Solving Vector Consensus with a Wormhole

Nuno F. Neves, *Member*, *IEEE*, Miguel Correia, *Member*, *IEEE*, and Paulo Veríssimo, *Member*, *IEEE*

Abstract—This paper presents a solution to the vector consensus problem for Byzantine asynchronous systems augmented with wormholes. Wormholes prefigure a hybrid distributed system model, embodying the notion of an enhanced part of the system with "good" properties otherwise not guaranteed by the "normal" weak environment. A protocol built for this type of system runs in the asynchronous part, where f out of $n \geq 3f+1$ processes might be corrupted by malicious adversaries. However, sporadically, processes can rely on the services provided by the wormhole for the correct execution of simple operations. One of the nice features of this setting is that it is possible to keep the protocol completely time-free and, in addition, to circumvent the FLP impossibility result by hiding all time-related assumptions in the wormhole. Furthermore, from a performance perspective, it leads to the design of a protocol with a good time complexity.

Index Terms—Distributed systems, Byzantine asynchronous protocols, consensus.

1 Introduction

Replication of software components is a well-known technique for improving the overall dependability of distributed systems. If one can realistically assume a bound on the number of failures, it is possible to avert their effect by voting on the values returned by the replicas. However, the construction of replicated services is a complex task, especially if one considers environments where time limits on the system operations are undefined and where malicious (or Byzantine) failures can occur. In this setting, consensus can play an important role because it can be used in the solution of several problems, such as total order broadcast or atomic commitment [1].

This paper presents a protocol for the vector consensus problem in a Byzantine asynchronous system augmented with a wormhole. Vector consensus is a form of agreement where processes start with a value and attempt to decide on a same vector. The main characteristic of this vector is that it must contain a majority of values proposed by correct (nonfaulty) processes. Although relatively simple in its specification, the consensus problem has been shown to be impossible to solve in a deterministic way in an asynchronous system [2]. The only two solutions for vector consensus that we are aware of utilize Byzantine failure detectors to circumvent this impossibility result [3], [4]. Failure detectors were originally proposed to discover if some processes engaged in a distributed computation had crashed [5]. Later, they were extended in order to detect malicious behavior and to support the execution of Byzantine resilient protocols [6], [7], [8].

Failure detectors are a very elegant concept from both a theoretical and architectural point of view. They lead to protocols that are completely asynchronous since all time-related assumptions can be hidden in their implementation. On a malicious environment, however, they are relatively difficult to build. Although recent advances in the area of intrusion detection have resulted in significant improvements to the existing solutions, there are still many aspects that need to be addressed. It is inherently difficult to envision all possible malicious activities and, at the same time, to separate them from correct actions. Practical experience demonstrates that is quite easy for a detection system to generate thousands of alarms per day, most of which are false alarms (i.e., benign events that are incorrectly considered attacks) [9]. This problem will tend to be exacerbated in the future since the hacker community has been developing tools whose only purpose is to flood the detection systems with unrelated attacks to prevent them from carrying on their work (i.e., a denial of service attack) [10]. Furthermore, it has been argued that Byzantine consensus cannot be solved with a generic failure detector and that at least some of its components will have to be protocol dependent [11].

In this paper, we use a different approach. We begin by postulating a hybrid distributed system model, foreseeing the notion of an enhanced part of the system with "good" properties otherwise not guaranteed by the "normal" weak environment. Wormholes are the constructs which materialize this notion, augmenting the asynchronous environment with the availability of a few services not supported by the weak environment [12]. For example, they can offer some secure operations in a system where Byzantine failures occur. Contrary to "normal" operations, these operations would always return the expected results or, in the worst case, they would deliver no data if a crash occurred. The paper utilizes a specific distributed wormhole that offers secure services. Since the wormhole is a small component and provides limited functionality, it can be proven correct and constructed with high coverage of its assumptions.[1]

The vector consensus protocol is executed by a set of processes running in the asynchronous part of the system where they can experience arbitrary failures. One can assume that an adversary controls the behavior of a

• *The authors are with the Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Bloco C6, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: {nuno, mpc, pjv}@di.fc.ul.pt.*

Manuscript received 10 Aug. 2004; revised 21 Jan. 2005; accepted 28 Jan. 2005; published online 20 Oct. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0200-0804.

1. An implementation of a wormhole has been built to prove the concept—for details see [13]. This wormhole, however, has stronger properties than what we will require in this paper.

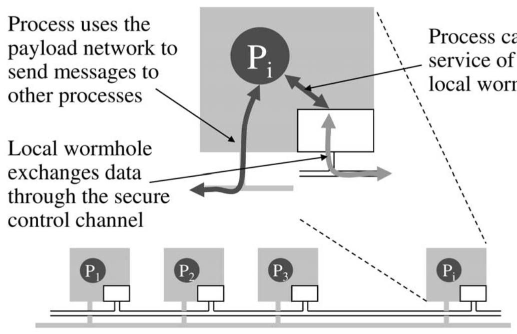

Process uses the payload network to send messages to other processes
Local wormhole exchanges data through the secure control channel
Process calls a service of the local wormhole
P_i
P_1
P_2
P_3
P_i

Fig. 1. Architecture of a system with a distributed wormhole (represented in dark, the payload system is asynchronous and can have Byzantine failures; represented in white, the wormhole is partially synchronous and can only crash).

number of processes (at most f out of a total $n \geq 3f+1$) and makes them act maliciously. For example, these processes can transmit bad values and collude with one another while attempting to violate the protocol properties. Moreover, the adversary can attack the messages exchanged among correct processes when they travel through the network. Normally, protocols for these environments would suffer in efficiency and/or determinism. However, since the wormhole is accessible to all processes, sporadically they can rely on its services for the correct execution of (small) crucial operations.

The main contributions of the paper are: 1) It demonstrates with a new protocol that vector consensus can be solved with the help of a wormhole. 2) It shows that the protocol can circumvent the FLP impossibility result without having to explicitly make any time assumptions. The only hypothesis it needs to make is that the wormhole services eventually terminate and provide the expected results. 3) From a performance perspective, the protocol has good time complexity.

2 System Model

The consensus protocol is run by a finite set $P = \{p_1, \ldots, p_n\}$ of n processes. Processes are executed in a system divided in two parts: the *payload* system and the *wormhole* (see Fig. 1). The payload system corresponds to what is usually perceived as "the system," and it includes one node per process, where computations are carried out, and a *payload* network, where messages can be exchanged. A node contains the expected software, such as the operating system and middleware components, which offers services that can be invoked by the processes. Processes can only send messages through the payload network, which is assumed to be fully connected.

The wormhole is a distributed component with local parts at each node and a private network called the *control channel*. The local wormholes can be treated by processes like any other software component in the node.[2] They provide a limited set of services that can be invoked (see Section 2.1). In some cases, the implementation of these services might require the exchange of information among the local wormholes. Whenever this happens, the control channel is used to accomplish this communication, and not the payload network.

2. For simplicity, we will be assuming throughout the paper that each node has a process and a local wormhole. In practice, we could have a process in one node and the local wormhole in another, and they would communicate with each other using a secure channel on the payload network.

Synchrony Assumptions. The payload system, and, consequently, the execution of the processes, is asynchronous. As a result, there can be no assumptions about the relative speed of processes and no bounds on message delivery delays (for messages transmitted through the payload network). The wormhole is assumed to have enough synchrony to ensure that its services eventually terminate and a result is returned to the processes. A partially synchronous model would be enough to implement the services considered in this paper [14], [15], [16].

