call to `TOW_sent` and f calls to `TOW_received`, the TO wormhole executes a consensus to decide on the message(s) to be delivered with the next order number(s). The consensus is needed since two or more messages may be being ordered concurrently by the TO wormhole. Consensus is always executed sequentially, never concurrently. If there is a consensus running and the servers request the ordering of other messages, the local TO wormholes wait for the termination of the consensus and then run a new one, not with the information about one of the pending messages but with information about *all* pending messages. This provides a form of batching that greatly reduces the number of executed consensuses when there are many requests to the SMR service (similarly to what was observed in [59] with an atomic multicast protocol). Therefore, with a "low" number of requests, the algorithm of the TO wormhole runs in 3 steps, and exchanges n^3+n^2-2n messages (without suspicions of the coordinator). If there is a high number of calls to the TO wormhole, the factor related to the execution of consensus is divided by all messages, so the number of messages sent inside the TO wormhole per payload message is lower. Furthermore, the messages do not take the full requests but only their hashes (see Algorithm 2).

4.2. Performance in Nice Runs

This section compares the performance of BFT-TO with other BFT-SMR algorithms in the literature: Rampart [25, 60], PBFT [24], HQ [27], Zyzzyva [23], BFT2F [61], A2M-BFT-EA [36], MinBFT [37], MinZyzzyva [37], and EBAWA [38]. This list includes some of the best known algorithms and all that require only $2f + 1$ replicas. We did not try to be exhaustive due to the long list of protocols available. We excluded upfront algorithms that have very different characteristics (e.g., Q/U [62] and quorum algorithms [63] that provide weaker consistency) and those for which an actual SMR algorithm is not provided (although being usable as part of an SMR algorithm).

The algorithms are evaluated in nice runs, in which there are no faulty clients/servers and the leader is not suspected. In the case of HQ we also do not consider write contention, which might require using PBFT instead of the light quorum based protocols for which the performance is provided. Finally, we do not consider read optimizations, which are possible with all algorithms [24, 27]. They typically involve 2 steps, $2n$ messages and no signatures.

The comparison is presented in Table 3. The metrics are divided between those that have more impact in the latency – communication steps (*ComSteps*), signatures in the critical path (*SignCP*) and verifications in the critical path (*VerifCP*) – and those that have more impact on the throughput – total number of messages sent (*MesgTot*), total numbers of signatures (*SignTot*), and total number of signature verifications (*VerifTot*).

In the case of the metrics related to the throughput (right hand side of the table), some values of Rampart, PBFT and BFT2F are presented as a sum ($\oplus$) of two terms. The first term is constant per request, while the second can be shared by several requests, so it has a low impact in the performance when there are many concurrent requests.

The table shows that in nice runs, if the TO wormhole is not a bottleneck, then BFT-TO has a performance similar to PBFT. The only parameter that is worse than PBFT is MesgTot inside the wormhole, but: (1) the messages inside

		LATENCY			THROUGHPUT		
	Algorithm	**ComSteps**	**SignCP**	**VerifCP**	**MesgTot**	**SignTot**	**VerifTot**
1	Rampart	8	3	$2(n-f)+n$	$4n \oplus 3(n-1)$	$n \oplus (n-1)$	$(n-f)n \oplus (n-f)(n-1)$
2	PBFT	5	0	0	$2n \oplus (n-1)(2n-1)$	0	0
3	HQ	4	0	0	$4n$	0	0
4	Zyzzyva	3	0	0	$2n$	0	0
5	Spinning	5	0	0	$2n \oplus (n-1)(2n-1)$	0	0
6	BFT2F	5	2	$2f$	$2n \oplus (n-1)(2n-1)$	$(n+1)$	$n(2f+1)$
7	A2M-BFT-EA	5	0	0	$n+1 \oplus (2n-1)n$	0	0
8	MinBFT	4	0	0	$2n \oplus (n-1)n-1$	0	0
9	MinZyzzyva	3	0	0	$3n-1$	0	0
10	EBAWA	4	0	0	$2n \oplus (n-1)n-1$	0	0
11	*BFT-TO*	5	0	0	$2n$ $[+(n^3+n^2-n)]$	0	0

TABLE 3. Comparison of BFT-SMR algorithms in nice runs

the wormhole do not take the whole request but only a hash (with, e.g., 32 bytes if obtained with SHA-256), so they tend to be smaller; and (2) the number of messages sent can trivially be reduced using piggybacking. Rampart is the oldest algorithm, so it is no wonder that it has the worse performance in most metrics. HQ and BFT2F can be quite efficient with cryptographic acceleration or if the network delays are high enough to compensate for the use of public-key cryptography [61]. Zyzzyva and MinZyzzyva are very efficient in nice runs, but are very sensitive to network delay variations and client misbehaviour.

4.3. Performance with Faults or Suspicions

The previous section compared the algorithms in nice runs, i.e., in runs in which there were no faulty servers and the delays were bounded, which prevented leader suspicions from occurring. For HQ we also considered that there was no contention. Here we remove these restrictions and analyze how the algorithms cope with faulty servers and leader suspicions, plus contention for HQ. A textual assessment of the effect of these conditions is shown in Table 4. The table only includes attacks against the protocol, not generic attacks like a malicious server flooding the network with messages to delay the service. We also do not assess the performance impact of malicious clients.

Most algorithms are affected by network delays or a faulty leader, which can cause one or more view changes and consequent delays. Zyzzyva and MinZyzzyva run two additional communication steps even if a single message exchanged between client and server is delayed (what they call a non-gracious execution). Our algorithm, in the payload, is immune to this effect as there is no leader. If there are long delays in the TO wormhole control channel, then consensus can also be delayed, but this effect is expected to be rare if this channel is a different network (like in the TTCB wormhole prototype).

Table 4 presents only a description of what is a bad run and its consequences. Table 5 instead, quantifies the additional cost in terms of latency of such a run in relation to a nice run. The column on the right indicates what is the cost being accounted in the columns on the left.

Most algorithms can run an arbitrary number of view changes, if the network experiences long delays, so it is not possible to assess what is the worst case, only the cost of each additional view change. If the network does not experience long delays but there are faulty servers, then the worst case is the execution of f new view changes (in case the leader is faulty and the following $f-1$ leaders are also faulty). The exception is our algorithm. We do not provide the metrics related to throughput because we expect the messages and signature operations done for view change to be not so relevant when compared with those of processing requests.

5. RELATED WORK

State machine replication The state machine approach was first introduced by Lamport for systems in which there are no faults [64]. Schneider evolved the idea for fault tolerance [9]. In the 1990s the first Byzantine-resilient state machine replication (BFT-SMR) libraries appeared. Reiter and colleagues introduced Rampart [25, 60] and later Castro and Liskov presented PBFT [24]. In both cases and in several systems that followed, $3f+1$ replicas are needed.

The reason for this bound is grounded in distributed computing theory. In practical distributed systems, like those that rely on the Internet for communication, there is a considerable uncertainty in terms of time, so they are often modeled as being asynchronous. This is the same as saying that these systems make no assumptions about bounds on communication and processing delays. State machine replication requires solving atomic broadcast (or total order broadcast), which is equivalent to consensus in several variations of the asynchronous model [55, 56, 57]. Byzantine fault-tolerant consensus has been shown to require $3f+1$ processes for being solved in several variations of these models [65, 66]. Therefore, it is natural to require $3f+1$ servers to solve BFT-SMR.

Rampart is an intrusion-tolerant group communication system. It provides a set of communication primitives and a membership service, which handles the joining and leaving of group members [25]. Replication is implemented by a set of servers that form a group. Clients send their requests to a server of their choice, similarly to our algorithm. The output of the service has to be voted so that the results provided by correct servers prevail over those returned by malicious servers. Two solutions were implemented: one

	Algorithm	Bad run	Consequence
1	Rampart	Long communication delays or faulty leader	One or more view changes
2	PBFT	Same as Rampart	Same as Rampart
3	HQ	Same as Rampart/PBFT if there is contention	Change to PBFT and run PBFT
4	Zyzzyva	Single longer delay or faulty server	More 2 comm. steps, view change
5	Spinning	Same as Rampart/PBFT	One or more merge operations
6	BFT2F	Same as Rampart/PBFT	Same as Rampart/PBFT
7	A2M-BFT-EA	Same as Rampart/PBFT	Same as Rampart/PBFT
8	MinBFT	Same as Rampart/PBFT	Same as Rampart/PBFT
9	MinZyzzyva	Same as Zyzzyva	Same as Zyzzyva
10	EBAWA	Same as Spinning	Same as Spinning
11	*BFT-TO*	Nothing (outside the wormhole)	Not affected (outside the wormhole)

TABLE 4. Comparison of what are bad runs and what are their consequences for the SMR services

	Algorithm	ComSteps	SignCP	VerifCP	Costs of
1	Rampart	6	2	$2(n-f)$	1 view change
2	PBFT	3	0	0	1 view change
3	HQ	3 to 5	0	0	1 change to PBFT + PBFT execution
4	Zyzzyva	3	0	0	non-gracious exec. and 1 view change
5	Spinning	2	0	0	1 merge operation
6	BFT2F	2	2	$2f+2$	1 view change
7	A2M-BFT-EA	3	0	0	1 view change
8	MinBFT	3	0	0	1 view change
9	MinZyzzyva	3	0	0	non-gracious exec. and 1 view change
10	EBAWA	2	0	0	1 merge operation
11	*BFT-TO*	—	—	—	

TABLE 5. Additional costs of bad runs

very similar to ours, and another one based on a (k, n)-threshold signature scheme, which has poorer performance.

Two additional intrusion-tolerant group communication systems are SecureRing [67] and SecureGroup [68]. They also require $3f + 1$ servers but there is no discussion about their use for state machine replication. SecureRing is based on ring protocols, which are very efficient in local networks. SecureGroup is based on a family of randomized atomic multicast protocols. We presented a fourth group communication system called Worm-IT [69]. That system is based on an earlier wormhole, the Trusted Timely Computing Base (TTCB) that provides several services, instead of the single service of the TO wormhole. The TTCB services do not allow to solve BFT-SMR with only $2f + 1$ replicas, so Worm-IT requires at least $3f + 1$ servers.

PBFT is a highly influential BFT-SMR library [24]. It was designed with performance in mind, so it minimizes the number of messages sent by having a leader that defines the sequence of execution of the requests and avoids using public-key cryptography. Experiments with PBFT have shown very good performance, leading many researchers to start to consider BFT-SMR practical for the first time. PBFT requires $3f + 1$ servers and it is not a group communication system, as the group of servers is fixed.

SINTRA provides a number of group communication primitives (reliable, causal, atomic multicast) that can be used to support SMR [26]. These primitives are based on a randomized Byzantine agreement protocol, therefore they are fully decentralized and strictly asynchronous. They also require $3f + 1$ processes.

HQ is an evolution of PBFT that uses quorum-based algorithms when there is no contention, but switches to PBFT when there is, thus ensuring liveness [27]. Another algorithm, Q/U, uses lighter, quorum-based algorithms, but does not ensure the termination of the requests in case there is contention [62].

Zyzzyva adds speculation to PBFT [23]. It executes client's requests before the backups actually agree with the execution order defined by the primary. If the server is faulty, something that is supposedly very rare, the service has to rollback these executions, which makes the programming model challenging. Zyzzyva achieves excellent performance in gracious executions, i.e., when there are neither faulty servers nor delays in the communication. Scrooge shows that BFT-SMR can perform even better with a few additional replicas [34]. Zzyzx also evolves Zyzzyva and extends it with locks, achieving near-linear throughput scaling [70]. UpRight is another BFT-SMR library, but it aims to be easy to adopt, instead of achieving better performance [30].

PBFT and other algorithms based on a leader may be vulnerable to attacks that delay the leader in order to make it suspicious and cause view changes [71]. The PABC atomic broadcast protocol forces the system to make progress by running a randomized agreement protocol that delivers any pending requests before changing the leader [71].

Lamport's Paxos algorithm trivially solves state machine replication if there are only crash faults [72]. With Byzantine faults, transforming Paxos in state machine replication

is not so simple. Zielinski and Martin et al. proposed Byzantine Paxos algorithms that terminate in a low number of communication steps in synchronous and fault-free executions [73, 74]. To implement SMR, these algorithms would have to be extended.

Amir et al. designed an hierarchical BFT-SMR system for WANs [75]. Servers in sites can be replicated, so they use a Byzantine Link protocol to connect sites and minimize the number of messages exchanged. The same authors presented Steward, which also does replication on WANs connecting possibly replicated servers in sites, but without providing BFT-SMR [76]. RAM is a BFT-SMR algorithm for WANs that uses a rotating leader and exploits the A2M abstraction also used in A2M-BFT-EA [77].

All these algorithms have in common that they require at least $3f + 1$ replicas.

BFT-SMR with less than $3f + 1$ *replicas* The need for $3f + 1$ servers to tolerate only f faulty is a well-known problem of BFT-SMR algorithms, so several works tried to break this lower bound.