An invocation of a wormhole service is initiated by an asynchronous process and then is carried out in the partially synchronous part of the system. From the point of view of a process, there are no limits on the time a request takes to reach the wormhole interface (a similar reasoning can be made for the response). Therefore, this implies that there are no time bounds on the invocations of wormhole services (one can only assume that they are eventually completed).

Failure Assumptions. The payload system can suffer Byzantine faults. Since processes are executed in this part of the system, they are allowed to exhibit arbitrary behavior whenever they fail. This might mean, for instance, that they can stop working, skip some steps in the protocol, give invalid information to the wormhole or the other processes, or start colluding with other malicious processes while attempting to break the properties of the protocol.

Communication through the payload network and service calls to the local wormhole can also suffer Byzantine faults. Messages can, for example, be removed or altered by an adversary while they are in transit between processes. Service invocations can be intercepted and modified while in progress between a process and the local wormhole interface. It is assumed, however, that communication channels are *fair*, which means that if a process sends infinitely many messages to a single destination, then infinitely many of those messages are correctly received. The same type of assumption is also made for the calls to the local wormhole. If, for some reason, one of these assumptions is violated for a process, then this process is considered faulty.

The wormhole, which includes the control channel, only suffers crash failures. Therefore, a local wormhole either provides its services as expected or it simply stops running. This assumption should hold even if malicious adversaries manage to attack and compromise a node with a local wormhole (for implementation details, see Section 5 and [13]).

The protocol presented in the paper requires the invariant that no more than f processes fail, out of a total of $n \geq 3f+1$.

Payload Channel Assumptions. To simplify the communication among processes, it is possible to use secure cryptographic algorithms (e.g., hash functions) to build channels with stronger properties. In the rest of the paper, it

TABLE 1
Interface of the Wormhole

Local Authentication: $eid, chlg\_sign \leftarrow W\_localAuth(key, chlg)$
Trusted Block Agreement: $error, value, prop\text{-}ok \leftarrow$ $\leftarrow W\_TBA(eid, elist, agid, quorum, decision, value)$

will be assumed that each pair of processes is connected by a *secure channel* on the payload network, with the following characteristics:

- *Eventual reliability*: If p_i and p_k are correct and p_i sends a message M to p_k, then p_k eventually receives M.
- *Integrity*: If p_i and p_k are correct and p_k receives a message M with $sender(M) = p_i$, then M was really sent by p_i and M was not modified in the channel.[3]

These two properties are simple to implement if one assumes that hash functions cannot be subverted by malicious entities and that each pair of correct processes shares a symmetric key only known to them. Eventual reliability is achieved by retransmitting the messages periodically until an acknowledgment arrives (do not forget our previous assumption of fair channels). Integrity is accomplished with the help of Message Authentication Codes (MACs), which allow the detection of message forgeries and modifications [17]. A MAC is a cryptographic checksum, obtained with a hash function and a symmetric key. When p_i sends a message M to p_k, it concatenates a MAC to M. When p_k receives the message, it calculates an equivalent MAC and compares it with the incoming MAC. If they are different, then the message is fake or was altered and, therefore, it is discarded.

2.1 Wormhole Interface

The wormhole provides a small number of services to the processes. The prototypes for these services are presented in Table 1. The first service is not directly used by the consensus protocol, but by the code that will eventually call the consensus function. The *Local Authentication Service* serves to set up the communication between the process and the local wormhole. It lets the process authenticate the wormhole (through the exchange of challenges—*chlg* and $chlg\_sign$) and establish a shared symmetric key (*key*) to protect the data transmitted between the process and the local wormhole. This service also returns a unique identifier, called the *entity identifier* (*eid*), that should be used in future interactions with the wormhole.

The only service invoked by the consensus protocol is the *Trusted Block Agreement Service* or, simply, *TBA Service*. In this service, processes propose a value and then they get a result. The result is calculated by applying an agreement function to the proposed values. Values are blocks of data with a small size (20 bytes). Since wormhole resources are limited and have to be shared among all applications, this service should be used only to execute critical steps of protocols that run mostly in the payload part of the system.[4] The service guarantees the following properties:

- *Integrity*. Every correct process decides at most one result.
- *Agreement*. No two correct processes decide differently.
- *Validity*. If a correct process decides *result*, then *result* is obtained applying the function *decision* to the values proposed.
- *Termination*. Provided that *quorum* proposals arrive by different processes to the wormhole, every correct process eventually decides a result.

The parameters of the TBA service have the following meanings (see Table 1): *eid* is the identifier previously mentioned, and *elist* is a list of the eid's of the processes involved in the TBA. *agid* is an unique identifier for the agreement and this group of processes (*agid* can be reused for different instances of *elist*). The TBA applies the *decision* function to at least *quorum* values. Therefore, *quorum* defines the minimum set of values that will be considered in the agreement. The consensus protocol will use only one decision function, called the *TBA_MAJORITY*, which returns the value proposed by most entities. After the conclusion of the agreement, the service returns the following values: 1) an error code, 2) the decided value, and 3) a mask *prop-ok* with one bit set for each process that proposed the value that was decided.

If a process is late and proposes after the TBA has started to be executed, then its value will probably be disregarded. Nevertheless, it will be able to collect the decision that was calculated using the values proposed by the other processes. For simplicity, it is assumed throughout the paper that the wormhole can record the output of a TBA for a long time. In practice, the wormhole will have to garbage collect these results, and a delayed process that is unable to obtain a decision necessary for its correct execution should be forced to crash and fail. One should notice that this action will not compromise the safety and liveness properties of the protocol since it is built to tolerate arbitrary failures (this behavior simply corresponds to a specific kind of failure).

3 The Problem

In the consensus problem, a group of processes attempts to reach agreement on a set of values despite a number of failures. The decision is calculated using the original values proposed by the processes. More precisely, the *Vector Consensus* problem can be defined as [3]:

- *Validity*. Every correct process decides on a vector $vect$ of size n such that:
  1. For every $1 \leq i \leq n$, if process p_i is correct, then $vect[i]$ is either the initial value of p_i or the value $\perp$ and

3. The predicate *sender(M)* returns the sender field of the message header.

4. The current implementation uses an admission control mechanism to prevent processes from causing the exhaustion of the wormhole's resources. If processes try to use the wormhole services too much, then they get delayed. Nevertheless, the correctness of the protocols is always ensured because they are built for asynchronous systems.

2. at least $f+1$ elements of the vector $vect$ are the initial values of correct processes.

- *Agreement*. No two correct processes decide differently.
- *Termination*. Every correct process eventually decides.

Validity and Agreement properties must always be true, otherwise something bad might occur. The termination property asserts that something good will eventually happen.

4 Solving Vector Consensus

This section presents a protocol for vector consensus on an asynchronous system augmented with a wormhole. The system, with the exception of the wormhole, can suffer from arbitrary failures. The protocol uses the payload channel to transmit most of the information, which includes the initial values and the vectors of values. Then, at certain points of the execution, the protocol relies on the TBA service of the wormhole to securely exchange a small amount of data, that enables it to determine if a decision can be reached.

Before we start a more detailed description of the protocol, we would like to emphasize that the consensus problem does not become much easier simply because our solution utilizes a low level agreement service (the TBA). First, the difficulties created by the asynchronous setting continue to exist because any of process execution, communication through the payload channel, and wormhole service invocations might be delayed by an unknown amount of time. Second, both the node and the communication channels might experience Byzantine failures, which means that wrong, contradictory, or malicious data can be received by the correct processes.

4.1 The Protocol

Processes can only start running the protocol after they perform some setup operations, such as calling the wormhole's Local Authentication Service to obtain an eid identifier and selecting the group of processes that will participate in the consensus. Moreover, they will also need to get a unique consensus identifier prior to each execution of the protocol.