Quorum systems are a way to reason about subsets of servers (quorums) from a group. Asynchronous quorums protocols can be used to implement read/write registers but not to make any service fault-tolerant like SMR. These registers can be implemented with only $2f + 1$ servers if the data is self-verifying, e.g., if it data objects are signed with a private key for which the corresponding public key is available to the readers [63].

Yin et al. presented a state machine replication scheme that separates agreement about the order in which the requests are to be run from the execution of the requests [78]. Agreement is carried out by a set of servers, and execution by another (not necessarily disjunct) set. The benefit is that agreement has to be done by $3f + 1$ servers, while request execution – the service itself – needs only $2f + 1$ servers. Nevertheless, the total number of servers is still at least $3f + 1$.

SPARE and ZZ go one step further by putting some execution replicas on hold until they are needed [79, 80]. This approach, which is especially useful for datacenters with many servers but where processing time should be spared, manages to use $3f + 1$ replicas for agreement and only $f + 1$ for execution when there are no faults. Distler and Kapitza use a similar idea to improve the performance of the BFT service [81].

Li and Mazieres proposed BFT2F, an algorithm that improves the resilience of PBFT in a different sense [61]. They consider $3f + 1$ and give the same guarantees as PBFT if f or less replicas are faulty. However, 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.

FS-NewTOP is a BFT-SMR system based on the notion of fail-signal (FS) processes, i.e., processes that announce when they fail [82]. Each FS process is implemented by two nodes connected by a synchronous channel. Each node monitors its peer. When one node detects that its peer has misbehaved in some way, it signals the failure to all processes and stops the FS process. The system needs $2f+1$ pairs of nodes to tolerate f faulty nodes in different pairs. However, it does not tolerate two faulty nodes if they are from the same pair.

The first algorithm that managed to implement BFT-SMR with only $2f + 1$ replicas was an earlier version of BFT-TO itself, first published in 2004 [35]. The second algorithm to do the same was A2M-BFT-EA [36]. This algorithm is based on tamperproof components that are local to each server, unlike the TO wormhole that is distributed (includes a control channel). These components provide an abstraction called Attested Append-Only Memory (A2M) that essentially implements a log in which all replicas agree. The abstraction provides functions to append, lookup and truncate values in the log, but no functions to replace values. This abstraction provides a means to avoid duplicity, i.e., the possibility of a faulty server sending inconsistent messages to different servers. The TO wormhole is more complicated as it actually assigns sequence numbers to hashes of requests (or batches of requests).

An even simpler wormhole is the Unique Sequential Identifier Generator (USIG) of Veronese et al. [38, 37]. It simply returns a signature of the concatenation of a counter and a message hash. It provides operations only to increment the counter and to verify if counter values are correctly signed. Three algorithms based on the USIG for BFT-SMR with $2f + 1$ replicas have been presented: MinBFT that is related to PBFT but with less replicas and one less communication step [37], MinZyzzyva that is inspired in Zyzzyva but with less replicas and one less communication step [37], and EBAWA that has improvements for WANs like rotating the primary (similarly to RAM and Spinning, which we discuss below) [38]. A good example of the independence between concept and implementation of wormholes, is that the USIG wormhole described in the paper can be implemented either from scratch or in hardware by the Trusted Platform Module (TPM) chip [42], wrapped in a thin layer of software that calls the appropriate TPM functions and provides the wormhole interface to the payload system. The main services of the TPM used are a counter, the signature function, and sessions. Abraham et al. presented a randomized consensus algorithm that also requires only $2f + 1$ processes by resorting to a different feature of the TPM, the platform configuration registers [83]. The USIG service is similar to TRINC [84].

Performance attacks Amir et al. have shown that PBFT is vulnerable to two attacks that do not compromise its safety, but that can seriously impair its performance [28]. In a *pre-prepare delay attack*, a faulty server delays the ordering of most requests. This not only delays those requests but also greatly reduces the throughput of the service. In PBFT the backups monitor the behaviour of the leader to prevent it from stopping to order requests. The backups do this by imposing a maximum delay for the execution of requests. However, it is difficult to monitor the time of all requests so they only monitor the first request of a queue

of pending requests (similarly to what happens in go-back-n protocols for reliable communication). This allows a faulty primary to process one request at a time, strongly delaying most requests. A *timeout manipulation attack* consists in faulty servers increasing the timeouts used in PBFT, again seriously degrading the performance of the system. These attacks not only affect PBFT but also others that follow the same leader-based pattern, e.g., Zyzzyva, UpRight, A2M-PBFT-EA, MinBFT, and MinZyzzyva.

Prime is the algorithm presented by Amir et al. to prevent these attacks [28]. It adds a pre-ordering phase of three communication steps to PBFT. This phase forces the leader to follow a certain pace in the request ordering process.

Clement et al. also presented a system that aims to deal with these and similar attacks, Aardvark [29]. It is also a modification of PBFT that tolerates these attacks by monitoring the performance of the primary and changing the view in case it seems to be performing slowly.

Spinning deals with these attacks in a different fashion [32]. It is leader-based like PBFT, but it rotates the leader whenever a request (or a batch of requests) is ordered. A faulty leader can only delay requests in its turn, but if such a delay is detected, that replica is put in a blacklist and is no longer used as one of the rotating leaders. EBAWA rotates the leader similarly to Spinning, although it also uses the USIG and other mechanisms to achieve good performance in WANs [38].

As already mentioned, BFT-TO is not vulnerable to such attacks because it is symmetric, not leader-based. When a faulty server receives a request from a client it can delay it, but this does not impact the overall performance of the system. It is up to the client to detect that the server is slow and to choose another one to contact.

Other works Pedone et al. used weak ordering oracles to solve crash-tolerant agreement problems in asynchronous systems [85]. The oracle gives a hint about the order of the messages, which may be right or wrong. The hint is simply the order in which the messages are received from the network, which is often the same in all servers in a LAN. Our TO wormhole might be considered to be a perfect ordering oracle that can be used to solve Byzantine agreement problems.

6. CONCLUSION

State machine replication is a well-known generic technique for implementing fault- and intrusion-tolerant distributed services. This paper proposes a novel state machine replication algorithm, BFT-TO. The algorithm is executed in an asynchronous and Byzantine environment, with the assistance of an ordering service executed in a simple component, the Trusted Ordering Wormhole.

The BFT-TO algorithm, originally published in [35], introduced a new lower bound on the necessary number of replicas to perform asynchronous Byzantine fault-tolerant state machine replication. In essence, we managed to design an atomic multicast protocol that requires only $2f + 1$ servers, instead of the usual $3f + 1$ or more. The BFT-TO algorithm circumvents the FLP impossibility result without any synchrony assumptions on the payload part of the system; the partial synchrony necessary to circumvent FLP is in the wormhole. Interestingly, the Trusted Ordering service is modular, and since its design is presented and a correctness proof given, it is re-usable by future works based on hybrid distributed system models.

From a practical viewpoint, this theoretical result also allows an important reduction in the complexity and cost of fault and intrusion tolerant services based on state machine replication, by 25%-33%. Moreover, a comparison with similar replicated services in the literature shows that BFT-TO improves on other BFT-SMR protocols in terms of several metrics, by exhibiting better performance in the presence of attacks and faults, whilst maintaining similar performance in nice runs. Also, BFT-TO is not sensitive to attacks known to hamper the performance of some of the best known BFT-SMR algorithms, which are leader-based. The fact that BFT-TO follows a symmetric structure is key to this stable behaviour.

In conclusion, it is worthwhile observing that the use of architectural hybridization has been gaining increasing acceptance over the past few years. Reputed authors have published very interesting results relying on the use of hybrid architectures and/or trusted components of various grades, with special evidence to some works nearer our approach [36, 86] but also [77, 87, 88]. We have published further advances on BFT [37, 38], for example featuring even simpler wormholes and/or more performant algorithms.

It is perhaps important to note that these hybrid architectures are best designed and validated under a hybrid model, since a homogeneous model cannot formally exploit the powerful fact of the existence of components following different assumptions. In other words, we argue that it may not be enough, for example, to just use trusted or synchronous components in the midst of untrusted or asynchronous systems; one should also use an adequate model to compute with the resulting architecture. A hybrid distributed system model does not necessarily change the system, it changes the *way* in which we look at the system [41]. It provides a formal reasoning framework for systems and algorithms where components exist that follow different assumptions or exhibit different properties from the remainder of the system (e.g., failure detectors; privileged synchronous channels; watchdogs or timers; secure logging, storage, signing or attestation). Such systems, if looked upon under a homogeneous model (i.e., the classical model), appear to work, but may very well conceal unexpected behaviours or unsubstantiated assumptions [89], which may end-up being a cause of system incorrectness and/or failure.

APPENDIX A. TO WORMHOLE ALGORITHM

This section presents the algorithm that implements the TO wormhole service, thus giving more details about the implementation of the TO wormhole (see Section

2.3). This section presents the algorithm executed when a process (server) invokes TOW_sent and TOW_received, and the TO wormhole generates the action TOW_decide. Remember that each correct server calls TOW_sent whenever it multicasts an ACAST message with one or more requests, and TOW_received whenever it receives an ACAST message. When the TO wormhole reaches a decision that an ACAST message can be delivered with a certain ordering, each local TO wormhole generates an action TOW_decide.

This section requires the reader to make a mental leap: the algorithm presented here is executed inside a secure environment, so it only has to tolerate crash faults, no longer Byzantine faults.

Appendix A.1. TO wormhole's System Model Revisited

Recall that the distributed TO wormhole is formed by n local TO wormholes: $T = \{t_1, t_2, ...t_n\}$. At most $f = \lfloor \frac{n-1}{2} \rfloor$ local TO wormholes can fail, only by crashing. The TO wormhole control channel fully connects all local TO wormholes. Disconnections can happen but communication will eventually be possible.

We do not make additional statements about the communication and time models of the TO wormhole[5] because the algorithm is presented in terms of two primitives that we assume are available inside the TO wormhole: a reliable broadcast primitive and a consensus primitive. These primitives can be implemented using channels that may or not be reliable. The implementation of a crash-failure consensus primitive is straightforward, so it is outside the scope of this paper. It suffices to say that the TO wormhole environment has the minimal properties needed to implement it, namely the partial synchrony assumption to circumvent the FLP impossibility result (randomization would also do, in alternative).

The broadcast and consensus primitives are defined in the usual way. If a message m is sent using the reliable broadcast primitive (m_broadcast(m)), then it is delivered to all correct (i.e., not crashed) local TO wormholes (m_deliver(m)), including the sender. The consensus primitive ensures agreement on a single value v (c_decide(v)) among those proposed by the local TO wormholes (by invoking c_propose(v)). Formal definitions of these primitives can be found, e.g., in [55].

Appendix A.2. The Algorithm

The internal algorithm of the TO wormhole (TOW) is relatively simple (Algorithm 3). This simplicity is important in the case of an assurance process for the secure TO wormhole implementation, whose cost and degree of confidence would depend strongly on the complexity of the component.

The protocol is composed of a set of routines. When TOW_sent or TOW_received are called, some tests are performed and the information is broadcast to all local TO wormholes in SENT and RCVD messages, including the local TO wormhole where the function was called (lines 4-8 and 9-13). Then, when one of these SENT/RCVD messages is delivered to the TOW (m_deliver), it is simply stored in a set (lines 14-16).

Recall that ACAST messages are the messages used by the atomic multicast protocol (Section 3.2.1). When there are RCVD messages from at least half plus one of the local TOW about the same ACAST message (one at each server), a consensus decision is performed (lines 17-21). Function $\#_t(Set)$ counts the number of occurrences of tuple t in set Set. Since the messages are reliably broadcast, each local TO wormhole delivers the same set of messages and they all engage in the same decision process. Each local TO wormhole proposes the information about the ACAST messages for which it has half plus one RCVD messages (lines 19-20). Consensuses are executed in sequence, never concurrently (variable $agreeing_i$ enforces this). This generates batching: while a consensus is running, the local TO wormholes are being called with information about other ACASTs, this information is stored and when the consensus ends a new one is started that does agreement about several ACASTs, instead of only one. Therefore, a single consensus can reach agreement about many ACASTs, reducing the overhead of doing consensus.

When a consensus ends, TOW_decide actions with sequential numbers are generated for each ACAST for which agreement was reached (lines 22-31). The payload system and the wormhole network do not ensure timeliness or ordering of messages, so it is theoretically possible for the consensus to end without a local TO wormhole having received the SENT and RCVD messages for one of the messages agreed upon. In that (improbable) case, line 24 blocks the task of lines 22-31 until a SENT or RCVD message is received. Notice that the *has_msg* set of TOW_decide is set in line 26 taking into account the SENT/RCVD messages received, so different local wormholes can return different *has_msg* sets. This does not violate the TO Wormhole Agreement property, which does not state that these sets are identical. Furthermore, the correctness of the atomic multicast protocol does not depend on this set, which is used simply to indicate processes to which the message does not need to be sent (Algorithm 2, line 41).