A process initiates the protocol when it calls the consensus function (see Algorithm 1 (Fig. 2)). This function has three arguments, and the first two should be the same in all processes, otherwise more than one instance of consensus is started. Argument $elist$ determines the group and contains the identifiers of all processes involved in the consensus, cid is the consensus identifier, and $value_i$ is the value being proposed by process p_i. $value_i$ has an arbitrary number of bytes, which means that it could have, for example, one bit or a billion bytes (the first case would correspond to the problem of *binary* vector consensus).

The execution of the protocol can be divided into two phases. In the first phase, a process digitally signs its value using an asymmetric algorithm (e.g., RSA) and then it broadcasts the signed value through the payload network (Lines 6 and **B-value** message in 8). Next, it constructs a vector with the initial and received values (Lines 7 and 9-13). Process p_i stores the vector in the i's position of *array-of-vectors*. This vector has $2f+1$ filled entries, which

Algorithm 1 Vector consensus protocol (run by every p_i).

```
1  function consensus(elist, cid, $value_i$)
2  round ← 0                                           {round number}
3  hash-v ← ⊥                          {hash of the decided vector of values}
4  bag-decide ← ⊥                        {bag of received Decide messages}
5  array-of-vectors ← ((⊥, ..., ⊥), ..., (⊥, ..., ⊥))          {array of vectors}

6  signed-value$_i$ ← sign(i, $value_i$)                        {sign my value}
7  array-of-vectors[i][i] ← signed-value$_i$       {save my value in my vector}
8  broadcast(B-value, i, signed-value$_i$)              {use payload network}
9  repeat
10     receive(B-value, k, signed-value$_k$)
11     if (signed-value$_k$ is correctly signed) then
12         array-of-vectors[i][k] ← signed-value$_k$            {save $p_k$'s value}
13 until (array-of-vectors[i] has $2f+1$ values from different processes)
14 broadcast(B-vector, i, array-of-vectors[i])          {use payload network}
15 activate task(T1, T2)                           {start two concurrent tasks}

16 Task T1:
17 when (receive(B-vector, k, vector$_k$)) do
18     array-of-vectors[k] ← vector$_k$
19 when (receive(Decide, k, vector$_k$)) do
20     bag-decide ← bag-decide ∪ {vector$_k$}
21 when (hash-v ≠ ⊥) and
           ($\exists_{vector_k \in bag-decide}$ : Hash(vector$_k$) = hash-v)) do
22     return (vector$_k$)

23 Task T2:
24 while true do
25     index ← (round mod n) + 1
26     round ← round+1
27     agid ← cid + 1/round
28     v ← getNextGoodVector(array-of-vectors, index)
29     out ← W_TBA(eid, elist, agid, $2f+1$, TBA_MAJORITY, Hash(v))
30     if (at least $f+1$ proposed the same value) then
31         if (out.value = Hash(v)) then
32             if (not all processes proposed the same value) then
33                 send (Decide, i, v) to all processes that did not propose v
34             return (v)
35         else
36             hash-v = out.value
37             break ()                          {finish the execution of this task}
```

Fig. 2. The vector consensus protocol (Algorithm 1).

ensures that at least $f+1$ of them belong to correct processes (at most f processes can fail). Moreover, since processes use secure channels to transmit information (see Section 2), these $f+1$ or more entries contain the originally proposed values and not some erroneous information that was replaced in the network by an adversary. The constructed vectors, however, will possibly be different at the various processes because **B-value** messages might suffer distinct delays, and malicious processes might not send the same values to all destinations.

In the second phase, the protocol chooses one of the constructed vectors. A process begins by broadcasting its vector through the payload network (**B-vector** message) and then it initiates two tasks that will run in parallel (Lines 14-15). Task T1 receives the several types of messages (Lines 17-20) and terminates the protocol when a decision is attained (Line 22). The other task ensures that correct processes agree on the same vector. To achieve this objective, it executes a round-based procedure where processes try to pick one of the exchanged vectors. The current round vector is selected if it collects enough votes. Otherwise, a new round is initiated, and another vector is considered for approval. A vector might not be chosen due to two basic reasons: It still has not arrived at the destinations and, consequently, not enough processes will give their support to it, or it was sent by an adversary who is trying to disrupt the protocol, either by broadcasting bad vectors or by transmitting the vector to a small subset of the processes.

A process executing task T2 carries out the following steps. It starts by updating the $round$ counter and the wormhole agreement identifier $agid$ (Lines 26-27). If one assumes that cid is an integer, it is possible to deterministically generate distinct agreement identifiers by simply adding $1/round$ to cid. Next, it calculates an index based on the current round number and chooses the vector that will be proposed (Lines 25 and 28). Below, we explain that the selection procedure guarantees that the vector contains good values. Then, the process calls the TBA service of the wormhole with an hash of the vector (obtained with the *Hash()* function) and a quorum of $2f+1$ and waits for the outcome of the agreement.

Processes can propose distinct hashes; however, they all get the same result from the TBA. If one of the hashes was backed up by $f+1$ or more processes, then at least one correct process has the vector corresponding to this hash. Therefore, this vector can be selected as the decision of consensus and the function can return (Lines 30-34).

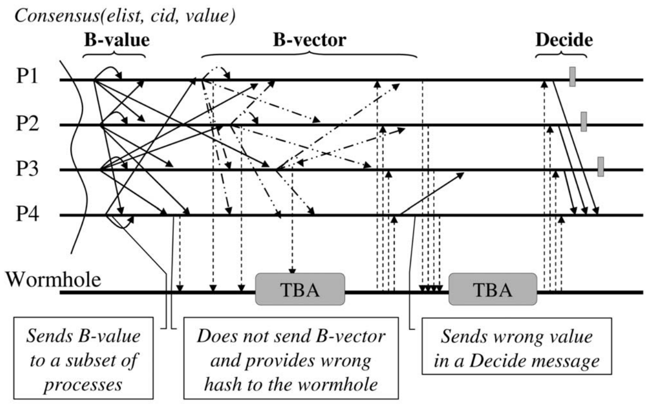
In some cases, a few correct processes might not have the decided vector (e.g., an adversary sends a good vector to a subset of the processes). To ensure that these processes are also able to terminate, a **Decide** message containing the vector is transmitted through the payload network (Lines 32-33). Although not shown in Algorithm 1, mask *out.prop-ok* is used to determine which processes proposed the chosen hash. A malicious process might also transmit a **Decide** message with bad content, attempting to trick processes to return a wrong value. To prevent this situation, processes save the hash provided by the TBA before stopping task T2 (Lines 36-37) and, then, they use the hash to find out which vector should be returned (Lines 19-22).

Algorithm 2. Select the vector to be proposed

```
1 function getNextGoodVector(array-of-vectors, index)
2 while true do
3   new-vector ← (⊥, ..., ⊥ )
4   for (each k nonempty entry of array-of-vectors[index]) do
5     if (array-of-vectors[index][k] is correctly signed) then
6       new-vector[k] ←
          array-of-vectors[index][k].value   {only save the value}
7   if (new-vector has at least 2f + 1 nonempty entries) then
8     return (new-vector)
9   index ← (index mod n) + 1
```

4.1.1 Selecting a Good Vector

When a process receives a vector, it has no simple way to determine if the sender was malicious or not. The process is only certain of one thing—the vector was not modified from the source until the arrival because communication channels are secure. However, if the vector was transmitted by a malicious process, then it can contain incorrect information (e.g., it can say that correct process p_k proposed value X when, in fact, it sent Y). To avoid this type of attack, the *getNextGoodVector()* function only returns good vectors. A vector is *good* if it satisfies the requirements imposed by the consensus specification (see Section 4), which basically implies that it has at least $2f+1$ filled entries and, from those entries, $f+1$ or more came from correct processes.