A malicious server can attempt a denial of service attack against the TO wormhole by repeatedly calling TOW_sent or TOW_received. The latter is clearly ineffective since a call to TOW_received with information about an ACAST message that is unknown or for which the call has already been made returns error codes UNKNOWN and RECALL, respectively (see Table 2). The former is prevented by returning error code NO_RESOURCES. For returning this error code, there has to be a notion of how many calls are too many for the TO wormhole to handle. This limit can be defined in terms of the maximum number of calls per unit of time, or a call can be rejected simply when set $M\_received_i$ has a certain number of messages. To return

[5] Our first wormhole, the TTCB, was synchronous, i.e., it relied on known bounds for the communication and processing times [49]. This is no longer true for the TO wormhole.

RECALL the wormhole has to store data about previous calls to `TOW_sent` or `TOW_received` that did not return error. This can be done efficiently by storing only the sender and message identifiers. In an asynchronous system this would require infinite buffers, but in practice message identifiers can be reused periodically, so this data can be stored in a circular buffer in each local TO wormhole.

Appendix A.3. Correctness

The algorithm has to satisfy the three properties in Section 2.2. Recall that the system model requires that local TO wormholes either follow the algorithm or crash, and that at most $\lfloor \frac{n-1}{2} \rfloor$ can crash.

TO Wormhole Validity. *If a correct (i.e., non-crashed) local TO wormhole generates an action* `TOW_decide`*(*sender_id, msg_id, order_n, hash, has_msg*), then there was a call to* `TOW_sent`*(*sender_id, members, msg_id, msg_hash*) and at least* $\lfloor \frac{n-1}{2} \rfloor$ *calls to* `TOW_received`*(*,* members, msg_id, msg_hash, sender_id*) in different correct local TO wormholes. Moreover, if order_n*> 1*, then the local TO wormhole has done an action* `TOW_decide`*(*, *,* order_n-1*, *, *).*

First part. An inspection of the algorithm shows that there is a `TOW_decide` action for an ACAST message if it is decided by a consensus (lines 22-31). Furthermore, an ACAST message is proposed for consensus only if there was a call to `TOW_sent` and $\lfloor \frac{n-1}{2} \rfloor$ calls to `TOW_received` corresponding to that message (lines 17-21). Second part. Trivial from lines 2 and 22-31.

TO Wormhole Agreement. *No two different local TO wormholes generate two actions* `TOW_decide` *with the same* order_n *but different* sender_id*, msg_id,* hash *or* has_msg.

All local TO wormholes generate all `TOW_decide` actions by applying the same deterministic algorithm to the values returned by the same sequence of consensuses (lines 22-31), therefore all generate the same sequence of actions.

TO Wormhole Termination. *If there is a call to* `TOW_sent`*(*sender_id, members, msg_id, msg_hash*) and* $\lfloor \frac{n-1}{2} \rfloor$ *calls to* `TOW_received`*(*,* members, msg_id, msg_hash, sender_id*) in different correct local TO wormholes, then eventually every correct local TO wormhole does an action* `TOW_decide`*(*sender_id, msg_id, order_n, hash, has_msg*).*

An inspection of the algorithm shows that this property is always satisfied.

APPENDIX B. PROTOCOL CORRECTNESS

This section makes an argument about the correctness of the atomic multicast and the state machine replication protocols.

Appendix B.1. Atomic Multicast

Algorithm 2 implements an atomic multicast protocol if it satisfies properties AM1-AM4 in Section 3.2.1, for messages $m = S_{req}$ (see lines 7 and 46), assuming no more than $f = \lfloor \frac{n-1}{2} \rfloor$ servers are faulty.

AM1 Validity. *If a correct server multicasts a valid message* m*, then some correct server eventually delivers* m*.*

A correct server multicasts a message $m = S_{req}$ by calling `atomic_mcast`(m) (line 7). As m is valid, it is sent in an ACAST message to all other servers and `TOW_sent` is called (lines 8-10).

The properties of the channels guarantee that the ACAST message is eventually received by all the correct servers except the sender (line 15). Let us call these servers "receivers". As the message is valid, it is inserted in the set $A\_received_j$ in all correct receivers (lines 16-17). As the sender called `TOW_sent`, eventually the call to `TOW_received` in the correct receivers does not return an error, and the message is inserted in $A\_wait\_decide_i$ (lines 23-32).

As there is one correct sender and at least $\lfloor \frac{n-1}{2} \rfloor$ receivers, the properties of the TO wormhole guarantee that it eventually does an action `TOW_decide` (line 33) giving an order number to that message (TO Wormhole Termination), that that number is the same to all servers (TO Wormhole Agreement) and that the previous numbers are also given to other ACAST messages (TO Wormhole Validity). The order number is inserted in $T\_decided_i$ in all correct servers (line 34).

As the message and the order number are inserted respectively in $A\_wait\_decide_i$ and $T\_decided_i$ in all correct servers, and the same happens with all previous messages, the message is eventually delivered in the loop in lines 36-48.

AM2 Agreement. *If a correct server delivers a message* m*, then all correct servers eventually deliver* m*.*

This property is intuitive. All servers must deliver the same message for the following reasons:

- the servers deliver the messages for which they get a `TOW_decide` action, and all local TO wormholes generate the same actions in all correct servers (properties TO Wormhole Agreement and TO Wormhole Termination);
- the servers use the hash of the message in `TOW_decide` to know if they have the message and we assume this function is collision-resistant, so the message delivered must be the same;
- a server must receive the message it has to deliver, either sent by its original sender (in line 9) or, if that sender is faulty, by some other correct server (line 41).

AM3 Integrity. *For any identifier* ID*, every correct server delivers at most one message* m *with identifier* ID*, and if* sender(m) *is correct then* m *was previously multicast by* sender(m)*.*

The *ID* of a message is what makes it unique, which in this case is the hash of the set S_{req} (for a certain sender and group of servers S). The property has two parts. The proof of the first part is simple: correct servers only deliver

Algorithm 3 TO wormhole algorithm (local TOW t_i).