Consensus(elist, cid, value)
B-value
B-vector
Decide
P1
P2
P3
P4
Wormhole
TBA
TBA
Sends B-value to a subset of processes
Does not send B-vector and provides wrong hash to the wormhole
Sends wrong value in a Decide message

Fig. 3. Example execution of the protocol (with $n=4$ and $f=1$).

Function *getNextGoodVector()* goes through the *array-of-vectors* in a round-robin fashion (*array-of-vectors[index], array-of-vectors[(index mod n) + 1], ...*) until a good vector is found (see Algorithm 2). A few tests are performed to accomplish this objective. The first test finds out if the entries of the vector contain valid values, i.e., the actually proposed values and not some replacements (Lines 4-6). Since a value has an associated digital signature that cannot be forged by an adversary, by verifying its signature one can be sure of the validity of a value. Next, the function finds out if the resulting vector has at least $2f+1$ values (Lines 7-8). Since empty vectors are automatically skipped, no time is wasted on trying to agree on vectors belonging to crashed processes, which potentially increases the overall performance of the protocol.

Since messages can suffer from unknown delays, when a process calls *getNextGoodVector()* for the first time, it might not have received the vectors from the other processes. However, the process will not be blocked forever in this function due to the following reasons: First, task T1 runs in parallel and continues to add vectors to *array-of-vectors* (Lines 17-18 of Algorithm 1); second, at least one good vector always exists—the vector belonging to this process.

4.2 An Example Execution

Fig. 3 contains a space-time diagram with the execution of four processes and the wormhole. To simplify the diagram, the wormhole is represented as a single line even though it is a distributed component. Vertical lines from processes to the wormhole correspond to wormhole service invocations that are carried out locally at each node.

The diagram represents three *correct* processes, p_1 through p_3, and one *malicious* process, p_4. Correct processes start by broadcasting their signed values (the **B-value** message). In this scenario, process p_4 did not follow the protocol and only sent its value to process p_1 and to itself. After receiving three (i.e., $2f+1$ with $f=1$) values, a correct process broadcasts a vector containing all the collected signed values (the **B-vector** message) and, then, attempts to propose a hash of a vector to the wormhole. Processes p_1 to p_3 propose, respectively, the vectors of: p_1, p_2 (since p_1 vector got delayed, this is the first nonempty vector in p_2's *array-of-vectors*), and p_1. Process p_4 misbehaves again and proposes a wrong hash in this round. The TBA starts running as soon as it gets three (i.e., $2f+1$) hashes, which means that p_3's proposal is rejected.

The decision of the TBA could either be the hash of p_1 or p_2's vectors or the wrong value of p_4 since all proposals had one vote. Let us assume that the TBA selected p_1's vector. Processes p_1 to p_3 get this result, but, at the same time, they notice that only one vote was cast for this proposal. Therefore, they will continue running the protocol because the quorum necessary for completion is two (i.e., $f+1$) votes. Process p_4 sends a wrong **Decide** message that is not accepted by the other processes.

In the second round, the three correct processes propose the hash of p_2's vector since the *index* variable is 2 for the current round. This time, p_4 chooses again to propose a value, but also a wrong value. In this round, the TBA ends up accepting all proposals because they arrive at approximately the same time. Next, processes determine that the result of the TBA was the hash of p_2's vector and, since there was a sufficiently large quorum (3), they are able to terminate with p_2's vector as the decision. Before returning, the correct processes send the decided vector to process p_4 (in a **Decide** message) because it did not propose the hash of the right vector.

4.3 Implications of the FLP Impossibility Result

Fischer et al. showed that any consensus protocol for asynchronous systems has the possibility of nontermination if a single process is allowed to crash [2]. In the past, basically two solutions have been employed to circumvent this result. The first resorts to the use of randomization [18], [19], and the second extends the basic asynchronous model with some time-related assumptions. These assumptions can be made explicitly [15], [14] or they can be encapsulated in some construct such as an unreliable failure detector [5]. Of course, in the implementation of the failure detector proper, they are necessary to ensure that an expected behavior is observed (e.g., certain completeness and accuracy properties are verified).

The consensus protocol described in the paper runs on the payload part of the system. Therefore, since this part is asynchronous, the protocol could potentially be bound by the FLP impossibility result. Now, just like in an architecture with a failure detector, the system is augmented with a box—the wormhole, that has stronger synchrony assumptions and that provides some simple services. These services can be called by the protocols and, eventually, they return some interesting results. Furthermore, they are sufficiently strong to allow FLP to be circumvented.

The appendix of the paper presents the proofs that the protocol implements all properties of vector consensus, including termination.

5 Expected Performance

This section evaluates the consensus protocol in terms of message and time complexities. It starts by giving a brief overview of the implementation of the wormhole and TBA service and then it analyzes the protocol. For more details on the assumptions and implementation details of the wormhole, we invite the reader to look into the papers [20], [13].

5.1 Wormhole Implementation

The wormhole used in this paper could be constructed in different ways and with distinct levels of coverage of the assumptions. For instance, a high coverage wormhole could be built with dedicated hardware. The implementation of the local wormhole could be done in a special hardware board (e.g., a PC/104 board) with a small and well-defined interface to the rest of the computer. Depending on the location of the nodes, the control channel could be based on a separate network, with messages protected with cryptographic operations.

At the moment, however, we have chosen a distinct approach, which has less coverage, but has the advantage of allowing the free distribution of the wormhole to the research community.[5] The existing prototype is called the Trusted Timely Computing Base (TTCB) and it is based on COTS components. The local TTCB resides inside a real-time kernel, which is hardened in order to be secure, and the control channel uses a dedicated Ethernet network. In relation to the system model that we have described previously, the TTCB offers stronger synchrony properties than what is required. The current implementation ensures that the wormhole is synchronous, and we would only need it to be partially synchronous. Regarding security, the TTCB was built in such a way that, with reasonable coverage, it only fails by crashing, which is exactly what we have assumed.

In the current prototype, each node is composed of standard PC hardware running RTAI, a real-time kernel based on Linux [21]. Two Ethernet networks connect the nodes, one for the payload network and another for the control channel. RTAI modifies the Linux kernel to allow a real-time executive to take control of the node and to enforce real-time behavior on some real-time (RT) tasks. RT tasks are special loadable kernel modules (LKM). Linux executes as the lowest priority task and its interruption scheme was changed to be intercepted by RTAI. The main components of the local TTCB are a library, an LKM, and some RT tasks. The library is linked with the applications and it offers an interface to the services provided by the wormhole. There are, at the moment, two versions of the library, one for C and another for Java applications. RT FIFOs are used to exchange data between the library and the rest of the TTCB. All operations with timeliness constraints are executed by RT tasks. A local TTCB has at least two RT tasks to handle its communication: one to send messages to the other local TTCBs and another to receive and process incoming messages.

From a security perspective, RTAI is quite similar to Linux. The main problem that has to be addressed is the (excessive) privileges of the superuser. A superuser usually has complete control of the node and can change the behavior of any individual resource. Since it is relatively simple to become a superuser once a node is penetrated, some extensions of Linux had to be used to limit its the power. Linux *capabilities* [22] are access control

5. The TTCB wormhole is available at http://www.navigators.di.fc.ul.pt/software/ttcb.

lists associated with processes, allowing a fine grain of control on how certain objects are utilized. At boot time, by removing some capabilities of the superuser, it is possible, for instance, to seal the kernel, i.e., to disable any changes in the kernel memory or the insertion of new code in the kernel.

The TTCB control channel must be protected. This objective is achieved, for instance, by restricting the access to the control channel device driver only to code running inside the kernel (the local TTCB).

5.1.1 TBA Service

Since the TBA service is provided by the TTCB, it can be implemented with a simple agreement protocol tolerant to crash failures and under the synchronous time model. TTCB protocols are built on top of an abstract network (AN) with a number of properties (see [13] for more details). The AN is implemented with a set of simple adaptation mechanisms and is used to keep protocols independent of the control channel network technology. Examples of AN properties are: bounded message transmission delay, bounded *omission degree (Od)*—maximum number of successive omissions in an interval of time, and integrity, confidentiality, and authenticity of transmitted data.

The agreement protocol works in periodic send and receive rounds. When the local TTCB receives a proposal, it puts the value and some control information in a table, awaiting for the beginning of the next send round. In order to tolerate omissions from the network, the messages are multicasted $Od + 1$ times.

The protocol uses a coordinator, typically the local TTCB corresponding to the first eid in $elist$, to select which votes are to be considered. The selection criteria tries to use as many votes as possible, with a minimum defined by $quorum$. Before a decision is taken, the coordinator waits for at least one message from each local TTCB (if they have nothing to send, they transmit an empty message). Therefore, if all processes propose to the TBA at approximately the same time, then all votes are considered. At the end, the coordinator distributes the decided value and the mask *prop-ok* to the local TTCBs to allow the termination of the TBA at every node.

5.2 Time Complexity

This section evaluates the performance of the consensus protocol under two scenarios: failure-free runs and executions with crash and Byzantine failures.

5.2.1 Performance without Failures

The main factors that influence the time complexity of a protocol developed for a Byzantine failure model are the communication delays and the cryptographic calculation costs. The actual values of these overheads, however, are highly dependent of the type of network being employed and on the processing capabilities of the nodes. For instance, on a LAN environment with common PCs, the computation of a signature usually takes much more time than the transmission of a message. On the other hand, the opposite might occur on a WAN setting because network delays can be much larger. Cryptographic operations do not all introduce the same performance penalties—MACs can be calculated quite efficiently, while signatures with asymmetric cryptography can introduce significant overheads [23]. Therefore, in order to make our evaluation as technology independent as possible, we have selected for the analysis a set of metrics that measure the number of times certain operations are executed. These operations correspond to the most important sources of overheads that the protocol will experience.

TABLE 2
Time Complexity Comparison for Best-Case Scenario

Protocol	LatDeg	MSign	GVer	Artifact
DS [3]	5	5	3	Failure detectors
BHRT [4]	3	3	2	Failure detectors
Our protocol	4	1	1	Wormhole

The metric *latency degree* was introduced by Schiper to measure the minimal number of communication steps needed by a protocol to solve a given problem [24]. The basic assumptions made by this metric are: Local processing is negligible (i.e., takes very little time), and message transmission time is constant and equal to a unit of logical time. The specification of the metric is relatively simple and is based on a slight variation of Lamport's logical clocks [25]. The following rules are used to define the latency degree:

1 Send/broadcast/multicast events and local events on a process p_i do not change p_i's logical clock.
2 Let $ts(send(M))$ be the timestamp of the $send(M)$ event; the timestamp carried by message M is defined as $ts(M) = ts(send(M)) + 1$.
3 The timestamp of a $receive(M)$ event on a process p_i is the maximum between $ts(M)$ and the timestamp of the event at p_i immediately preceding the $receive(M)$ event.

Given a run produced by protocol $\mathcal{P}$, the *latency* of $\mathcal{P}$ is defined as the largest timestamp of all *decide* events in this run. The *latency degree* of protocol $\mathcal{P}$ is the minimum latency of $\mathcal{P}$ over all possible runs that can be generated by this protocol.

In the best case, where all processes are correct and execute at the same time, the vector consensus protocol has a latency degree of 2 plus the latency degree of the TBA service of the wormhole. This corresponds to the following scenario: All processes start with a timestamp equal to 0; then, processes broadcast their signed value (carrying a timestamp of 1) and wait for the arrival of $2f + 1$ values; next, they broadcast their vectors (carrying a timestamp of 2); processes receive the vector of p_1 and propose a hash of it to the TBA service (at this moment, the local timestamp of all processes is 2); the TBA includes every vote in the decision because processes are proposed at the same instant; since each process gets the same decision, they can all terminate immediately, with a timestamp of 2 plus the latency of the TBA.

If one applies the above rules to the current implementation of the TBA service, the resulting latency is 2. Consequently, the overall latency degree of the vector consensus protocol is 4. Table 2 compares the latency degree (*LatDeg* column) of our protocol with the two other solutions for vector consensus. Both protocols by Doudou and Schiper (DS) [3] and Baldoni et al. (BHRT) [4] rely on Byzantine failure detectors. In the calculations, we have assumed that failure detectors make no mistakes (i.e., they behave as a perfect failure detector).

The metrics *message signature degree* and *group verification degree* are used to measure the number of times certain cryptographic operations are executed. These operations were selected because they are the only ones that introduce relevant processing delays in the protocols under study. The following rules define these operations:

- Message signature: The sender process signs a message or some fields of a message and the receiver verifies the signature when the message arrives.
- Group verification: At some point during the execution of a process, the signatures of a group of (at least $2f+1$) data elements are verified.

Given the run of protocol $\mathcal{P}$ with minimum latency (which corresponds to the latency degree), the *message signature degree* counts the number of message signature operations executed in the critical path of the run. Likewise, the *group verification degree* counts the number of group verification operations.

Our protocol uses a single message signature when processes transmit their values in the **B-value** message. Messages sent through the payload channel are protected with MACs that can be efficiently calculated (see Section 2). A group verification is performed whenever a process selects a good vector in function *getNextGoodVector()*.

Protocols DS and BHRT sign all messages exchanged among processes and add to certain messages a certificate. Certificates are used by the failure detectors to determine if processes are behaving in a malicious manner. They contain a number of previously observed messages (at least $\lceil (2n+1)/3 \rceil$), whose signatures have to be verified whenever certificates are received. Therefore, the arrival of a certificate corresponds to a group verification operation.

Table 2 also presents the message signature and group verification degrees of the three protocols (*MSign* and *GVer* columns). If one takes into consideration both the communication delays and the cryptographic overheads, in a LAN setting and with the typical processing capabilities of the current nodes, our protocol will perform better than previous solutions.

5.2.2 Performance with Failures

The vector consensus protocol was designed to tolerate intrusions in some processes. Therefore, it is important to understand the behavior of the protocol in some failure scenarios. In particular, we will determine the worst performance of the protocol when there are f failed processes at the beginning of the run. Throughout the analysis, we will assume that the network works correctly and that correct processes start to execute at the same time. Table 3 presents the performance of our protocol when the f processes are crashed and when they act in a Byzantine way.

In the crash failure scenario, our protocol has the same time complexity as in the best case run. Function *getNextGoodVector()* disregards the empty vectors and processes simply decide on the vector of the first correct process. Consequently, no performance is lost. Protocols DS and BHRT have a cost associated with skipping a failed coordinator. Two or one messages, respectively, for each protocol, have to be exchanged among correct processes to ensure that a new round is initiated with a different coordinator. In the worst case, the first f coordinators have all crashed, which introduces some overheads.