1: $M\_received_i \leftarrow \emptyset$ {set with messages received from the other TO wormholes}
2: $order_i \leftarrow 1$ {order of the next message}
3: $agreeing_i \leftarrow$ FALSE {indicates if t_i is running a consensus}

4: **when** TOW_sent(*id*, *members*, *msg_id*, *msg_hash*) is called **do**
5: {do tests and return error code in case of error (see Table 2)}
6: m_broadcast(⟨SENT, *id*, *members*, *msg_id*, *msg_hash*⟩)
7: return OK
8: **end when**

9: **when** TOW_received(*id*, *members*, *msg_id*, *msg_hash*, *sender_id*) is called **do**
10: {do tests and return error code in case of error (see Table 2)}
11: m_broadcast(⟨RCVD, *id*, *members*, *msg_id*, *msg_hash*, *sender_id*⟩)
12: return OK
13: **end when**

14: **when** m_deliver(m) **do**
15: $M\_received_i \leftarrow M\_received_i \cup \{m\}$
16: **end when**

17: **when** ($\exists members, \exists msg\_id, \exists sender\_id : \#_{\langle RCVD,*,members,msg\_id,*,sender\_id\rangle}(M\_received_i) \geq \lfloor \frac{n-1}{2} \rfloor + 1) \wedge (agreeing_i = \text{FALSE})$ **do**
18: $agreeing_i \leftarrow$ TRUE
19: $S_i \leftarrow \{\langle$SENT, *id*, *members*, *msg_id*, *msg_hash*$\rangle \in M\_received_i : \#_{\langle RCVD,*,members,msg\_id,*,sender\_id\rangle}(M\_received_i) \geq \lfloor \frac{n-1}{2} \rfloor + 1\}$
20: c_propose(S_i)
21: **end when**

22: **when** c_decide(S) **do**
23: **for all** $m \in S$ **do** {in deterministic order}
24: **wait until** $\exists m' \in M\_received_i : m'.sender\_id = m.id \wedge m'.members = m.members \wedge m'.msg\_id = m.msg\_id$
25: $S'_i \leftarrow \{m' \in M\_received_i : m'.sender\_id = m.id \wedge m'.members = m.members \wedge m'.msg\_id = m.msg\_id\}$
26: TOW_decide($m.sender\_id$, $m.msg\_id$, $order_i$, $m.msg\_hash$, $\{m'.id : m' \in S'_i\}$)
27: $order_i \leftarrow order_i + 1$
28: $M\_received_i \leftarrow M\_received_i \setminus S'_i$
29: **end for**
30: $agreeing_i \leftarrow$ FALSE
31: **end when**

messages for which they get an action TOW_decide, and the TO wormhole is trustworthy so it does not do two such actions for messages with the same hash.

The second part is trivial since a correct server gives the TO wormhole not only the hash of the message but also its own *id* (s_i in line 10), the TO wormhole is trustworthy, and the servers only deliver a message when they get an action TOW_decide with the *id* of the sender and the hash.

AM4 Total order. *If two correct servers deliver two messages m_1 and m_2 then both servers deliver the two messages in the same order.*

Correct servers deliver the messages in the order defined by the TO wormhole and given in the actions TOW_decide and the TO wormhole is trustworthy, so this is always true.

Appendix B.2. State Machine Replication

As discussed in Section 3, property SM1 (Initial state) is an issue of the initial system configuration and property SM4 (Determinism) is not enforced by the algorithm. Therefore, the properties we have to prove here are SM2 and SM3:

SM2 Agreement. *All correct servers execute the same commands.*

All servers execute commands in line 17 (Algorithm 1), picking these commands from the messages delivered by the atomic multicast protocol (lines 13-20), which delivers the same messages to all correct servers (property AM2 Agreement), therefore all servers execute the same commands.

SM3 Total order. *All correct servers execute the commands in the same order.*

The proof is similar to the proof of property SM2 but taking into account that the atomic multicast protocol delivers the messages in the same order to all correct servers and that the commands in the messages are executed in a deterministic order.

ACKNOWLEDGEMENTS

A preliminary version of this paper appeared in the Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems, 2004, pages 174–183. This work was partially supported by EC through projects FP7-

257475 (MASSIF) and FP7-257243 (TCLOUDS), and by the FCT through project PCT/EIA-EIA/115211/2009 (RC-Clouds), the Multiannual Programme (LASIGE), project PEst-OE/EEI/LA0021/2011 (INESC-ID), and the CMU-Portugal Programme. We thank Alysson Bessani and the anonymous reviewers for their comments on versions of this paper.

REFERENCES

[1] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004) Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, **1**, 11–33.

[2] Rushby, J. and Randell, B. (1983) A distributed secure system. *IEEE Computer*, **16**, 55–67.

[3] Verissimo, P., Neves, N. F., Cachin, C., Poritz, J., Powell, D., Deswarte, Y., Stroud, R., and Welch, I. (2006) Intrusion-tolerant middleware: The road to automatic security. *IEEE Security and Privacy*, **4**, 54–62.

[4] Horning, J. J., Lauer, H. C., Melliar-Smith, P. M., and Randell, B. (1974) A program structure for error detection and recovery. *Proceedings of International Symposium on Operating Systems: Theoretical and Practical Aspects*, Rocquencourt, France, 23-25 April, pp. 171–187, Springer-Verlag, Berlin.

[5] Bessani, A., Daidone, A., Gashi, I., Obelheiro, R. R., Sousa, P., and Stankovic, V. (2009) Enhancing fault / intrusion tolerance through design and configuration diversity. *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*, Estoril, Portugal, 29 June, IEEE Computer Society, Los Alamitos, CA.

[6] Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2011) OS diversity for intrusion tolerance: Myth or reality? *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, Hong Kong, 27–30 June, pp. 383 –394, IEEE Computer Society, Los Alamitos, CA.

[7] Alsberg, P. A. and Gray, J. D. (1976) A principle for resilient sharing of distributed resources. *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, CA, 13–15 October, pp. 562–570, IEEE Computer Society, Los Alamitos, CA.

[8] Gifford, D. K. (1979) Weighted voting for replicated data. *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 10–12 December, pp. 150–162, ACM, New York, NY.

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

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006) Bigtable: a distributed storage system for structured data. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 6–8 November, pp. 205–218, USENIX Association.

[11] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007) Dynamo: Amazon's highly available key-value store. *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, WA, 14–17 October, pp. 205–220, ACM, New York, NY.

[12] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003) The Google file system. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 19–22 October, pp. 29–43, ACM, New York, NY.

[13] White, T. (2009) *Hadoop: The Definitive Guide*. O'Reilly.

[14] Dean, J. and Ghemawat, S. (2004) MapReduce: simplified data processing on large clusters. *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 6-8 December, pp. 137–150, USENIX Association.

[15] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010) Zookeeper: wait-free coordination for internet-scale systems. *Proceedings of the USENIX Technical Conference*, Boston, MA, 23–25 June, USENIX Association.