TABLE 3
Worst Time Complexity for a Scenario with f Failures

	Protocol	LatDeg	MSign	GVer	SDeg
Crash	DS [3]	5 + 2f	5 + 2f	3 + f	0
	BHRT [4]	3 + f	3 + f	2 + f	0
	Our protocol	4	1	1	0
Byzantine	DS [3]	5 + 2f	5 + 2f	3 + f	f
	BHRT [4]	3 + f	3 + f	2 + f	f
	Our protocol	4 + 2f	1	1 + f	0

Since Byzantine processes cannot break the properties of vector consensus, the worst action they can perform is to delay (as much as possible) the execution of the protocols without being discovered. For all three protocols, the most damaging situation corresponds to a setting where the first f processes are malicious. In our protocol, each bad process can postpone the completion of consensus by sending contradictory vectors to the correct processes, which results in an inconclusive TBA return value. The cost associated with trying the vector of the next process is a TBA execution.

The two main overheads that can be introduced by a malicious process in the DS and BHRT protocols are the need for extra messages to skip the coordinator and a suspicion delay. The *suspicion delay* is defined as the interval necessary for a failure detector to start suspecting that a process has failed. A malicious coordinator process can perform the following simple attack to delay the computation: It fakes a congested network by stopping message sends and, then, when enough processes suspect its failure, it reinitiates communication. With this procedure, the process forces the execution of a new round, without being detected as malicious and, at the same time, it maximizes the duration of the current round. A failure detector that tries to reduce the number of the false suspicions necessarily has a suspicion delay (several times) larger than the typical message transmission interval. Table 3 also presents a *suspicion degree* that counts the number of suspicions delays introduced by malicious processes in the critical path of the minimum latency run (column *SDeg*).

5.3 Message Complexity

It is relatively simple to see that, in the best case, the number of messages transmitted through the payload network is $O(n^2)$ if point-to-point communication is used and $O(n)$ if broadcast communication is available.

6 Related Work

Throughout the years, several agreement problems have been proposed and solved in the literature (a few examples can be found in [26], [5], [27], [28]). These problems basically allow processes to reach consensus on a value or set of values despite the existence of a number of faulty processes. Depending on the system model, in particular on the synchrony assumptions, it has been shown that, in some cases, it is impossible to deterministically solve consensus (see, for instance, [29]). Therefore, several ways have been proposed to circumvent these impossibility results, such as randomization [19], [18] and failure detectors [5].

Dwork et al. [14] studied how Byzantine consensus could be solved in different kinds of partially synchronous systems. Protocols were based on the rotating coordinator paradigm, where, in each round, a process attempts to impose its value as decision. Some of the protocols had to utilize signatures with asymmetric cryptography.

More recently, a few consensus protocols that rely on Byzantine failure detectors have been proposed [6], [7], [11], [3], [4]. All these protocols employ the rotating coordinator paradigm and use failure detectors to find out if the current coordinator is preventing progress. If this is the case, then processes cooperate to initiate a new round with a different coordinator. Message signatures and certificates help failure detectors to find out malicious actions during the processes execution.

Vector consensus has some similarities to the interactive consistency problem [30] that has been extensively studied in the literature. The main requirement where they differ is on the number of initial values of correct processes that the vector must contain—in one case, it has to include a majority and, in the other, all values. Most of the work on interactive consistency was done in synchronous systems, while vector consensus has been solved in asynchronous environments.

To our knowledge, only two protocols have been specifically proposed to solve vector consensus [3], [4]. Doudou and Schiper described a protocol based on two failure detectors, called Mute and Byzantine, respectively [3]. The first detector finds out which processes refuse to send messages either benignly (due to a crash) or maliciously, and the second detector looks for incorrect behavior during the protocol execution. The protocol is based on a rotating coordinator scheme where, in each round, one process proposes a vector that the others confirm and accept. If the coordinator is suspected, then a new round is initiated with a different coordinator. Baldoni et al. described a vector consensus protocol with two failure detectors that have comparable functions as the ones mentioned above [4]. The protocol is also based on a rotating coordinator, but the organization of each round is quite different both in terms of messages and information exchanged.

Our architecture for an asynchronous system augmented with a wormhole was first described when we presented the *Timely Computing Base (TCB)* [31]. This wormhole mainly provided services related to time (e.g., measurement of durations). Later, we extended the TCB so that it could be used to support Byzantine resilient applications by providing a limited number of security related services [13]. This new wormhole was called Trusted Timely Computing Base. Recently, we found out about another work that defines an architecture with constructs that are somewhat similar to our wormholes. Baldoni et al. [32] use a sequencer to solve an agreement problem on a three tier replicated service. The architecture basically assumes a crash asynchronous model with the exception of the sequencer, which is built with partially synchronous assumptions. Wormholes are a quite generic concept that could also be applied in a crash failure scenario to provide either sequencing or failure detection services, which could be used to circumvent certain impossibility results, such as FLP.

The protocol presented in this paper uses a wormhole to solve vector consensus. Instead of relying on failure detectors to determine which processes are malicious, it uses a low-level agreement service provided by a wormhole. Therefore, it does not exclude any processes from the computation and uses all processes that behave correctly at any given time (including malicious process that alternate between bad and good behavior). Furthermore, on Byzantine systems, it is relatively difficult to construct a failure detector and, in fact, at this moment, it can be considered as an open problem. On the other hand, it is quite feasible to implement an agreement protocol in a synchronous secure system like the TTCB.

7 Conclusion

The paper presents a new approach to solve vector consensus in a Byzantine asynchronous system augmented with a wormhole. In this approach, the protocol is executed by a group of processes running in the payload part of the system, where they might experience undefined delays and Byzantine failures. During the execution, however, they use a low-level agreement service provided by a wormhole. Since the wormhole is both secure and partially synchronous, there is the guarantee that eventually a good result will be returned by this service.

The proposed protocol has three interesting features. First, it demonstrates how a wormhole can be used to solve vector consensus in an asynchronous Byzantine setting. Second, it shows that it is possible to keep the protocol completely time-free and circumvent the FLP impossibility result by moving all synchrony assumptions to the wormhole. The only hypothesis one needs to make is to assume that, eventually, the wormhole services will finish and return the expected results. Third, the protocol exhibits a good time complexity if one considers both the communication delays and cryptographic operations.

Appendix

Correctness Proofs

This section proves the correctness of the protocol presented in Algorithms 1 and 2. The protocol is correct if it satisfies the three properties of the vector consensus specification—Validity, Agreement, and Termination. We will consider a system model as defined in Section 2, where at most there are f malicious processes out of a total number of $n \geq 3f + 1$.

Lemma 1. *Correct processes only propose to the TBAs hashs of vectors v that have at least $f+1$ values from correct processes and, for any correct p_i, $v[i] = value_i$ or $v[i] = \bot$.*

Proof. A correct process p_i starts by setting all entries of *array-of-vectors* to $\bot$ (Line 5). Next, it signs $value_i$ using asymmetric cryptography (Line 6) and saves the signed value in the i's entry of its vector *array-of-vectors[i]* (Line 7). Then, it broadcast the signed value using secure channels (Line 8).

Process p_i waits for the arrival of $2f+1$ signed values from different processes and saves them in its vector (Lines 10-13). Before inserting an entry in the vector, it verifies that the signature was correctly made. Therefore, the vector produced by p_i will have $2f+1$ entries filled with correctly signed values.

Next, the process broadcasts the vector through secure channels and initiates the two parallel tasks (Lines 14-15). Task T1 stores the arriving vectors in *array-of-vectors* (Lines 17-18). Due to the Integrity property of secure channels (Section 2), the sender of the vector is correctly identified and the vector contents are received without changes. Task T2 calls the *getNextGoodVector()* function to obtain a vector v whose hash is going to be proposed to the TBA (Lines 28-29).

Function *getNextGoodVector()* goes through *array-of-vectors* until it finds a vector that verifies the following conditions: 1) Each entry different from $\bot$ is correctly signed (Lines 4-5 of Algorithm 2); 2) there are at least $2f+1$ of such entries (Line 7 of Algorithm 2). The returned vector, *new-vector*, is built by first making all entries equal to $\bot$ and then by copying the values from the selected vector of *array-of-vectors* (Lines 3 and 6 of Algorithm 2). Therefore, *new-vector* will satisfy the conditions of this lemma because there are at most f malicious processes and it has $2f+1$ entries filled with the initial values. The rest of the entries are set to $\bot$. □

Lemma 2. *All correct processes participate in the same TBAs and get identical results in each TBA execution.*

Proof. Correct processes initiate the consensus protocol with the same list of processes, $elist$, and consensus identifier, cid.

All correct processes broadcast a **B-value** message (Line 8) and then they wait for the arrival of $2f+1$ **B-value** messages from different processes (Lines 9-13). Since there are at least $2f+1$ correct processes and since messages are transmitted through secure channels (see Section 2), eventually they will receive enough messages that will allow them to continue the protocol. Next, processes broadcast a **B-vector** message, and they start the two parallel tasks (Lines 14-15).

The following occurs for every task T2 of any correct process:

1. In each k execution of the loop (Line 24), task T2 increments the $round$ variable and then it uses a deterministic function to generate the agreement identifier, $agid$ (Lines 26-27). Next, it calls W_TBA with $agid$ and $elist$ (Line 29), which are identical in all correct processes. Therefore, all correct processes participate in the same TBA and, due to the Agreement property of the TBA service, they also get equivalent results—*out.error*, *out.prop-ok*, and *out.value*.
2. After collecting the result from W_TBA, task T2 finds out if *out.value* is supported by $f+1$ processes using the mask *out.prop-ok* (Line 30). In the affirmative case, all T2 tasks stop running, either in Lines 34 or 37. Otherwise, they initiate the $k+1$ execution of the loop. Therefore, all correct processes run the same number of TBAs, and conclude the execution of the loop in the same round. □

Lemma 3. *If a correct process decides, then eventually every correct process will decide.*

Proof. Let p_i be a correct process deciding at line 34 of Algorithm 1. This process sends a **Decide** message containing the selected vector to all processes that did not propose the correct hash to the wormhole, using the *prop-ok* mask to find out who belongs to this group (Lines 32-33).

Let p_k be a correct process that has not yet decided. This process gets the same result from the TBA as process p_i (Lemma 2). Therefore, one of two cases will occur:

1. p_k proposed a hash equal to the one that was decided by p_i. This process will also decide at Line 34.
2. p_k proposed a hash different than the one that was decided. Process p_k saves the hash returned by the TBA in variable *hash-v* and terminates task T2 (Lines 36-37). In parallel, it waits for the arrival of **Decide** messages and saves the vectors in the bag *bag-decide* (Lines 19-20). Eventually, the **Decide** message sent by p_i will be received since it was transmitted through a secure channel (see Section 2). When this happens, process p_k is able to decide at Line 22. □

Theorem 1 (Validity). *Every correct process decides on a vector, vect, of size n, such that:*

1. *For every $1 \leq i \leq n$, if process p_i is correct, then vect[i] is either the initial value of p_i or the value $\bot$ and*
2. *at least $f+1$ elements of the vector vect are the initial values of correct processes.*

Proof. A correct process only decides on a vector in two situations. We prove that, in both, the decided vector satisfies the two properties of the theorem:

1. It proposes a hash of a vector v to the TBA and this hash is supported by at least $f+1$ (malicious or correct) processes (Lines 29-34). By Lemma 1, a correct process only proposes to the wormhole TBA a hash of a vector v satisfying the conditions of this theorem.
2. It receives a **Decide** message containing a vector with the same hash as *hash-v* (Lines 19-22). Hash

functions satisfy the weak and strong collision resistance properties.[6] It is therefore computationally infeasible for an adversary to produce a bad vector with an hash equal to a given *hash-v*. Consequently, the adversary cannot send a **Decide** message containing a wrong vector and make the process incorrectly decide on the vector.

Variable *hash-v* is initialized to $\perp$ (Line 3) and is set to *out.value* if this value was proposed by more than $f+1$ processes in one TBA execution (Lines 30 and 36). Since there are at most f malicious processes, then at least one correct process proposed this value. Due to Lemma 1, the hash corresponding to this value was calculated using a vector v satisfying the conditions of this theorem. □

Theorem 2 (Agreement). *No two correct processes decide differently.*

Proof. By applying Lemma 2, all correct processes execute the same TBAs and get identical results. Moreover, once there is a TBA with a value which has at least $f+1$ votes (Line 30), no more rounds of the loop are executed because either processes decide (Line 34) or they stop the task T2 (Line 35). This value, or hash of a vector, was proposed by at least one correct process since there are at most f malicious processes.

Now, let p_i and p_j be two correct processes that decide vectors v_i and v_j, and let $hashDecision$ be the value returned by the TBA with $f+1$ or more votes. There are three cases:

1. Both decide at line 34. In this case, we must have $Hash(v_i) = Hash(v_j) = hashDecision$. Then, $v_i = v_j$ due to the collision resistance properties of the hash function.
2. Both decide at line 22. In this case, we must have $Hash(v_i) \neq hashDecision$ and $Hash(v_j) \neq hashDecision$. Nevertheless, both processes save $hashDecision$ in variable *hash-v* (Line 36) and wait for the arrival of a **Decide** message carrying a $vector_k$ whose hash is equal to *hash-v* (Lines 19-22). Also, due to the collision resistance of the hash function, both processes will return an equivalent $vector_k$.
3. One decides at Line 34 and another at Line 22 (respectively, p_i and p_j without loss of generality). In this case, we must have

$$Hash(v_i) = hashDecision$$

and $Hash(v_j) \neq hashDecision$. Using arguments similar to 1 and 2, process p_i will decide v_i and process p_j will select a $vector_k$ with a hash equal to $hashDecision$. Both vectors will be identical due to the collision resistance properties of the hash functions. □

6. The weak collision resistance property requires that, given X, it is computationally infeasible to find a Y such that hash(Y) = hash(X). The strong collision resistance property says that it is computationally infeasible to find a pair (X, Y) such that hash(X) = hash(Y) [17].

Theorem 3 (Termination). *Every correct process eventually decides.*

Proof. By applying Lemma 3, once a correct process decides, the rest will also do the same. Consequently, we only need to prove that at least one correct process will decide.

A correct process is able to decide if the hash of its vector is also proposed by f other (correct or malicious) processes (Lines 30-34). This condition is satisfied if the protocol guarantees that a vector will sooner or later get $f+1$ or more votes in a TBA.

A process starts by constructing a vector with the digitally signed initial values of $2f+1$ different processes. This vector will eventually be built because there are at least $2f+1$ correct processes and communication channels are secure (and, therefore, they have the eventual reliability property) (Lines 6-13). Next, it broadcasts the vector and initiates tasks T1 and T2 (Lines 14-15).

Task T1 receives the **B-vector** messages and stores the arriving vectors in *array-of-vectors* (Lines 17-18). Consequently, since all correct processes are doing the same, there will be a time when *array-of-vectors* at each process will contain one vector from every correct process (plus a few others from malicious processes). Each of these vectors can potentially be returned by *getNextGoodVector()*.