[16] Fraga, J. S. and Powell, D. (1985) A fault- and intrusion-tolerant file system. *Proceedings of the 3rd IFIP International Conference on Computer Security*, Dublin, Ireland, 12–15 August, pp. 203–218, Elsevier, North-Holland.

[17] Powell, D. and Stroud, R. J. (eds.) (2003) *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*.

[18] Lala, J. H. (ed.) (2003) *Foundations of Intrusion Tolerant Systems*. IEEE Computer Society Press.

[19] Bessani, A. N., Sousa, P., Correia, M., Neves, N. F., and Verissimo, P. (2008) The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy*, pp. 44–51.

[20] Vukolić, M. (2010) The Byzantine empire in the intercloud. *ACM SIGACT News*, **41**, 105–111.

[21] Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2011) DepSky: Dependable and secure storage in a cloud-of-clouds. *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*, Saltzburg, Austria, 10-13 April, pp. 31–46, ACM, New York, NY.

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

[23] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2007) Zyzzyva: Speculative Byzantine fault tolerance. *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, WA, 14–17 October, ACM, New York, NY.

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

[25] Reiter, M. (1995) The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, vol. 938 of *Lecture Notes in Computer Science*, pp. 99–110, Springer-Verlag, Berlin.

[26] Cachin, C. and Poritz, J. A. (2002) Secure intrusion-tolerant replication on the Internet. *Proceedings of the IEEE/IFIP 32nd International Conference on Dependable Systems and Networks*, Bethesda, MD, 23-26 June, pp. 167–176, IEEE Computer Society, Los Alamitos, CA.

[27] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006) HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 6–8 November, pp. 177–190, USENIX Association.

[28] Amir, Y., Coan, B., Kirsch, J., and Lane, J. (2008) Byzantine replication under attack. *Proceedings of the IEEE/IFIP*

38th International Conference on Dependable Systems and Networks, Anchorage, AK, 24-27 June, pp. 197–206, IEEE Computer Society, Los Alamitos, CA.

[29] Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009) Making Byzantine fault tolerant systems tolerate Byzantine faults. *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, Boston, MA, 22–24 April, pp. 153–168, USENIX Association.

[30] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009) Upright cluster services. *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, Big Sky, MT, 11–14 October, pp. 277–290, ACM, New York, NY.

[31] Singh, A., Fonseca, P., Kuznetsov, P., Rodrigues, R., and Maniatis, P. (2009) Zeno: eventually consistent Byzantine-fault tolerance. *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, Boston, MA, 22–24 April, pp. 169–184, USENIX Association.

[32] Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. (2009) Spin one's wheels? Byzantine fault tolerance with a spinning primary. *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, Niagara Falls, NY, 28–30 September, pp. 135–144, IEEE Computer Society, Los Alamitos, CA.

[33] Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010) The next 700 BFT protocols. *Proceedings of the 5th ACM SIGOPS/EuroSys European Systems Conference*, Paris, France, 13–16 April, pp. 363–376, ACM, New York, NY.

[34] Serafini, M., Bokor, P., Dobre, D., Majuntke, M., and Suri, N. (2010) Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. *Proceedings of the IEEE/IFIP 40th International Conference on Dependable Systems and Networks*, Chicago, IL, 28 June–1 July, pp. 353 –362, IEEE Computer Society, Los Alamitos, CA.

[35] Correia, M., Neves, N. F., and Verissimo, P. (2004) How to tolerate half less one Byzantine nodes in practical distributed systems. *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, Florianopolis, Brazil, 18–20 October, pp. 174–183, IEEE Computer Society, Los Alamitos, CA.

[36] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. (2007) Attested append-only memory: making adversaries stick to their word. *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, WA, 14–17 October, pp. 189–204, ACM, New York, NY.

[37] Veronese, G. S., Correia, M., Bessani, A. N., C., L., and Verissimo, P. (2011) Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, to appear.

[38] Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. (2010) EBAWA: Efficient Byzantine agreement for wide-area networks. *Proceedings of the 12th IEEE International Symposium on High-Assurance Systems Engineering*, San Jose, CA, 3–4 November, pp. 10–19, IEEE Computer Society, Los Alamitos, CA.

[39] Littlewood, B. and Strigini, L. (2004) Redundancy and diversity in security. Samarati, P., Ryan, P., Gollmann, D., and Molva, R. (eds.), *Computer Security – ESORICS 2004, 9th European Symposium on Research Computer Security*, pp. 423–438, LNCS 3193, Springer-Verlag, Berlin.

[40] Stroud, R., Welch, I., Warne, J., and Ryan, P. (2004) A qualitative analysis of the intrusion tolerance capabilities of the MAFTIA architecture. *Proceedings of the IEEE/IFIP 34th International Conference on Dependable Systems and Networks*, Florence, Italy, 28 June–1 July, IEEE Computer Society, Los Alamitos, CA.

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

[42] Trusted Computing Group (2006) Trusted computing platform module main specification. version 1.2, revision 94, http://www.trustedcomputinggroup.org.

[43] Casimiro, A., Mendizabal, O., and Verissimo, P. (2006) On the development of dependable embedded applications using specialized wormholes. *Proceedings of the 3rd International Workshop on Dependable Embedded Systems*, Leeds, UK, 1 October, pp. 67–71, IEEE Computer Society, Los Alamitos, CA.

[44] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985) Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, **32**, 374–382.

[45] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2010) Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, **21**, 452–465.

[46] Menezes, A. J., Oorschot, P. C. V., and Vanstone, S. A. (1997) *Handbook of Applied Cryptography*. CRC Press.

[47] Kent, S. and Atkinson, R. (1998), Security architecture for the internet protocol. IETF Request for Comments: RFC 2093.

[48] NIST (2002), FIPS 180-2, Secure Hash Standard.

[49] Correia, M., Verissimo, P., and Neves, N. F. (2002) The design of a COTS real-time distributed security kernel. *Proceedings of the 4th European Dependable Computing Conference*, Toulouse, France, 23–25 October, pp. 234–252, Springer-Verlag, Berlin.

[50] Security evaluations for IBM products. http://www.ibm.com/security/standards/st_evaluations.shtml.

[51] Lantronix XPort embedded ethernet device server. http://www.lantronix.com/device-networking/embedded-device-servers/xport.html.

[52] Barham, P., Dragovic, B., Fraiser, K., Hand, S., Harris, T., Ho, A., Neugebaurer, R., Pratt, I., and Warfield, A. (2003) Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 19–22 October, pp. 164–177, ACM, New York, NY.

[53] Viega, J. and McGraw, G. (2002) *Building Secure Software*. Addison Wesley.

[54] Castro, M., Rodrigues, R., and Liskov, B. (2003) BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, **21**, 236–269.