Function *getNextGoodVector()* picks, in a round-robin fashion, a vector from each process in the system. The selection criteria is deterministic and is based on the current round number—the $index$ variable (Line 25 and Line 4 of Algorithm 2). Therefore, it will eventually choose a vector from a correct process, and this vector will be supported by all other correct processes. Since the TBA *quorum* is $2f+1$ and there are at most f malicious processes, this vector will receive $f+1$ or more proposals.

The TBA service has the termination property, which ensures that, sooner or later, a result is returned and, in this case, it will allow the completion of the protocol because the hash of the vector will have at least $f+1$ votes. □

Acknowledgments

This work was partially supported by the FCT through projects POSI/CHS/39815/2001 (COPE) and POSI/EIA/60334/2004 (RITAS) and the Large-Scale Informatic Systems Laboratory (LASIGE).

References

[1] R. Guerraoui and A. Schiper, "Consensus Service: A Modular Approach for Building Fault-Tolerant Agreement Protocols in Distributed Systems," *Proc. 26th IEEE Int'l Symp. Fault-Tolerant Computing Systems,* pp. 168-177, June 1996.

[2] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, Apr. 1985.

[3] A. Doudou and A. Schiper, "Muteness Failure Detectors for Consensus with Byzantine Processes," Technical Report 97/30, École Polytechnique Fédérale de Lausanne, 1997.

[4] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy, "Consensus in Byzantine Asynchronous Systems," *Proc. Int'l Colloquium Structural Information and Comm. Complexity,* pp. 1-16, June 2000.
[5] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, pp. 225-267, Mar. 1996.
[6] D. Malkhi and M. Reiter, "Unreliable Intrusion Detection in Distributed Computations," *Proc. 10th Computer Security Foundations Workshop,* pp. 116-124, June 1997.
[7] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith, "Byzantine Fault Detectors for Solving Consensus," *The Computer J.,* vol. 46, no. 1, pp. 16-35, Jan. 2003.
[8] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness Failure Detectors: Specification and Implementation," *Proc. Third European Dependable Computing Conf.,* pp. 71-87, Sept. 1999.
[9] K. Julisch and M. Dacier, "Mining Intrusion Detection Alarms for Actionable Knowledge," *Proc. Eighth ACM Int'l Conf. Knowledge Discovery and Data Mining,* pp. 366-375, July 2002.
[10] H. Debar and A. Wespi, "Aggregation and Correlation of Intrusion-Detection Alerts," *Proc. Fourth Int'l Symp. Recent Advances in Intrusion Detection,* 2001.
[11] A. Doudou, B. Garbinato, and R. Guerraoui, "Encapsulating Failure Detection: From Crash-Stop to Byzantine Failures," *Proc. Int'l Conf. Reliable Software Technologies,* pp. 24-50, June 2002.
[12] P. Veríssimo, "Uncertainty and Predictability: Can They Be Reconciled?" *Future Directions in Distributed Computing,* pp. 108-113, 2003.
[13] M. Correia, P. Veríssimo, and N.F. Neves, "The Design of a COTS Real-Time Distributed Security Kernel," *Proc. Fourth European Dependable Computing Conf.,* pp. 234-252, Oct. 2002.
[14] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM,* vol. 35, no. 2, pp. 288-323, Apr. 1988.
[15] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM,* vol. 34, no. 1, pp. 77-97, Jan. 1987.
[16] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 6, pp. 642-657, 1999.
[17] A.J. Menezes, P.C.V. Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.
[18] M.O. Rabin, "Randomized Byzantine Generals," *Proc. 24th Ann. IEEE Symp. Foundations of Computer Science,* pp. 403-409, Nov. 1983.
[19] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proc. Second ACM Symp. Principles of Distributed Computing,* pp. 27-30, Aug. 1983.
[20] A. Casimiro, P. Martins, and P. Veríssimo, "How to Build a Timely Computing Base Using Real-Time Linux," *Proc. IEEE Int'l Workshop Factory Comm. Systems,* pp. 127-134, Sept. 2000.
[21] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour, "DIAPM-RTAI Position Paper," *Real-Time Linux Workshop,* Nov. 2000.
[22] B. Tobotras, "Linux Capabilities FAQ 0.2," 1999.
[23] M. Castro and B. Liskov, "Proactive Recovery in a Byzantine-Fault-Tolerant System," *Proc. Fourth Symp. Operating Systems Design and Implementation,* pp. 273-288, Oct. 2000.
[24] A. Schiper, "Early Consensus in an Asynchronous System with a Weak Failure Detector," *Distributed Computing,* vol. 10, pp. 149-157, Oct. 1997.
[25] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.
[26] G. Bracha and S. Toueg, "Asynchronous Consensus and Broadcast Protocols," *J. ACM,* vol. 32, no. 4, pp. 824-840, Oct. 1985.
[27] C. Fetzer and F. Cristian, "On the Possibility of Consensus in Asynchronous Systems," *Proc. Pacific Rim Int'l Symp. Fault-Tolerant Systems,* Dec. 1995.
[28] A. Mostefaoui, S. Rajsbaum, and M. Raynal, "Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems," *Proc. 33rd ACM Symp. Theory of Computing,* pp. 152-162, July 2001.
[29] N. Lynch, "A Hundred Impossibility Proofs for Distributed Computing," *Proc. Eighth Ann. ACM Symp. Principles of Distributed Computing,* pp. 1-28, Aug. 1989.
[30] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM,* vol. 27, no. 2, pp. 228-234, 1980.
[31] P. Veríssimo, A. Casimiro, and C. Fetzer, "The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 533-542, June 2000.
[32] R. Baldoni, C. Marchetti, and S. Piergiovanni, "Asynchronous Active Replication in Three-Tier Distributed Systems," *Proc. Ninth Pacific Rim Int'l Symp. Dependable Computing,* Dec. 2002.

Nuno F. Neves received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1998. He is an assistant professor of the Department of Informatics, University of Lisboa. His research interests are in parallel and distributed systems, in particular in the areas of security and fault tolerance. His work has been recognized with a Fulbright Fellowship during his doctoral studies and with the William C. Carter Best Student Paper award at the 1998 IEEE International Fault-Tolerant Computing Symposium. More information about him is available at http://www.di.fc.ul.pt/~nuno. He is a member of the IEEE.

Miguel Correia received the PhD degree in computer science from the University of Lisboa in 2003. He is an assistant professor in the Department of Informatics, University of Lisboa Faculty of Sciences. He is a member of the LASIGE laboratory and the Navigators research group. He was involved in the EC-IST MAFTIA project and the FCT DeFeATS project, both in the area of fault and intrusion tolerance. More information about him is available at http://www.di.fc.ul.pt/~mpc. He is a member of the IEEE and the IEEE Computer Society.

Paulo Veríssimo is currently a professor in the Department of Informatics, University of Lisboa Faculty of Sciences (http://www.di.fc.ul.pt/~pjv). He has coordinated the CORTEX IST/FET project and belonged to the executive board of the CaberNet European NoE. He is a past chair of the IEEE Technical Committee on Fault-Tolerant Computing and of the steering committee of the DSN Conference and is currently a member of the European Security and Dependability Task Force Advisory Board. He leads the Navigators research group of LASIGE and is currently interested in architecture, middleware, and protocols for distributed, and pervasive and embedded systems, namely, the facets of adaptive real-time and fault/intrusion tolerance. He is the author of more than 100 refereed publications in international scientific conferences and journals in the area and he is a coauthor of four books. He is a member of the IEEE and the IEEE Computer Society.