[55] Hadzilacos, V. and Toueg, S. (1994) A modular approach to fault-tolerant broadcasts and related problems. Tech. Rep. TR94-1425, Cornell University, Department of Computer Science.

[56] Cachin, C., Kursawe, K., Petzold, F., and Shoup, V. (2001) Secure and efficient asynchronous broadcast protocols (extended abstract). Kilian, J. (ed.), *Advances in Cryptology: CRYPTO 2001*, vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer-Verlag, Berlin.

[57] Correia, M., Neves, N. F., and Verissimo, P. (2006) From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Computer Journal*, **41**, 82–96.

[58] Schiper, A. (1997) Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, **10**, 149–157.

[59] Moniz, H., Neves, N. F., Correia, M., and Verissimo, P. (2006) Randomized intrusion-tolerant asynchronous services. *Proceedings of the IEEE/IFIP 36th International Conference on Dependable Systems and Networks*, Philadelphia, PA, 25-28 June, pp. 568–577, IEEE Computer Society, Los Alamitos, CA.

[60] Reiter, M. (1994) Secure agreement protocols: Reliable and atomic group multicast in Rampart. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, VA, 2-4 November, pp. 68–80, ACM, New York, NY.

[61] Li, J. and Mazieres, D. (2007) Beyond one-third faulty replicas in Byzantine faul tolerant systems. *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, Cambridge, MA, 11–13 April, pp. 131–144, USENIX Association.

[62] Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., and Wylie, J. (2005) Fault-scalable Byzantine fault-tolerant services. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 23–26 October, pp. 59–74, ACM, New York, NY.

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

[64] Lamport, L. (1978) Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**, 558–565.

[65] Bracha, G. and Toueg, S. (1985) Asynchronous consensus and broadcast protocols. *Journal of the ACM*, **32**, 824–840.

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

[67] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. (2001) The SecureRing group communication system. *ACM Transactions on Information and System Security*, **4**, 371–406.

[68] Moser, L. E., Melliar-Smith, P. M., and Narasimhan, N. (2000) The SecureGroup communication system. *Proceedings of the IEEE Information Survivability Conference*, Hilton Head, SC, 25–27 January, pp. 507–516, IEEE Computer Society, Los Alamitos, CA.

[69] Correia, M., Neves, N. F., Lung, L. C., and Verissimo, P. (2007) Worm-IT – a wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, **80**, 178–197.

[70] Hendricks, J., Sinnamohideen, S., Ganger, G., and Reiter, M. (2010) Zzyzx: Scalable fault tolerance through Byzantine locking. *Proceedings of the IEEE/IFIP 40th International Conference on Dependable Systems and Networks*, Chicago, IL, 28 June–1 July, pp. 363 –372, IEEE Computer Society, Los Alamitos, CA.

[71] Ramasamy, H. and Cachin, C. (2006) Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. *Proceedings of the 9th International Conference on Principles of Distributed Systems*, vol. 3974 of *Lecture Notes in Computer Science*, pp. 88–102, Springer-Verlag, Berlin.

[72] Lamport, L. (1998) The part-time parliament. *ACM Transactions Computer Systems*, **16**, 133–169.

[73] Zielinski, P. (2004) Paxos at war. Tech. Rep. UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.

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

[75] Amir, Y., Coan, B., Kirsch, J., and Lane, J. (2007) Customizable fault tolerance for wide-area replication. *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, Beijing, China, 10–12 October, pp. 66–80, IEEE Computer Society, Los Alamitos, CA.

[76] Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J., and Zage, D. (2010) Steward: Scaling Byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, **7**, 80–93.

[77] Mao, Y., Junqueira, F. P., and Marzullo, K. (2009) Towards low latency state machine replication for uncivil wide-area networks. *Proceedings of the 5th Workshop on Hot Topics in System Dependability*, Estoril, Portugal, 29 June, IEEE Computer Society, Los Alamitos, CA.

[78] Yin, J., Martin, J., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003) Separating agreement from execution for Byzantine fault tolerant services. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 19–22 October, pp. 253–267, ACM, New York, NY.

[79] Distler, T., Popov, I., Schröder-Preikschat, W., Reiser, H. P., and Kapitza, R. (2011) SPARE: Replicas on hold. *Proceedings of the 18th Network and Distributed System Security Symposium*, San Diego, CA, 6–9 February, pp. 407–420, Internet Society.

[80] Wood, T., Singh, R., Venkataramani, A., Shenoy, P., and Cecchet, E. (2011) ZZ and the art of practical BFT execution. *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*, Saltzburg, Austria, 10-13 April, pp. 123–138, ACM, New York, NY.

[81] Distler, T. and Kapitza, R. (2011) Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*, Saltzburg, Austria, 10-13 April, pp. 91–106, ACM, New York, NY.

[82] Mpoeleng, D., Ezhilchelvan, P., and Speirs, N. (2003) From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits. *Proceedings of the IEEE/IFIP 33rd International Conference on Dependable Systems and Networks*, San Francisco, CA, Jun., pp. 227–236, IEEE Computer Society, Los Alamitos, CA.

[83] Abraham, I., Aguilera, M. K., and Malkhi, D. (2010) Fast asynchronous consensus with optimal resilience. *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, MA, 13–15 September, pp. 4–19, Springer-Verlag, Berlin.

[84] Levin, D., Douceur, J. R., Lorch, J. R., and Moscibroda, T. (2009) Trinc: small trusted hardware for large distributed systems. *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, 8–10 May, pp. 1–14, USENIX Association.

[85] Pedone, F., Schiper, A., Urbán, P., and Cavin, D. (2002) Solving agreement problems with weak ordering oracles. *Proceedings of the 4th European Dependable Computing Conference*, Toulouse, France, 23–25 October, pp. 44–61, Springer-Verlag, Berlin.

[86] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. (2009) Tiered fault tolerance for long-term integrity. *Proccedings of the 7th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 24–27 February, pp. 267–282, USENIX Association.

[87] Roeder, T. and Schneider, F. B. (2010) Proactive obfuscation. *ACM Transactions on Computer Systems*, **28**, 4:1–4:54.

[88] Marchetti, C., Baldoni, R., Tucci-Piergiovanni, S., and Virgillito, A. (2006) Fully distributed three-tier active software replication. *IEEE Transactions on Parallel and Distributed Systems*, **17**, 633 –645.

[89] Sousa, P., Neves, N. F., and Verissimo, P. (2007) Hidden problems of asynchronous proactive recovery. *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*, Edinburgh, UK, 25–28 June, IEEE Computer Society, Los Alamitos, CA.